Robust Identification of Fuzzy Duplicates

Surajit Chaudhuri Venkatesh Ganti Rajeev Motwani
Microsoft Research *Stanford University*
{surajitc, vganti}@microsoft.com *rajeev@cs.stanford.edu*

Abstract

Detecting and eliminating fuzzy duplicates is a critical data cleaning task that is required by many applications. Fuzzy duplicates are multiple seemingly distinct tuples which represent the same real-world entity. We propose two novel criteria that enable characterization of fuzzy duplicates more accurately than is possible with existing techniques. Using these criteria, we propose a novel framework for the fuzzy duplicate elimination problem. We show that solutions within the new framework result in better accuracy than earlier approaches. We present an efficient algorithm for solving instantiations within the framework. We evaluate it on real datasets to demonstrate the accuracy and scalability of our algorithm.

1. Introduction

Detecting and eliminating duplicated data is an important problem in the broader area of data cleaning and data quality. For example, when Lisa purchases products from SuperMart twice, she might be entered as two different customers, e.g., [Lisa Simpson, Seattle, WA, USA, 98025] and [Simson Lisa, Seattle, WA, United States, 98025]. Many times, the same logical real world entity has multiple representations in a relation, due to data entry errors, varying conventions, and a variety of other reasons. Such duplicated information can cause significant problems for the users of the data. For example, it can lead to increased direct mailing costs because several customers like Lisa may be sent multiple catalogs. Or, such duplicates could cause incorrect results in analytic queries (say, the number of SuperMart customers in Seattle), and lead to erroneous data mining models. Hence, a significant amount of time and money are spent on the task of detecting and eliminating duplicates. We refer to this problem of detecting and eliminating multiple *distinct* records representing the *same* real world entity or phenomenon as the *fuzzy duplicate elimination* problem. This problem is similar to the merge/purge, deduping, and record linkage problems [e.g., 13, 17, 15, 21, 26, 1, 25]. Note that our problem generalizes the standard duplicate elimination problem for answering "select distinct" queries in relational database systems, which consider two tuples to be duplicates if they match exactly on all attributes [3]. However, in this paper, we use duplicate elimination to mean fuzzy duplicate elimination.

Previous solutions (as discussed further in Section 6) to duplicate elimination can be classified into supervised and unsupervised approaches. Supervised approaches learn rules characterizing pairs of duplicates from training data consisting of known duplicates [11, 5, 26, 28]. These approaches are limited by their dependence on comprehensive training data which exhibit the variety and distribution of errors observed in practice, or on manual guidance. In many real data integration scenarios, it is not possible to obtain good training data or interactive user guidance. Therefore, we focus on unsupervised approaches.

Unsupervised approaches for duplicate elimination typically rely on distance functions to measure similarity between tuples. Previous approaches adopted clustering algorithms to partition a relation into groups of duplicates [e.g., 15, 21]. In particular, the single linkage clustering algorithm has been the predominant choice as it automatically determines the number of clusters (or the number of groups of duplicates) unlike many clustering algorithms, which require the number of clusters to be specified. However, the single linkage clustering approaches require as a parameter an *absolute global* threshold to decide when two tuples are duplicates. Distance functions are mathematical approximations to an abstract and qualitative notion of duplication. Even in a relation consisting only of clean tuples, some tuples are inherently close to each other according to a reasonable distance function, even though they are not duplicates. Similarly, distance between duplicate tuples may higher than several distinct pairs of tuples. For the example in Table 1, duplicate tuples *3* and *4* are farther (according to edit distance) from each other than distinct tuples *9* and *10*. Therefore, approaches based on global distance thresholds lead to poor *recall* (fraction of pairs of true duplicate tuples an algorithm recognizes as duplicates) and *precision* (fraction of tuple pairs the algorithm returns which are truly duplicates).

The crucial characteristic differentiating the duplicate elimination problem from the standard clustering formulations is that it is very important to consider local structural properties while identifying groups of

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 **IEEE**

duplicates. In this paper, we identify two new criteria—the *compact set* (CS) and the *sparse neighborhood* (SN) criteria—which explicitly capture local structural properties of data to characterize groups of duplicate tuples. These criteria capture the properties that duplicates in a group are closer to each other than to other tuples, and that their "local neighborhood" is empty or sparse. Informally, the local neighborhood of a group of tuples is the immediate vicinity defined in terms of a surrounding region of size dependent on the local distribution of tuples, viz., a sphere of radius twice (say) the *nearest neighbor* distance of each tuple (as illustrated in Figure 1). These localized structural properties differentiate the duplicate elimination problem from standard clustering formulations. Tuples that satisfy these criteria may be grouped together as duplicates even though they are relatively far from each other while tuples that are closer but do not satisfy these criteria may not be grouped together. Note that the CS and SN criteria are orthogonal to the choice of specific distance functions and domain knowledge such as abbreviations. Therefore, our approach is orthogonal to these choices.

We formalize the duplicate elimination problem to effectively exploit the CS and SN criteria and show that solutions to the formulations within our framework have several desirable characteristics: varying the scale of the distance function does not change the solution; the range of partitions that a duplicate elimination problem can produce is large. Our analysis here is similar in spirit to Kleinberg's development of an axiomatic framework for clustering [18].

We present a scalable and efficient algorithm for our formulation of duplicate elimination. Our algorithm exploits nearest neighbor indexes over distance functions as well as the database server to achieve scalability and efficiency. In an extensive experimental study over several real datasets, we evaluate the CS and SN criteria and show that solutions to the duplicate elimination problems within our framework yield better precision-recall tradeoffs than existing single linkage approaches with global thresholds.

The remainder of the paper is organized as follows. In Section 2, we formally define the CS and SN criteria. In Section 3, we formalize the duplicate elimination framework. In Section 4, we describe an efficient algorithm for solving instantiations within this framework. In Section 5, we discuss a thorough experimental evaluation. In Section 6, we discuss related work, and we conclude in Section 7.

2. Criteria Characterizing Duplicates

In this section, we define the CS and SN criteria which are useful for characterizing duplicates. The intuition behind these two criteria stems from the following observations: (i) duplicate tuples are "closer" to each other than to others; and, (ii) the local neighborhood of duplicate tuples is sparse. In order to illustrate that these criteria are better at characterizing duplicate tuples, let us consider Table 1 which shows an example drawn from a real music database. The first six tuples (tagged with an asterisk) are duplicate tuples while the remaining tuples are unique. Observe that some pairs (e.g., 7 and 8, 9 and 10) among unique tuples are closer to each other according to standard distance functions (edit distance, or cosine metric), than some of the pairs among the six duplicate tuples. Therefore, the traditional threshold-based approach for duplicate elimination cannot correctly distinguish the set of duplicates without simultaneously collapsing unique tuples together.

ID	ArtistName	TrackName
1*	The Doors	LA Woman
2*	Doors	LA Woman
3*	The Beatles	A Little Help from My Friends
4*	Beatles, The	With A Little Help From My Friend
5*	Shania Twain	Im Holdin on to Love
6*	Twian, Shania	I'm Holding On To Love
7	4th Elemynt	Ears/Eyes
8	4th Elemynt	Ears/Eyes - Part II
9	4th Elemynt	Ears/Eyes - Part III
10	4th Elemynt	Ears/Eyes - Part IV
11	Aaliyah	Are You Ready
12	AC DC	Are You Ready
13	Bob Dylan	Are You Ready
14	Creed	Are You Ready

Table 1: Examples from a media database.

The intuition behind the compact set (CS) criterion is that duplicate tuples are closer to each other than they are to other distinct tuples. That is, duplicate tuples are usually mutual nearest neighbors. For the example in Table 1, tuples with duplicates are tagged with asterisks. Tuple 1 is the nearest neighbor of tuple 2 and vice-versa. In contrast, tuple 8 may be the nearest neighbor of tuple 7 and tuple 9 that of tuple 8. (The nearest neighbor relation is not symmetric.) Reflecting this intuition, our first criterion is that a set of duplicates must be a compact set of mutual nearest neighbors; note that the set may consist of more than two tuples. In contrast, global threshold approaches based on single linkage clustering assume transitivity (i.e., if 'a' is a duplicate of 'b'—$d(a,b) < \theta$—and 'b' that of 'c' then 'a' is a duplicate of 'c') and identify connected components in the *threshold-graph*. In the threshold-graph, each tuple in the relation corresponds to a node; two nodes are connected by an edge if the distance between corresponding tuples is less than the threshold. Hence, global threshold approaches based on single linkage clustering are more likely to yield a large number of false positives. Observe that at an appropriate

threshold (the maximum pair wise distance within a group) a compact set forms a clique in the threshold-graph. The key requirement is that the threshold needs to be varied across groups depending on the local neighborhood, and this variability is crucial for accurately characterizing duplicates. Therefore, earlier approaches based on single linkage clustering are limited because they do not allow such variability.

The CS criterion alone is not sufficient to characterize groups of duplicates because several pairs of tuples in a clean relation (consisting of unique tuples) may still be mutual nearest neighbors and hence compact. In the extreme case, the set of all tuples in a relation forms a compact set. We also require the *sparse neighborhood (SN)* property to better characterize groups of duplicates. The observation behind the SN criterion is that the local neighborhood of a tuple with duplicates is sparse. Alternatively, tuples which have a lot of tuples within their close vicinity are usually not duplicates of other tuples. For the example in Table 1, the unique tuples 7-14 occur in larger (4, for this example) groups than tuples with duplicates. Figure 1 illustrates the idea behind sparse neighborhood – the region within a sphere of radius *2·nn(v),* where *nn(v)* is the nearest neighbor distance around the tuple v. If the number of tuples in the larger sphere around a tuple is small, we say that its local neighborhood is sparse. We extend this notion to a group of tuples and say that their joint local neighborhood is sparse if an *aggregate* of individual growth rates of tuples is small, say, less than a threshold *c*. For instance, the aggregation function *max* requires the neighborhood sizes of all tuples in the group to be less than c. Observe that the neighborhood size of a tuple is a property of the domain (not the specific relation instance) to which tuples belong. We use the neighborhood sizes of a tuple in a relation as a directly computable approximation of the ideal measure.

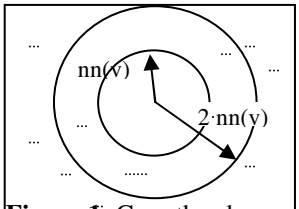

Figure 1: Growth spheres

We now formalize the CS and SN criteria. In the following definitions, let *R* be a relation and *d: R x R → [0, 1]* be a symmetric distance function over tuples in R. For expositional clarity, we assume that all tuples in a relation are unique and distances between distinct tuples are non-zero.

CS Criterion: We say that a set S of tuples from R is a *compact set* iff for every tuple v in S, the distance *d(v, v')* between *v* and any other tuple *v'* in *S* is less than the distance *d(v, v'')* between *v* and any other *v'' in R−S*.

Observe that if the distance function *d* is *well-behaved* in that a tuple is closer to its (fuzzy) duplicates than to other distinct tuples, then each group of duplicate tuples in a relation *R* is a compact set.

We now introduce additional terminology to define the sparse neighborhood criterion. For a tuple v, let *nn(v)* denote the distance between v and its nearest neighbor. Let the *neighborhood N(v)* be defined by a sphere of radius *p·nn(v)* around *v*. In this paper, we fix p=2; functions more general than linear functions may be used to define neighborhood. The *neighborhood growth ng(v)* is the number of tuples in the neighborhood N(v) of v. Formally, $ng(v) = |\{u \mid d(u, v) < p \cdot nn(v)\}|$

SN Criterion: Let *AGG : $2^R \rightarrow R$* be an aggregation function and *c* (*>0*) be a constant. We say that a set of tuples S is an *SN(AGG, c)* group if (i) |S| = 1, or (ii) the aggregated value of neighborhood growths of all tuples in S is less than c; i.e., *AGG({ng(v): v in S}) < c*.

In this paper, we only consider the *max* and the *average* aggregation functions.

3. The Duplicate Elimination Problem

We first propose an initial formulation based only on the CS and SN criteria and show that it can sometimes lead to unintuitive solutions. Therefore, we add standard constraints that sizes of groups of duplicates are small or that the maximum distance between tuples in a group is bounded. We then show that the resulting duplicate elimination problem yields a unique solution, and also discuss other interesting properties.

The Duplicate Elimination (DE) Problem—Initial Formulation: Given a relation R, a distance function d, an aggregation function AGG, and a positive real number c, partition R into the *minimum* number of groups $\{G_1,\ldots,G_m\}$ such that for all $1 \le i \le m$ (i) G_i is a compact set, and (ii) G_i is an SN(AGG, c) group.

To illustrate that the above formulation may result in unintuitive results, consider applying the DE formulation as stated above on the following instance *R* of positive integers: {101, 102, 104, 201, 202, 301, 302}. Suppose the distance function is the absolute difference between values, the aggregation function is the *max* function, and the SN threshold *c* is set to 2. Then, all tuples in R are all put together in one group. Ideally, we may want 3 groups {101, 102, 104}, {201, 202}, and {301, 302}. We can replicate this phenomenon on instances with arbitrarily large numbers of tuples and groups. In order to counter this behavior, we rely on standard intuitive notions that groups of duplicates are small and that distances between tuples in a group are small. Other types of specifications are possible within our framework.

Size Specification: The *size* specification captures the informal notion that groups of duplicates are small in size.

Diameter Specification: The *diameter* of a group is the maximum pair wise distance between tuples in a group. The diameter specification captures the intuitive notion that distance between pairs of tuples within a group of duplicates is bounded. This specification is similar in spirit to the traditional threshold constraint but each group of duplicates still satisfies the CS and SN constraints.

The DE Problem*:* Given a relation R, a distance function d, an aggregation function AGG, a positive real number c, and a positive integer K or a positive real number θ ($0 < \theta < 1$), partition R into the *minimum* number of groups $\{G_1,\ldots,G_m\}$ such that for all $1 \leq i \leq m$ (i) G_i is a compact set, and (ii) G_i is an SN(AGG, c) group, and (iii) $|G_i| \leq K$ or Diameter(G_i) $\leq \theta$.

For the rest of the paper, we assume that the common parameters d, AGG and c are implicit, and use $DE_S(K)$ to denote the instantiation where the size of each group is less than or equal to K, and $DE_D(\theta)$ to denote the instantiation where the diameter of each group is less than or equal to θ. We sometimes use DE to generically denote either of the two formulations. Note that it is also possible to use size and diameter specifications together.

3.1. Properties of the DE Formulation

In this section, we show that the DE problems have several desirable properties. Some of these properties characterize the behavior of a domain-independent distance-based duplicate elimination function under intuitive transformations to distances between tuples. Similar properties were proposed for domain-independent clustering functions by Kleinberg [18]. For the rest of the section, we assume that the distance function d, the aggregation function AGG, and SN threshold c are fixed. Constrained by space, we skip proof sketches of lemmas.

Uniqueness: The solutions to $DE_S(K)$ and $DE_D(\theta)$ are unique. Therefore, $DE_S(K)$ and $DE_D(\theta)$ can be viewed as *functions* which partition a given relation R.

Lemma 1: For a given set of parameters *K* or θ, the $DE_S(K)$ *and* $DE_D(\theta)$ problems have unique solutions.

Scale Invariance: Intuitively, the scale of a distance function does not impact the local structural properties of tuples, and hence duplicate elimination may not change. Formally, a partitioning function *f* is scale-invariant if for any distance function d and an $\alpha > 0$, $f(d) = f(\alpha \cdot d)$.

Lemma 2: $DE_S(K)$ is scale-invariant.

Split/Merge Consistency: Intuitively, shrinking distances between tuples in a group of duplicates, and expanding distances between tuples across groups may only change the partition resulting from a duplicate elimination function in limited ways. Under such a distance transformation, a tuple in group of duplicates cannot simultaneously become closer to tuples in a different group of duplicates, and farther from tuples in the original group. That is, a group of duplicates in the new partitioning under the transformed distances cannot be a union of proper subsets of groups in the original partition.

Let P be a partition of R. We say that a distance function d' is a *P-conscious transformation* of d if for any pair of tuples v_1, v_2 within the same group of P, $d'(v_1, v_2) \leq d(v_1, v_2)$ and for any pair v_1, v_2 in two different groups of P, $d'(v_1, v_2) \geq d(v_1, v_2)$. A partitioning function *f* is *split/merge consistent* if for any distance function d such that P = f(d) and any P-conscious transformation d' of d, each group in f(d') is either subset of a group in P or equal to the union of groups in P.

Lemma 3: The duplicate elimination functions $DE_S(K)$and $DE_D(\theta)$ are split/merge consistent.

Let P be the partition obtained by solving the duplicate elimination problem on R. Suppose, we construct a new relation R' by "homogenizing" duplicate tuples. That is, we make them very similar to each other (say, a very small distance apart). The new distances correspond to a P-conscious transformation of the original distance function *d*. Since the duplicate elimination functions are split/merge-consistent, re-invoking either one of them on R' would result in a partition that consists of union of groups in P or subsets of groups in P.

Constrained Richness: In most scenarios where duplicate elimination is applied, only a small fraction of tuples in the relation have duplicates and, the sizes of groups of duplicates tend to be small. Therefore, the range of duplication functions must include all partitions into a large number of small groups. This is in contrast to the original richness property in [18], which requires all possible partitions of a relation to be in the range. Our modification (which constrains the range from all possible partitions) differentiates the duplicate elimination from the clustering problem.

Let $0 \leq \alpha, \beta < 1$ be two positive constants. A partitioning function is *(α, β)-rich* if its range includes all partitions of a relation R having at least $|R|^{(1-\alpha)}$ groups such that the maximum size of any group is less than $|R|^{\beta}$. We now show that the range of $DE_S(K)$ is rich for a variety of parameter settings.

Lemma 4: $DE_S(K)$ is *(α, β)-rich* if $c < |R|^{(1-\alpha)}$ and $K \geq |R|^{\beta}$.

4. Duplicate Elimination Algorithm

In this section, we describe a scalable and efficient algorithm for solving the DE problem. We effectively exploit the ``cut'' (size or diameter) specifications to design an efficient algorithm. The main challenges are: (i) efficient identification of mutual nearest neighbors, and (ii) efficient partitioning of an input relation into the minimum number of compact SN groups which satisfy the cut specification. Our insight is to determine tuple pairs whose nearest neighbor sets are equal and then extend the pair wise equality to groups of tuples. We take a 2-phase approach: first determine the nearest neighbors (either the best K neighbors or all neighbors within a certain radius, depending on the cut specification) of every tuple in the relation, and then partition the relation into compact SN sets. By dividing it into two phases, we exploit nearest neighbor indexes in the first phase and the query processing ability of the database backend in the second phase to make the algorithm efficient and scalable. Figure 3 shows the architecture of our system implemented as a client over Microsoft SQLServer.

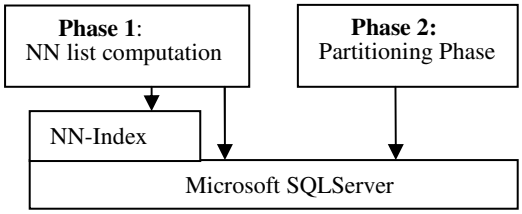

Figure 3: Architecture

In the first *nearest neighbor computation* phase, we assume the availability of an index for efficiently answering the following query: for any given tuple v in R, fetch its nearest neighbors. (Otherwise, we apply nested loop join methods in this phase.[1]) Since designing exact nearest neighbor indexes for distance (or similarity) functions typically used for duplicate elimination is very hard (e.g., [16]), several approximate and probabilistic indexes have been developed for standard distance functions (like cosine metric, edit distance, and fuzzy match similarity) [24, 23, 9]. For the purpose of this paper, we treat these probabilistic indexes as exact nearest neighbor indexes. The experimental results in Section 5 illustrate that this assumption does not negatively impact the actual results of our algorithm.

In the second *partitioning* phase, we partition the relation into the minimum number of valid groups of duplicates. We perform most of the processing required in this phase using standard SQL queries. In the process, (i) we exploit efficient query processing abilities of database systems and (ii) we avoid moving large amounts of data between the client and the server.

4.1. Nearest Neighbor Computation Phase

In this phase, we determine for each tuple in the relation R, its nearest neighbors and its neighborhood growth. The output of this phase is a relation *NN_Reln[ID, NN-List, NG]* where ID is the identifier of a tuple v whose neighbor growth is NG. For the $DE_S(K)$ problem, NN-List is the list of tuple identifiers of the K nearest neighbors of the tuple v; for the $DE_D(\theta)$ problem, NN-List is the list of tuple identifiers of all tuples whose distance from the tuple v is less than θ. Given an index that can be used for fetching the nearest neighbors (top-K or within a certain distance) and for computing the neighborhood growth, a straightforward procedure is to scan the input relation R and for each tuple v in R lookup the index, and write the tuple [v, NN-List(v), ng(v)] to the output.

However, the index structures typically used for fetching nearest neighbors are disk-based. For example, nearest neighbor indexes for the edit distance or the fuzzy match similarity functions have a structure similar to inverted indexes in IR [24, 9], and are usually large. Therefore, if consecutive tuples being looked up against these indexes are close to each other, then the lookup procedure is likely to access the same portion of the index, and the second lookup benefits due to the first. This significantly improves the buffer hit ratio and the overall running time. We now describe a lookup order that can be implemented efficiently.

4.1.1. Index Lookup Order

Let us consider the example tuples in Table 1. Suppose the order in which we lookup the nearest neighbors of tuples in R is 1, 12, 5, etc. In order to fetch the nearest neighbors of tuple 1 ("The Doors, LA Woman"), the indexing procedure would access a portion of the index and in the process cache it in the database buffer. A similar lookup for nearest neighbors of tuple 12 ("Aliyah, Are you ready") would access a completely different portion of the index because tuple 12 is very far from tuple 1. Alternatively, if we lookup nearest neighbors of tuple 2 ("Doors, LA Woman") after processing tuple 1, we will use almost the same portion of the index. Consequently, we can exploit the benefits of it being already in the database buffer.

A good lookup order must have two properties. First, tuples immediately preceding any tuple in the order must be close to it. Second, the procedure for ordering input tuples has to be efficient. For instance, if the lookup order requires the entire relation to be grouped (using an expensive clustering algorithm), then we would have lost

[1] Join-based methods developed for specific distance functions such as edit distance may be employed for the diameter specification [GIJ+01].

the benefit of ordering the input since the ordering process itself is very expensive.

We adopt the following *breadth first (BF)* order, which satisfies both the above requirements. The order corresponds to a breadth first traversal of a tree T constructed as follows. We select any input tuple to be the root of the tree. The children of any node in the tree are its nearest neighbors, which have not already been inserted into the tree. Figure 4 illustrates the input ordering. Note that we do not actually build such a tree but just fetch input tuples in the appropriate order.

Each tuple (except the root) in the BF order is always preceded by its siblings or the children of its parent's sibling. These tuples are much closer to each other than arbitrary pairs of tuples. For example, the tuple numbered 5 in Figure 4 is preceded by its siblings 2, 3, and 4. Therefore, all tuples in the lookup order are preceded by tuples that are very close to them. Consequently, the lookup algorithm makes localized index accesses and we observe a significant improvement in the buffer hit ratio. Our experimental study in Section 5 shows a 100% improvement in the number of K nearest neighbor queries processed per unit time interval, thus confirming the above intuition.

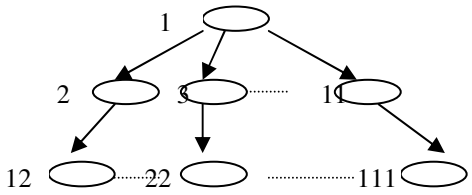

Figure 4: An example breadth first (BF)

For each lookup of an input tuple, we fetch its nearest neighbor tuples. Therefore, when we encounter a tuple, say the tuple numbered 12 in Figure 4, in the BF order we would have already fetched it when its parent tuple numbered *2* was looked up. Therefore, the database buffer would already have cached tuple numbered *12*. We can either explicitly cache these tuples, memory permitting, or rely on the database system to buffer recent accesses. In our implementation, we rely on the latter.

Figure 5 presents the pseudo-code for the nearest neighbor computation phase. Step 1 describes the initialization. Step 2 iterates over the BF order queue consisting of tuples to be looked up in the appropriate order. Step 3 describes the re-initialization of the queue when it empties. Ensuring that every tuple in R has been looked up can also be combined with the initialization of the queue into a single scan of R. That is, we start a scan of R, initialize the queue Q, and continue the scan whenever Q is empty until we lookup all tuples.

Space requirements: We now discuss the space required for implementing the lookup order. We maintain a bit vector in order to mark tuples whose nearest neighbors we have already looked up. In all practical scenarios, such a bit vector fits in main memory given current main memory sizes. Second, we maintain a queue to lookup tuples in the BF order. Since we only maintain the identifiers (long integers) of tuples in this queue, the size of the queue is relatively small and fits in main memory. In any case, when the queue outgrows a certain size, we stop inserting new tuples into it until it empties out. Thus, the additional main memory required is limited to the amount available for consumption.

PrepareNNLists(relation R, NNIndex I, int K, double θ)

1 Setup
   a. Create the relation NN_Reln(ID, NN-List, NG)
   b. Initialize queue Q by inserting a tuple from R
   c. Initialize bit vector H of size |R|
2 While Q is not empty
   a. v=front(Q); if H[v] is not set, get NN-List(v) and the number of neighbors within radius 2·NN(v) using index I;
   b. Compute neighbor growth NG(v)
   c. Add neighbors of v to Q, if Q has space
   d. Write the tuple [v, NN-List(v),NG(v)] to NN_Reln
   e. Set the bit H[v] in H
3 Insert another tuple not set in H from R into Q; goto 2.

Figure 5: Procedure for materializing NN_Reln

4.2. Partitioning Phase

The second *partitioning* phase uses the output of the first phase to partition the input relation into the minimum number of compact SN sets. The resulting partition is the solution to the DE problem. For clarity of description, we describe the procedure for the $DE_S(K)$ problem, and point out the straight-forward adjustments for the $DE_D(\theta)$ problem as required. We illustrate the intuition with the following example. Consider an example where the group {10, 50, 100, 150} forms a compact SN set. It is enough to know, besides the neighborhood growth (NG) values of each tuple, that the 4 nearest neighbor sets of the pairs {10, 50}, {10, 100}, {10, 150} are all equal. We deduce from the pair wise equality that the group {10, 50, 100, 150} is a compact set. As illustrated in Figure 6, our two-step procedure automates this intuition.

CSPairs Construction Step: The first step computes equality of neighbor sets of varying sizes between tuple pairs. That is, for a tuple pair (10, 100) as in Figure 6, we determine whether their 2-nearest neighbor sets, 3-nearest neighbor sets, and so on until K-nearest neighbor sets are equal. Such a comparison between tuples v_1 and v_2 yields the following list of boolean values [CS2, …, CSK] along with their neighbor growths $ng(v_1)$ and $ng(v_2)$. The value CSi ($2 \le i \le K$) denotes whether the i-neighbor sets of v_1

and v_2 are equal. We store the result in a temporary relation *CSPairs*.

We issue a SQL (*select into*) query against the output (NN_Reln) of the first phase to compute the *CSPairs* relation. The query involves a self-join of NN_Reln (say, FROM NN_Reln1, NN_Reln2) on the predicate that a tuple NN_Reln1.ID is less than NN_Reln2.ID and that it is in the K-nearest neighbor set of NN_Reln2.ID and vice-versa. The selected column list has the following two parts: (i) NN_Reln1.ID, NN_Reln2.ID, NN_Reln1.NG, NN_Reln2.NG, and (ii) for each j between 2 and K a case statement which returns 1 if the set of j-nearest neighbors of NN_Reln1.ID equals the set of j-nearest neighbors of NN_Reln2.ID. We can use user-defined functions to efficiently compute part (ii) of each record in CSPairs. However, we observe that for the Size-K specification when the ID-List attribute is expanded into K attributes, one per neighbor, we can use standard SQL and perform all of the computation at the database server. For the $DE_D(\theta)$ problem, we rely on a user-defined function to compute the list of boolean values [CS1,…]. Note that the sizes of these lists can be different for different pairs for the Diameter-θ specification.

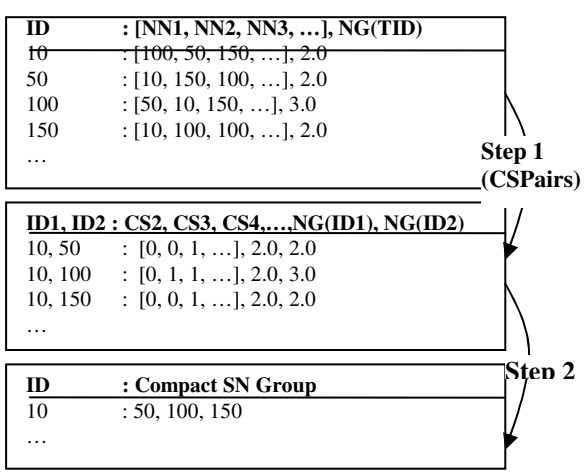

Figure 6: Example illustrating the partitioning phase

Partitioning Step: In this step, we extend the equality between neighbor sets of tuple pairs to sets of tuples and determine whether a set of neighbors is compact and satisfies the SN criterion. In Figure 6, the 4-neighbor sets of tuple pairs (10, 50), (10, 100), and (10, 150) are equal and therefore form a compact set of size 4. The set {10, 50, 100, 150} can be output as a group of duplicates provided (i) the aggregate SN value of this group is less than the threshold *c*, and (ii) it cannot be extended to a larger compact SN group. Observe that we do not explicitly check whether or not the 4-neighbor sets of pairs (50, 100), (50, 150), etc. because set equality is transitive. We now describe the procedure.

We process the CSPairs relation (output of CSPairs construction step) by issuing the following *CS-group* query "*select * from* CSPairs *order by* ID" to group all neighbor set comparison results between a tuple v and its neighbors v' where v.ID < v'.ID. Observe that in the result of the CS-group query, each compact SN set G will be grouped together under the tuple with the minimum ID in G. We process each group Q[ID=v] (identified by the same ID) of tuples in the result of the CS-group query. For a group Q[ID=v] of tuples, if v has not already been identified as belonging to a compact SN set, we determine the largest non-trivial (of size greater than *1*) compact SN set G_v that v can belong to. This set can be identified from just the group Q[ID=v]. We output the set G_v and mark each tuple in G_v to indicate its assignment to a compact set. For example, the set against tuple 10 is {10, 50, 100, 150}. We output the set {10, 50, 100, 150} and mark tuple identifiers 10, 50, 100, 150 as belonging to a valid compact SN set so that they are not processed again.

Space Requirements: We require a bit vector for marking whether or not a tuple has been processed.

4.3. Correctness & Complexity

The correctness of the algorithm follows from the following observation: Each compact SN set $G=\{v_1,\ldots,v_m\}$ such that $m \leq K$ and $v_i < v_j$ $(i < j)$ in the solution to DE is grouped under v_1 in the result of CS-group query. The reason is that the m-nearest neighbor sets of (v_1, v_2), …, (v_1, v_m) are all equal because G is a compact set. Also, no tuple in G can belong to a larger compact SN set in the solution for DE. Otherwise, v_1 cannot be the minimum identifier in G. Therefore, we do not have to further process groups in the query result under any of these tuples.

Phase 1 is an index nested loops join using the nearest neighbor index. Since indexes are effective at significantly reducing comparisons between tuples, the overall cost is linear in the number of tuples in the size of the input relation R. The cost of the second phase is the sum of the CSPairs construction step and the partitioning step costs. The CSPairs construction involves comparing pairs of lists of nearest neighbor sets of tuples, which for the Size-K specification is less than $K \cdot |R|$. The cost of sorting the CSPairs relation dominates the partitioning step cost as that of processing each group is very small. Observe that Phase 1 dominates the overall cost of the algorithm. Therefore, with an effective nearest neighbor index, our algorithm scales to large input relations. In contrast, standard clustering formulations (with some exceptions like the single linkage formulation) are usually NP-hard.

4.4. Determining the SN threshold c

In our formulation of the DE problem, we require the user to specify the sparse neighborhood threshold *c*. We

now describe a technique to assist a user in setting the threshold appropriately. The insight is to ask users to input aggregated information that is easier for them to estimate than it is to set a sparse neighborhood threshold, which requires a deeper understanding of the data distribution. We ask the user to estimate the fraction *f* of duplicate tuples in the input relation. The intuition is that most tuples whose NG values are less than the SN threshold are duplicate tuples. Therefore, in an ideal scenario when NG values of all unique tuples are above this threshold, the *f-percentile*[2] in the cumulative NG distribution *D* is the SN threshold. However, a *small* percentage of unique tuples may have lower NG values (besides the value *f* being only an estimated fraction of duplicate tuples). In order to be robust, we also rely on the aggregate characteristics that the fraction of unique tuples is much higher than the fraction of duplicates and that unique tuples have higher NG values. Reflecting this intuition, our heuristic is that the actual SN threshold is the least value $x=D^{-1}(y)$ around the *f*-percentile (say, $|D^{-1}(y)-f| \leq 0.05$) where there is a "spike" in the distribution D, i.e., the rate of growth $D'(x)$ of *D* is high. A spike is (heuristically) defined to occur when the $D'(x) > 1.0$. When no such spike exists, we use the value $D^{-1}(f+0.05)$ as the SN threshold. The parameters for defining the vicinity of *f* (placing an interval around *f*) and the spike may be guided by a user. Since the SN threshold value is not required until the second partitioning phase, we can re-use the NG values from the first phase.

4.5. Discussion

We now discuss a few issues relating to our formulation of the duplicate elimination problem, and extensions.

4.5.1. Incorporating Additional Knowledge

Consider a scenario where the domain expert knows that if two tuples together satisfy a predicate (e.g., two product descriptions are identical but for the version number at the end), then they cannot be duplicates of each other. We note that it is easy to add such additional *constraining predicates*—which rule out tuple pairs from being called duplicates—into our formulation of the duplicate elimination problem. The extended algorithm proceeds unchanged except for an additional check at the end to ensure that all groups satisfy this new criterion. If any group violates the new constraining predicate, we would further split the group. The ability to add additional constraining predicates allows us to incorporate constraining knowledge obtained via supervised learning [5, 26, 28] into our formulation of the DE problem. However, note that it is not possible to add "positive" knowledge, which stipulates that tuples are duplicates if a rule or predicate is satisfied, into our formulation easily. Our algorithm is not adaptable easily to the new problem extended with positive knowledge.

4.5.2. Minimality of Compact sets

Consider the set of tuples $\{v_1, v_1', v_2, v_2', v_3, v_3'\}$ such that v_i and v_i' $(1 \leq i \leq 3)$ are duplicates. Under appropriate distance and parameter assignment, the solution to the $DE_S(K)$ problem is a single group consisting of all six tuples.[3] Such an outcome occurs if the union of non-trivial compact sets is also a compact SN set allowing us to merge disjoint compact sets into a larger compact set. To avoid such unintuitive outcomes, we can impose the following notion of "minimality" on compact sets in addition to the mutual nearest neighbor restriction. *S* is a *minimal compact set* if *S* consists of mutual nearest neighbors and there do not exist disjoint subsets *S'* and *S''* of *S* such that $|S'| > 1$ and $|S''| > 1$ and *S'* and *S''* are compact sets. The algorithm for this new formulation is a straightforward adaptation of the algorithm for the original formulation. We just have to add an additional post-processing check of ensuring that each compact set is actually minimal. Otherwise, we would further split such groups into minimal groups.

Our experiments on a variety of real datasets however indicate that scenarios where we merge multiple minimal non-trivial compact sets together without violating the SN and the group size criteria are very rare. Such mergers can only occur if tuples across smaller compact sets are still very close to each other. However, in most real scenarios, either (i) all of them are really duplicates of each other or (ii) the neighborhood growths of individual tuples would be high preventing us from grouping them together. Therefore, in our DE formulation we do not constrain the solution to consist of minimal compact sets.

5. Experimental Evaluation

We present a thorough experimental evaluation of our duplicate elimination algorithm on real datasets to show that it is more accurate than the current threshold-based approaches. We first describe the setup and our evaluation metrics and then discuss the experimental results.

Real Datasets: We consider *Media*[artistName, trackName] and *Org*[name, address, city, state, zipcode] relations from internal data warehouses as well as publicly available datasets: *Restaurants*[Name], *BirdScott*[Name], *Parks*[Name], and *Census*[LastName, Last name, Middle initial, Number, Street], from the Riddle repository [8].

[2] The *f-percentile* of a cumulative distribution *D* is the value *x* at which *D(x)* equals *f*.

[3] An example assignment: the distance between v_i and v_i' is less than half that between v_i (v_i') and v_j (v_j') for all $i \neq j$; the SN threshold is greater than 1.0 and K is greater than 6.

Algorithms Compared: We compare our $DE_S(K)$ and $DE_D(\theta)$ formulations with a standard thresholding strategy (denoted *thr*) based on single linkage clustering [e.g., 15, 20]. As discussed earlier, we induce (for *thr*) the threshold graph using the output (NN_Reln) of the nearest neighbor computation phase and a distance threshold θ. Each maximal connected component is returned as a set of duplicates. Note that most (almost 80-90%) sets of duplicates just consist of tuple pairs. Hence, alternative methods for componentizing the threshold graph into stars or cliques still return similar results.

Evaluation Metrics: We use *precision* and *recall* metrics to evaluate duplicate elimination algorithms [6]. Recall is the fraction of true pairs of duplicate tuples identified by an algorithm. And, precision is the fraction of tuple pairs an algorithm returns which are truly duplicates. Higher recall and precision values are better. A precision versus recall graph plots the (recall, precision) values for various parameter settings. Comparing the precision versus recall plots allows us to comprehensively compare algorithms across several parameter settings.

Distance Functions: We evaluate our formulation using two distance functions. We employ the *edit distance (ed)* [27] and a function combining edit distance and cosine metric with IDF weights, called *fuzzy match similarity (fms)*. We consider a symmetric variant of the original fms function [9]. The function *fms* was shown to be very effective for matching erroneous tuples with their correct counterparts. To illustrate, consider the following example strings: "microsoft corp" and "microsft corporation." They are close because 'microsoft' and 'microsft' are close according to edit distance and the IDF weights of 'corp' and 'corporation' are relatively small. In contrast, edit distance between "microsoft corp" and "mic corporation" is less than that between "microsoft corp" and "microsft corporation." Similarly, cosine metric (with IDF weighting) places "microsft corporation" and "boeing corporation" closer to each other than "microsoft corp" and "microsft corporation." Further, the fuzzy match similarity function is efficiently (probabilistically) indexable [9].

5.1.1. Quality

Figures 10 plots the precision versus recall graphs for duplicate elimination algorithms using the edit distance function: threshold-based single linkage (thr), $DE_S(K)$ with c=4 (i.e., neighborhood growth must be less than 4 for sparseness) and c=6, and $DE_D(\theta)$ with c=4 and c=6. We fix the aggregation function (AGG) to be Max and vary the parameters K and θ, respectively and plot the precision versus recall values. Figure 11 plots similar graphs for algorithms using the fuzzy match similarity function. For several datasets, and for both edit distance and fuzzy match similarity, the $DE_S(K)$ and $DE_D(\theta)$ plots outperform thresholding approaches. For the same recall, our DE approaches yield higher precision (often 5-10% and sometimes 20% or more), especially for higher recall values. Only for the Parks dataset, there is no improvement over threshold approaches. Further, the precision-recall graph for the $DE_S(.)$ plots are mostly concentrated around the same recall-precision values whereas the $DE_D(.)$ plots have a wider spread of precision-recall values. The reason is that the nearest neighbor lists for the $DE_K()$ formulation are dependent only on K and not on the specific distance thresholds. Usually, the number of compact groups of size 2 is far greater than those with number 3, etc. Therefore, the variations in precision and recall with variations in K are not very high. In contrast, the sizes of the nearest neighbor lists for the $DE_D(\theta)$ varies significantly with variations in θ. Consequently, the precision and recall values also vary with θ. Therefore, the Diameter-θ specification is useful for better control on precision.

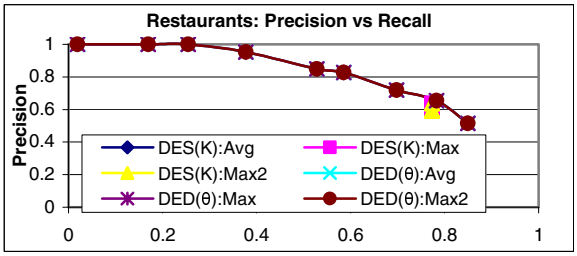

Figure 7: Aggregation functions

Figure 7 plots the precision versus recall plots on the Restaurants dataset of $DE_S()$ and $DE_D()$ for different aggregation functions (Max, Avg, Max2—the 2nd maximum value). All three aggregation functions yield very similar results because a large percentage of groups are of size 2.

5.1.2. Run-time Performance

We illustrate the performance of our algorithm via the $DE_S(K)$ problem with the fuzzy match similarity function; results for the $DE_D(\theta)$ would be similar. We first evaluate the impact of BF ordering on improving the performance of the first nearest neighbor computation phase, and then evaluate the overall scalability.

BF Ordering: We evaluate the impact of BF ordering using a relation consisting of *3 million* organization addresses. We measure (i) database buffer hit ratio (*BHR*), (ii) processor usage (*PU*), and (iii) the throughput or the number of input tuples looked up per unit time interval (*Thrpt*) for both *BF* and random (*rnd*) orders. We vary the memory sizes allocated to the database buffer between 32MB and 128MB. Figure *8* presents the results. We observe that BHR, PU, and Thrpt for the BF order are higher than that for the random order. In particular, the

overall throughput improved by almost 100% due to the BF order. That is, BF ordering *halves* the time required for the first phase.

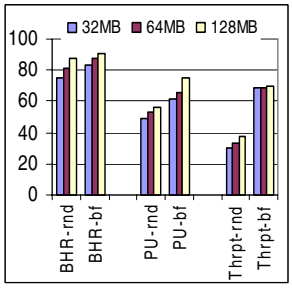

Figure 8: Comparing buffer hit ratio % (BHR), processor usage % (PU), and Lookup throughput (Thrpt) for random (rnd) and breadth first (bf) ordering.

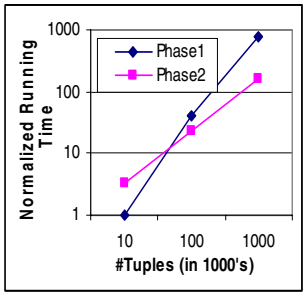

Figure 9: Normalized running times for both phases

Scalability: We now study the scalability of our algorithm using a relation containing 3 million organization addresses. Figure *9* reports the normalized running times (normalized by the time required for executing Phase 1 over *10K* tuples) of both phases versus the size (in multiples of 1000) of the dataset; both axes are on logarithmic scales. The linearity of the plots demonstrates the scalability of both phases of our algorithm.

5.1.3. Summary

We have shown our DE formulation to be accurate and robust and that they yield better precision-recall tradeoffs than previous approaches. We then illustrated the efficiency and scalability of our algorithm. In particular, we demonstrated the benefit of BF ordering for significantly reducing (to almost 50%) the running time of the first nearest neighbor computation phase of the algorithm for solving the DE problem.

6. Related Work

The problem of duplicate elimination has recently received a lot of attention due to its practical significance in a variety of data management scenarios. Previous solutions to duplicate elimination can be classified into supervised and unsupervised approaches.

Supervised approaches learn rules characterizing pairs of duplicates from training data consisting of known duplicates [11, 5]. These approaches assume that training data exhibit the variety and distribution of errors observed in practice. It is difficult, if not impossible, to obtain such comprehensive training data, an issue that was addressed to a limited extent by active learning approaches [26, 28] which have the drawback of requiring interactive manual guidance. In many real data integration scenarios, it is not possible to obtain good training data or interactive user guidance.

As discussed earlier, previous unsupervised methods ignore local structural properties and rely on global thresholds over distances (in particular, edit distance or cosine similarity) to detect duplicates, and hence lead to poor recall-precision tradeoffs [e.g., 15, 21, 14]. The duplicate elimination problem has also been studied in the record linkage literature (e.g., [13, 17, 19]). These approaches still rely on threshold-based notions but use similarities aggregating matches between attribute values. See [29] for a recent survey of these methods. Alternative approaches for partitioning the threshold-graph into components that are strongly connected (e.g., cliques or almost cliques) would result in almost the same groups of tuples as those from single linkage partitioning because most groups of duplicates in practice are very small (of size 2 or 3). Recent research has focused on improving the distance functions [10, 1] and in determining appropriate thresholds, but they still inherently involve global thresholds for single linkage clustering algorithms. Most of these distance or similarity functions can be used with our DE formulations thus achieving better precision-recall tradeoffs.

Our formulation of the duplicate elimination problem is different from standard clustering formulations [12, 22] primarily because of the CS and SN criteria. Most clustering formulations insist that each cluster be very dense and contain a large number of tuples whereas our DE formulation focuses on groups consisting of mutual nearest neighbors and the local neighborhood being sparse. Consequently, we cannot directly use standard clustering formulations and algorithms. A notion similar to the SN criterion has been explored in the context of outlier detection [4].

Several *blocking* approaches have been proposed to speed up algorithms for solving the threshold-based duplicate elimination problem [2, 15]. The idea (similar to that of hash join algorithms) is to partition the relation into blocks and to only compare records within blocks. However, they do not guarantee that all required nearest neighbors of a tuple are also in the same block. Hence, we are unable to use these blocking strategies.

Braunmueller et al. develop techniques for optimizing batches of similarity queries where they simultaneously process several nearest neighbor queries [7]. However, these approaches require access to all candidate tuples being fetched by the index for any single nearest neighbor query. Implementing such a strategy requires changes to the indexing structure and/or the database backend. Since implementing our system as a client to standard database systems is one of our design goals, we are unable to adopt their approach.

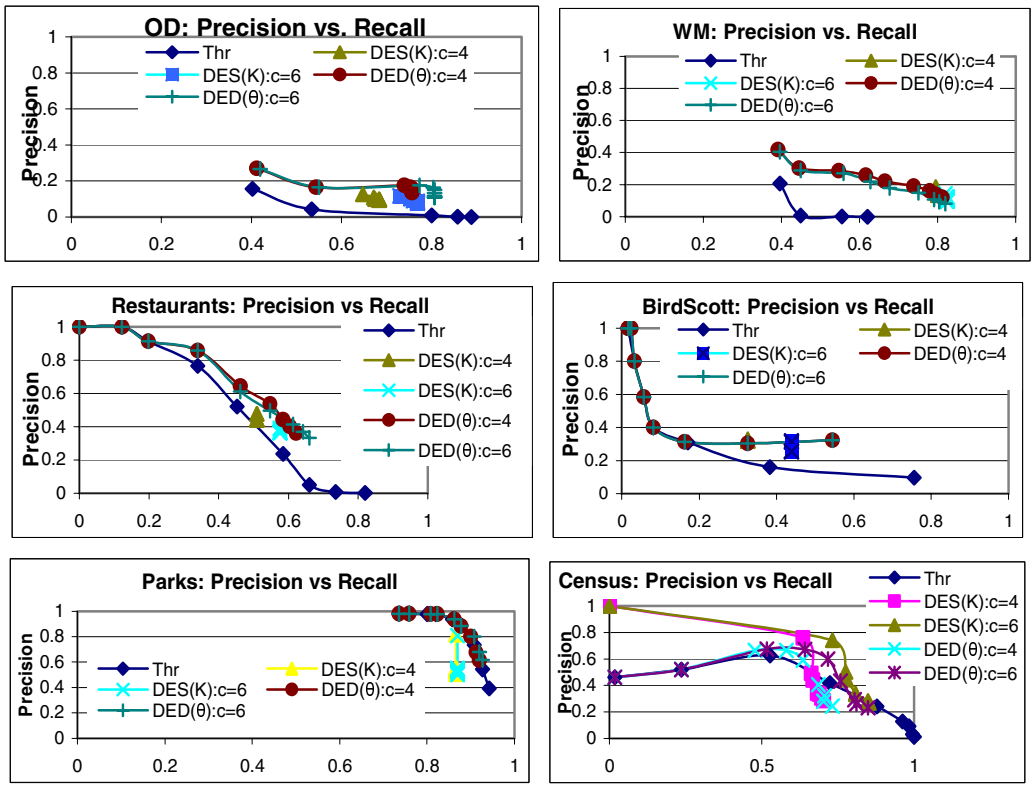

Figure 10: Precision vs. Recall using Edit distance

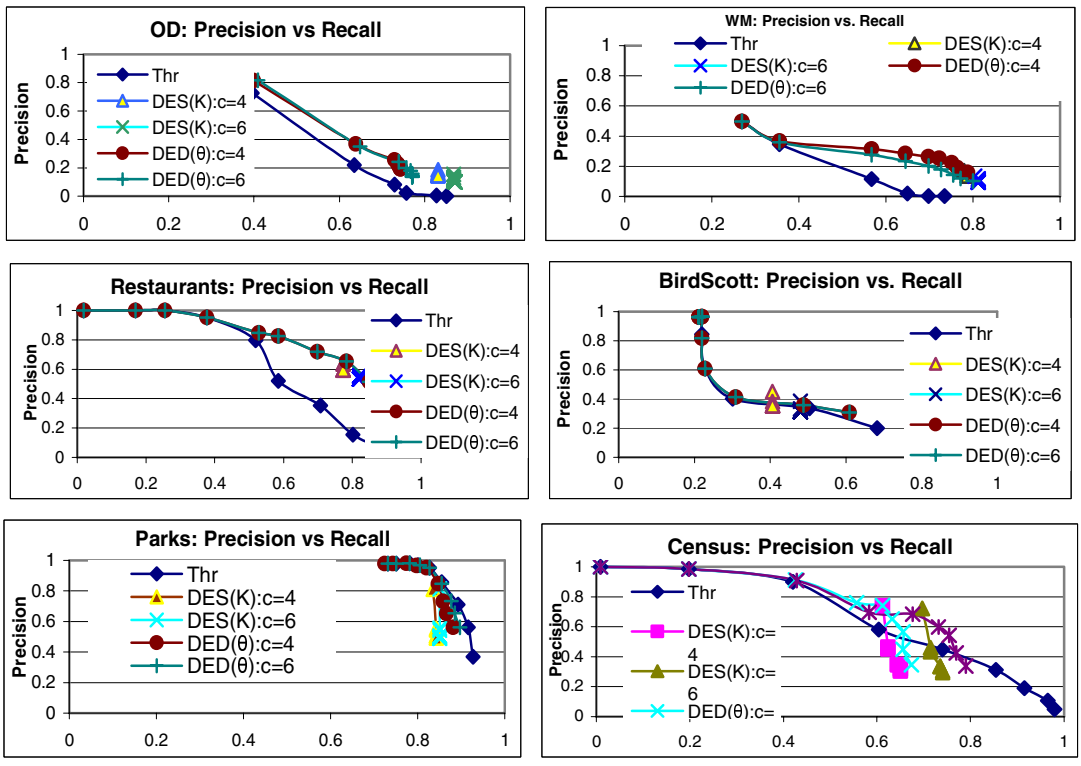

Figure 11: Precision vs. Recall plots using the fuzzy match similarity function

7. Conclusions

In this paper, we propose a new formulation for the duplicate elimination problem based on two fundamental properties (compact set and sparse neighborhood) which characterize duplicate tuples. We show that our formulation has several desirable characteristics under intuitive transformations to distances between tuples. We develop an efficient algorithm for solving the duplicate elimination problem. Using real datasets, we show the quality and robustness of our formulation as well as the scalability of our algorithm.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the International Conference on Very Large Databases*, 2002.

[2] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proceedings of the ACM SIGKDD workshop on data cleaning, record linkage, and object identification*, August 2003.

[3] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2), 1983.

[4] Breunig S., Kriegel H.-P., Ng R., Sander J.: 'LOF: Identifying Density-Based Local Outliers'. In *Proceedings of ACM SIGMOD*, 2000

[5] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery in databases*, 2003.

[6] M. Bilenko and R. J. Mooney. On evaluation and training-set construction for duplicate detection. In *Proceedings of the ACM SIGKDD workshop on data cleaning, record linkage, and object identification*, August 2003.

[7] Braunmueller B., Ester M., Kriegel H.-P., and Sander J. Multiple similarity queries: A basic dbms operation for mining in metric databases. *IEEE Transactions on Knowledge and Data Engineering*, 13(1), 2001.

[8] M. Bilenko. RIDDLE: Repository of information on duplicate detection, record linkage, and identity uncertainty. http://www.cs.utexas.edu/users/ml/riddle/index.html

[9] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD*, June 2003.

[10] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD*, pages 201--212, Seattle, WA, June 1998.

[11] W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery in databases*, Edmonton, Canada, July 23-26 2002.

[12] E. Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics*, 21, 1965.

[13] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64:1183--1210, 1969.

[14] 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 *Proceedings of the 27th international conference on very large databases (VLDB)*, pages 491--500, Roma, Italy, September 11-14 2001.

[15] M. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem for large databases. *Data mining and knowledge discovery*, 2(1): 9—37, 1998.

[16] Piotr Indyk. Approximate nearest neighbor under edit distance via product metrics. In *15th Symposium on discrete algorithms*, 2004.

[17] B. Kilss and W. Alvey. Record linkage techniques--1985. statistics of income division. Internal revenue service publication, 1985.

[18] J. Kleinberg. An impossibility theorem for clustering. In *Advances in Neural Information Processing Systems*, 2002.

[19] L. Gill. Ox-link: The oxford medical record linkage system. *Record Linkage Techniques 1997*, 1999.

[20] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the 2nd international conference on knowledge discovery and databases (KDD)*, 1996.

[21] A. Monge and C. Elkan. An efficient domain independent algorithm for detecting approximately duplicate database records. In *Proceedings of the SIGMOD Workshop on Data Mining and Knowledge Discovery*, Arizona, May 1997.

[22] Ester M., Kriegel H.-P., Sander J., and Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd int.l conference on knowledge discovery and databases (KDD)*, 1996.

[23] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19--27, 2001.

[24] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28--46, 2002.

[25] E. Rahm and H. Hai Do. Data cleaning: Problems and current approaches. IEEE Data Engineering Bulletin, 23(4):3--13, December 2000.

[26] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery in databases*, Edmonton, Canada, July 23-26 2002.

[27] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology,* 147:195--197, 1981.

[28] S. Tejada, C. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of the eighth ACM SIGKDD international conference on knowledge discovery in databases*, Edmonton, Canada, July 23-26 2002.

[29] W. Winkler. Data cleaning methods. In Proceedings of the ACM SIGKDD workshop on data cleaning, record linkage, and object identification, August 2003.