Generic Entity Resolution with Data Confidences

David Menestrina Omar Benjelloun
Hector Garcia-Molina

Stanford University
{dmenest,benjello,hector}@cs.stanford.edu

Abstract

We consider the *Entity Resolution* (*ER*) problem (also known as deduplication, or merge-purge), in which records determined to represent the same real-world entity are successively located and merged. Our approach to the ER problem is *generic*, in the sense that the functions for comparing and merging records are viewed as black-boxes. In this context, managing numerical confidences along with the data makes the ER problem more challenging to define (e.g., how should confidences of merged records be combined?), and more expensive to compute. In this paper, we propose a sound and flexible model for the ER problem with confidences, and propose efficient algorithms to solve it. We validate our algorithms through experiments that show significant performance improvements over naive schemes.

1 Introduction

When data from different sources is integrated, often multiple input records refer to the same real-world entity, e.g., to the same customer, the same product or the same organization. *Entity resolution* (ER) identifies the records that refer (or are likely to refer) to the same entity, and merges these records. A merged record becomes a "composite" of the source records. In general, a merged record needs to be compared and possibly merged with other records, since the composition of information may now make it possible to identify new relationships. For instance, say record r_1 gives the name and driver's license of a person, while record r_2 gives an address and the same driver's license number. Say we merge r_1 and r_2 based on the matching driver's license. Now we have both a name and an address for this person, and this combined information may make it possible to connect this merged record with say r_3, containing a similar name and address. Note that neither r_1 nor r_2 may match with r_3, because they do not contain the combined information that the merged record has. Entity resolution is also referred to as deduplication and record linkage.

Often, numerical confidences (or uncertainties) play a role in entity resolution. For instance, the input records may come from unreliable sources, and have confidences associated with them. The comparisons between records may also yield confidences that represent how likely it is that the records refer to the same real-world entity. Similarly, the merge process may introduce additional uncertainties, as it may not be certain how to combine the information from different records. In each application domain, the interpretation of the confidence numbers may be different. For instance, a confidence number may represent a "belief" that a record faithfully reflects data from a real-world entity, or it may represent how "accurate" a record is.

Even though ER is a central problem in information integration, and even though confidences are often an integral part of resolution, relatively little is known about how to *efficiently* deal with confidences. Specifically, confidences may make the ER process computationally more expensive, as compared to a scenario where confidences are not taken into account. For instance, without confidences, the order in which records are merged may be unimportant, and this property can be used to find efficient ER strategies. However, confidences may make order critical. For instance, say we merge r_1 to r_2 and then to r_3, giving us a record r_{123}. Because r_1 and r_2 are "very similar", we may have a high confidence in the intermediate result, which then gives us high confidence in r_{123}. However, say we merge r_1 to r_3 and then to r_2, giving us record r_{132}. In this case, r_1 and r_3 may not be "that similar", leading t a lower confidence r_{132}. Records r_{123} and r_{132} may even have the same attributes, but may have different confidences because they were derived differently. Thus, ER must consider many more potential derivations of composite

records.

Our goal in this paper is to explore ways to reduce the high computational costs of ER with confidences. We wish to achieve this goal without making too many assumptions about the confidence model and how confidences are computed when record are merged. Thus, we will use a generic *black-box model* for the functions that compare records, that merge records, and that compute new confidences. We will then postulate properties that these functions may have: if the properties hold, then efficient ER with confidences will be possible. If they do not hold, then one must run a more-general version of ER (as we will detail here). Since we use generic match and merge functions, the algorithms we present can be used in many domains. All that is required is to check what properties the match and merge functions have, and then to select the appropriate algorithm.

The contributions of this paper are the following:

- We define a generic framework for managing confidences during entity resolution (Sections 2 and 3).
- We present Koosh, an algorithm for resolution when confidences are involved (Section 4).
- We present three improvements over Koosh that can significantly reduce the amount of work during resolution: domination, packages and thresholds. We identify properties that must hold in order for these improvements to be achievable (Sections 5, 6, and 7).
- We evaluate the algorithms and quantify the potential performance gains (Section 8).

2 Model

Each *record* r consists of a *confidence* $r.\mathcal{C}$ and a set of *attributes* $r.\mathcal{A}$. For illustration purposes, we can think of each attribute as a label-value pair, although this view is not essential for our work. For example, the following record may represent a person:

0.7 [name: "Fred", age: $\{45, 50\}$, zip: 94305]

In our example, we write $r.\mathcal{C}$ (0.7 in this example) in front of the attributes. (A record's confidences could simply be considered as one of its attributes, but here we treat confidences separately to make it easier to refer to them.) Note that the value for an attribute may be a set. In our example, the age attribute has two values, 45 and 50. Multiple values may be present in input records, or arise during integration: a record may report an age of 45 while another one reports 50. Some merge functions may combine the ages into a single number (say, the average), while others may decide to keep both possibilities, as shown in this example.

Note that we are using a single number to represent the confidence of a record. We believe that single numbers (in the range 0 to 1) are the most common way to represent confidences in the ER process, but more general confidence models are possible. For example, a confidence could be a vector, stating the confidences in individual attributes. Similarly, the confidence could include lineage information explaining how the confidence was derived. However, these richer models make it harder for application programmers to develop merge functions (see below), so in practice, the applications we have seen all use a single number.

Generic ER relies on two black-box functions, the *match* and the *merge* function, which we will assume here work on two records at a time:

- A match function $M(r, s)$ returns true if records r and s represent the same entity. When $M(r, s) = true$ we say that r and s match, denoted $r \approx s$.
- A merge function creates a composite record from two matching records. We represent the merge of record r and s by $\langle r, s \rangle$.

Note that the match and merge functions can use global information to make their decisions. For instance, in an initialization phase we can compute say the distribution of terms used in product descriptions, so that when we compare records we can take into account these term frequencies. Similarly, we can run a clustering algorithm to identify sets of input records that are "similar." Then the match function can consult these results to decide if records match. As new records are generated, the global statistics need to be updated (by the merge function): these updates can be done incrementally or in batch mode, if accuracy is not essential.

The pairwise approach to match and merge is often used in practice because it is easier to write the functions. (For example, ER products from IBM, Fair Isaac, Oracle, and others use pairwise functions.) For instance, it is extremely rare to see functions that merge more than two records at a time. To illustrate, say we want to merge 4 records containing different spellings of the name "Schwartz." In principle, one could consider all 4 names and come up with some good "centroid" name, but in practice it is more common to use simpler strategies. For example, we can just accumulate all spellings as we merge records, or we can map each spelling to the closest name in a dictionary of canonical names. Either approach can easily be implemented in a pairwise fashion.

Of course, in some applications pairwise match

functions may not be the best approach. For example, one may want to use a set-based match function that considers a set of records and identifies the pair that should be matched next, i.e., $M(S)$ returns records $r, s \in S$ that are the best candidates for merging. Although we do not cover it here, we believe that the concepts we present here (e.g., thresholds, domination) can also be applied when set-based match functions are used, and that our algorithms can be modified to use set-based functions.

Pairwise match and merge are generally not arbitrary functions, but have some properties, which we can leverage to enable efficient entity resolution. We assume that the match and merge functions satisfy the following properties:

- *Commutativity:* $\forall r, s,\ r \approx s \Leftrightarrow s \approx r$ and if $r \approx s$ then $\langle r, s\rangle = \langle s, r\rangle$.
- *Idempotence:* $\forall r, r \approx r$ and $\langle r, r\rangle = r$.

We expect these properties to hold in almost all applications (unless the functions are not property implemented). In one ER application we studied, for example, the implemented match function was not idempotent: a record would not match itself if the fields used for comparison were missing. However, it was trivial to add a comparison for record equality to the match function to achieve idempotence. (The advantage of using an idempotent function will become apparent when we see the efficient options for ER.)

Some readers may wonder if merging two identical records should really give the same record. For example, say the records represent two observations of some phenomena. Then perhaps the merge record should have a higher confidence because there are two observations? The confidence would only be higher if the two records represent independent observations, not if they are identical. We assume that independent observations would differ in some way, e.g., in an attribute recording the time of observation. Thus, two *identical* records should really merge into the same record.

Note that in [2] we present two additional properties, associativity and representativity. These properties generally do *not* hold when confidences are involved. As argued in the introduction, merging records in different orders may lead to different confidences. Similarly, when a record r_1 merges with r_2 forming r_3, we cannot discard r_1 and r_2, as they may have higher confidence that r_3 (see example below). Because we not to assume associativity and representativity, in this paper we require as a starting point a more general algorithm (called Koosh, see Section 4).

3 Generic Entity Resolution

Given the match and merge functions, we can now ask what is the correct result of an entity resolution algorithm. It is clear that if two records match, they should be merged together. If the merged record matches another record, then those two should be merged together as well. But what should happen to the original matching records? Consider:
$r_1 = 0.8[name : \text{Alice}, areacode : 202]$
$r_2 = 0.7[name : \text{Alice}, phone : \text{555-1212}]$.

The merge of the two records might be:
$r_{12} = 0.56[name : \text{Alice}, areacode : 202, phone : \text{555-1212}]$

In this case, the merged record has all of the information in r_1 and r_2, but with a lower confidence. So dropping the original two records would lose information. That is, if we drop r_1 we would no longer know that we are quite confident (confidence = 0.8) that Alice's area code is 202. The new record r_3 connects Alice to area code 202 with lower confidence. Therefore, to be conservative, the result of an entity resolution algorithm must contain the original records as well as records derived through merges. Based on this intuition, we define the correct result of an entity resolution algorithm as follows.

Definition 3.1 *Given a set of records R, the* result of Entity Resolution $ER(R)$ *is the smallest set S such that:*

1. $R \subseteq S$,
2. *For any records $r_1, r_2 \in S$, if $r_1 \approx r_2$, then $\langle r_1, r_2\rangle \in S$.*

Note that S could be an infinite set. (We say that S_1 is smaller than S_2 if $S_1 \subseteq S_2$.) The terminology "smallest" implies that there is a unique result, which we show next.

First, we need to introduce derivation trees for records.

Definition 3.2 *A* derivation tree D *for record r is a binary tree whose nodes represent records. The root represents r, and for any node, its children represent the two records that merged to produce it (e.g., if a and b are children of c, then $a \approx b$ and $\langle a, b\rangle = c$.). The leaves of D, denoted $L(D)$ represent base records. The derivation tree of a base record contains only that record as the root.*

Intuitively, the derivation tree D explains how r was derived. Note incidentally that two nodes in D can represent the same record. For example, say that a and b merge to produce d; a and c merge to yield e; and d and e merge into f. In this example two separate leaf nodes in D represent a.

```
1: input: a set R of records
2: output: a set R' of records, R' = ER(R)
3: R' ← R; N ← ∅
4: repeat
5:    R' ← R' ∪ N; N ← ∅
6:    for all pairs (r_i, r_j) of records in R' do
7:       if r_i ≈ r_j then
8:          merged ← ⟨r, r'⟩
9:          if merged ∉ R' then
10:            add merged to N
11:         end if
12:      end if
13:   end for
14: until N = ∅
15: return R'
```

Algorithm 1: The BFA algorithm for ER(R)

```
1: input: a set R of records
2: output: a set R' of records, R' = ER(R)
3: R' ← ∅
4: while R ≠ ∅ do
5:    r ← a record from R
6:    remove r from R
7:    for all r' ∈ R' do
8:       if r ≈ r' then
9:          merged ← ⟨r, r'⟩
10:         if merged ∉ R ∪ R' ∪ {r} then
11:            add merged to R
12:         end if
13:      end if
14:   end for
15:   add r to R'
16: end while
17: return R'
```

Algorithm 2: The Koosh algorithm for ER(R)

Definition 3.3 *Given a base set of records R, a record r is* well-formed *if there exists a derivation tree D with root r s.t. $L(D) \subseteq R$.*

PROPOSITION 3.1. Let r be a well-formed record, and let D be its derivation tree ($L(D) \subseteq R$). Then every internal node of D represents a well-formed record.

LEMMA 3.2. Given a set R, every record in $ER(R)$ is well-formed.

THEOREM 3.3. The solution to the ER problem is unique.

Proofs for the lemmas and theorems are given in Appendix.

Intuitively, ER(R) is the set of all records that can be derived from the records in R, or from records derived from them. A natural "brute-force" algorithm (BFA) for computing $ER(R)$ is given in Figure 1. The following proposition states the correctness of BFA. Its proof is given in Appendix.

THEOREM 3.4. For any set of records R such that $ER(R)$ is finite, BFA terminates and correctly computes $ER(R)$.

4 Koosh

BFA is correct but inefficient, essentially because the results of match comparisons are forgotten after every iteration. As an example, suppose $R = r_1, r_2$, $r_1 \approx r_2$, and $\langle r_1, r_2 \rangle$ doesn't match anything. In the first round, BFA will compare r_1 with r_2, and merge them together, adding $\langle r_1, r_2 \rangle$ to the set. In the second round, r_1 will be compared with r_2 a second time, and then merged together again. This comparison is redundant. In data sets with more records, the number of redundant comparisons is even greater.

We give in Figure 2 the Koosh algorithm, which improves upon BFA by removing these redundant comparisons. The algorithm works by maintaining two sets. R is the set of records that have not been compared yet, and R' is a set of records that have all been compared with each other. The algorithm works by iteratively taking a record r out of R, comparing it to every record in R', and then adding it to R'. For each record r' that matched r, the record $\langle r, r' \rangle$ will be added to R.

Using our simple example, we illustrate the fact that redundant comparisons are eliminated. Initially, $R = \{r_1, r_2\}$ and $R' = \emptyset$. In the first iteration, r_1 is removed from R and compared against everything in R'. There is nothing in R', so there are no matches, and r_1 is added to R'. In the second iteration, r_2 is removed and compared with everything in R', which consists of r_1. $r_1 \approx r_2$, so the two records are merged and $\langle r_1, r_2 \rangle$ is added to R. Record r_2 is added to R'. In the third iteration, $\langle r_1, r_2 \rangle$ is removed from R and compared against r_1 and r_2 in R'. Neither matches, so $\langle r_1, r_2 \rangle$ is added to R'. This leaves R empty, and the algorithm terminates. In the above example, r_1 and r_2 were compared against each other only once, so the redundant comparison that occurred in BFA has been eliminated.

Before establishing the correctness of Koosh, we show the following lemmas, whose proof is given in Appendix.

LEMMA 4.1. At any point in the Koosh execution, $R' \subseteq ER(R)$. If $ER(R)$ is finite, Koosh terminates.

LEMMA 4.2. When Koosh terminates, all records in the initial set R are in R'.

LEMMA 4.3. Whenever Koosh reaches line 4:

1. all pairs of distinct records in R' have been compared
2. if any pair matches, the merged record is in R or R'.

The correctness of Koosh is established by the following theorem, proven in Appendix.

THEOREM 4.4. For any set of records R such that $ER(R)$ is finite, Koosh terminates and correctly computes $ER(R)$.

Let us now consider the performance of Koosh. First, we observe the initial motivation for Koosh.

LEMMA 4.5. For any records r_1 and r_2, Koosh will never compare them more than once.

THEOREM 4.6. Koosh is optimal, in the sense that no algorithm that computes $ER(R)$ makes fewer comparisons.

5 Domination

Even though Koosh is more efficient than BFA, both algorithms are very expensive, especially since the answer they must compute can be very large. In this section and the next two, we explore ways to tame this complexity, by exploiting additional properties of the match and merge functions (Section 6), or by only computing a still-interesting subset of the answer (using thresholds, in Section 7, or the notion of domination, which we introduce next).

To motivate the concept of domination, consider the following records r_1 and r_2, that match, and merge into r_3:
$r_1 = 0.8[name : \text{Alice},\ areacode : 202]$
$r_2 = 0.7[name : \text{Alice},\ phone : \text{555-1212}]$.
$r_3 = 0.7[name : \text{Alice},\ areacode : 202,\ phone : \text{555-1212}]$.

The resulting r_3 contains all of the attributes of r_2, and its confidence is the same. In this case it seems natural to consider a "dominated" record like r_2 to be redundant and unnecessary. Thus, a user may only want the ER answer to contain only non-dominated records. These notions are formalized by the following definitions.

Definition 5.1 *We say that a record r* dominates *a record s, denoted $s \leq r$, if the following two conditions hold:*

1. $s.\mathcal{A} \subseteq r.\mathcal{A}$
2. $s.\mathcal{C} \leq r.\mathcal{C}$

Definition 5.2 *Given a set of base records R, the non-dominated entity-resolved set, $NER(R)$ contains all records in $ER(R)$ that are* non-dominated. *That is, $r \in NER(R)$ if and only if $r \in ER(R)$ and there does not exist any $s \in ER(R)$, $s \neq r$, such that $r \leq s$.*

Note that just like $ER(R)$, $NER(R)$ may be infinite. In case it is not, one way to compute $NER(R)$ is to first compute $ER(R)$ (assuming it is also finite) and then remove the dominated records. This strategy does not save much effort since we still have to compute $ER(R)$. A significant performance improvement is to discard a dominated record as soon as it is found in the resolution process, on the premise that a dominated record will never participate in the generation of a non-dominated record. This premise is stated formally as follows:

- *Domination Property:* If $s \leq r$ and $s \approx x$ then $r \approx x$ and $\langle s, x\rangle \leq \langle r, x\rangle$.

This domination property may or may not hold in a given application. For instance, let us return to our r_1, r_2, r_3 example at the beginning of this section. Consider a fourth record $r_4 = 0.9[name :$ Alice, $areacode :$ 717, $phone :$ 555-1212, $age :$ 20]. A particular match function may decide that r_4 does *not* match r_3 because the area codes are different, but r_4 and r_2 may match since this conflict does not exist with r_2. In this scenario, we cannot discard r_2 when we generate a record that dominates it (r_3), since r_2 can still play a role in some matches.

However, in applications where having more information in a record can never reduce its match chances, then the domination property can hold and we can take advantage of it. In particular, if the domination property holds, the following lemma tells us that in computing $NER(R)$ we can throw away dominated records as we find them. The proof is given in Appendix.

LEMMA 5.1. If the domination property holds, then every record in $NER(R)$ has a well-formed derivation tree with no dominated records.

5.1 Algorithm Koosh-ND

Both algorithms BFA and Koosh can be modified to eliminate dominated records early. There are several ways to remove dominated records, presenting a tradeoff between how many domination checks are performed and how early dominated records are dropped. For example, a simple way would be to test for domination of one record over another just before they are compared for matching. If a record is dominated, it is dropped. Since both algorithms end up comparing all records in the result set against

one another, this would guarantee that all dominated records are removed. However, this scheme does not identify domination as early as possible. For instance, suppose that an input base record r_1 is dominated by another base record r_2. Record r_1 would not be eliminated until it is compared with r_2. In the meantime, r_1 could be unnecessarily compared to other records.

To catch domination earlier, we propose here a more advanced scheme. We will focus on modifying Koosh to obtain Koosh-ND; the changes to BFA are analogous. First, Koosh-ND begins by removing all dominated records from the input set. Second, within the body of the algorithm, whenever a new merged record m is created (line 10), it is checked to see if it is dominated by any record in R or R'. If so, then the newly merged record is immediately discarded, before it is used for any unnecessary comparisons. Note that we do *not* check if m dominates any other records, as this check would be expensive in the inner loop of the algorithm. Finally, because we do not incrementally check if m dominates other records, at the end of the algorithm we add a step to remove all dominated records from the output set.

Koosh-ND relies on two complex operations: removing all dominated records from a set and checking if a record is dominated by a member of a set. These seem like expensive operations that might outweigh the gains obtained by eliminating the comparisons of dominated records. However, using an inverted list index that maps label-value pairs to the records that contain them, we can make these operations quite efficient. Checking if a record is dominated takes time proportional to the number of label-value pairs in the record. Thus, the overhead in Koosh-ND of checking for dominated records has a cost linear in the size of $ER(R)$.

The correctness of Koosh-ND is established by the following theorem (proven in Appendix).

THEOREM 5.2. For any set of records R such that $NER(R)$ is finite, Koosh-ND terminates and computes $NER(R)$.

6 The Packages algorithm

In Section 3, we illustrated why ER with confidences is expensive, on the records r_1 and r_2 that merged into r_3:

$r_1 = 0.8[name : \text{Alice}, areacode : 202]$,

$r_2 = 0.7[name : \text{Alice}, phone : \text{555-1212}]$,

$r_3 = 0.56[name : \text{Alice}, areacode : 202, phone : \text{555-1212}]$.

Recall that r_2 cannot be discarded essentially because it has a higher confidence than the resulting record r_3. However, notice that other than the confidence, r_3 contains more label-value pairs, and hence, if it were not for its higher confidence, r_2 might not be necessary. This observation leads us to consider a scenario where the records minus confidences can be resolved efficiently, and then to add the confidence computations in a second phase.

In particular, let us assume that our merge function is "information preserving" in the following sense: When a record r merges with other records, the information carried by r's attributes is not lost. We formalize this notion of "information" by defining a relation "$\sqsubseteq$": $r \sqsubseteq s$ means that the attributes of s carry more information than those of r. We assume that this relation is transitive. Note that $r \sqsubseteq s$ and $s \sqsubseteq r$ does *not* imply that $r = s$; it only implies that $r.\mathcal{A}$ carries as much information as $s.\mathcal{A}$.

The property that merges are information preserving is formalized as follows:

- *Property P1:* If $r \approx s$ then $r \sqsubseteq \langle r, s\rangle$ and $s \sqsubseteq \langle r, s\rangle$.
- *Property P2:* If $s \sqsubseteq r$, $s \approx x$ and $r \approx x$, then $\langle s, x\rangle \sqsubseteq \langle r, x\rangle$

For example, a merge function that unions the attributes of records would have properties P1 and P2. Such functions are common in "intelligence gathering" applications, where one wishes to collect all information known about entities, even if contradictory. For instance, say two records report different passport numbers or different ages for a person. If the records merge (e.g., due evidence in other attributes) such applications typically gather all the facts, since the person may be using fake passports reporting different ages.

Furthermore, we assume that adding information to a record does not change the outcome of match. In addition, we also assume that the match function does not consider confidences, only the attributes of records. These characteristics are formalized by:

- *Property P3:* If $s \sqsubseteq r$ and $s \approx x$, then $r \approx x$.

Having a match function that ignores confidences is not very constraining: If two records are unlikely to match due to low confidences, the merge function can still assign a low confidence to the resulting record to indicate it is unlikely. The second aspect of Property P3, that adding information does not change the outcome of a match, rules out "negative evidence:" adding information to a record cannot rule out a future match. However, negative information also can be handled by decreasing the confidence of the resulting record. For example, say that record

r_3 above is compared to a record r_4 with area code "717". Suppose that in this application the differing area codes make it very unlikely the records match. Instead of saying the records do not match, we do combine them but assign a very low probability to the resulting record.

To recap, properties P1, P2 and P3 may not always hold, but we believe they do occur naturally in many applications, e.g., intelligence gathering ones. Furthermore, if the properties do not hold naturally, a user may wish to consider modifying the match and merge functions as described above, in order to get the performance gains that the properties yield, as explained next.

Incidentally, note that the $\sqsubseteq$ relation is different from the domination order $\leq$ defined earlier. For example, if $r_1 = 0.7[a, b, c]$ and $r_2 = 0.9[a, b]$, we have that $r_2 \sqsubseteq r_1$ but it is *not* true that $r_2 \leq r_1$. That is, r_2 does carry more information in its attributes that r_1, but does not dominate r_1 due to the lower confidence.

Note that Properties P1, P2, P3 imply representativity as defined in [2] (this follows from Lemma 6.1, given in Appendix).

The algorithm of Figure 3 exploits these properties to perform ER more efficiently. It proceeds in two phases: a first phase bypasses confidences and groups records into disjoint packages. Because of the properties, this first phase can be done efficiently, and records that fall into different packages are known not to match. The second phase runs ER with confidences on each package separately. We next explain and justify each of these two phases.

```
1: input: a set R of records
2: output: a set R' of records, R' = ER(R)
3: Define for Packages:
4: match: p ≈ p' iff r(p) ≈ r(p')
5: merge: ⟨p, p'⟩ = p'' :
          with root: r(p'') = ⟨r(p), r(p')⟩
          and base: b(p'') = b(p) ∪ b(p')
6: Phase 1:
7: P ← ∅
8: for all records rec in R do
9:    create package p:
          with root: r(p) = rec
          and base: b(p) = {rec}
10:   add p to P
11: end for
12: compute P' = ER(P) (e.g., using Koosh) with
    the following modification: Whenever packages
    p, p' are merged into p'', delete p and p' immedi-
    ately, then proceed.
    Phase 2:
13: R' ← ∅
14: for all packages p ∈ P' do
15:   compute Q = ER(b(p)) (e.g. using Koosh)
16:   add all records in Q to R'
17: end for
18: return R'
```

Algorithm 3: The Packages algorithm

6.1 Phase 1

In Phase 1, we use any ER algorithm to resolve the base records, but with some additional bookkeeping. For example, when two base records r_1 and r_2 merge into r_3, we combine all three records together into a *package* p_3. The package p_3 contains two things: (i) a root $r(p_3)$ which in this case is r_3, and (ii) the base records $b(p_3) = \{r_1, r_2\}$.

Actually, base records can also be viewed as packages. For example, record r_2 can be treated as package p_2 with $r(p_2) = r_2$, $b(p_2) = \{r_2\}$. Thus, the algorithm starts with a set of packages, and we generalize our match and merge functions to operate on packages.

For instance, suppose we want to compare p_3 with a package p_4 containing only base record r_4. That is, $r(p_4) = r_4$ and $b(p_4) = \{r_4\}$. To compare the packages, we only compare their roots: That is, $M(p_3, p_4)$ is equivalent to $M(r(p_3), r(p_4))$, or in this example equivalent to $M(r_3, r_4)$. (We use the same symbol M for record and package matching.) Say these records do match, so we generate a new package p_5 with $r(p_5) = \langle r_3, r_4 \rangle$ and $b(p_5) = b(p_3) \cup b(p_4) = \{r_1, r_2, r_4\}$.

The package p_5 represents, not only the records in $b(p_5)$, but also any records that can be derived from them. That is, p_5 represents all records in $ER(b(p_5))$. For example, p_5 implicitly represents the record $\langle r_1, r_4 \rangle$, which may have a higher confidence that the root of p_5. Let us refer to the complete set of records represented by p_5 as $c(p_5)$, i.e., $c(p_5) = ER(b(p_5))$. Note that the package does not contain $c(p_5)$ explicitly, the set is just implied by the package.

The key property of a package p is that the attributes of its root $r(p)$ carry more information (or the same) than the attributes of any record in $c(p)$, that is for any $s \in c(p)$, $s \sqsubseteq r(p)$. This property implies that any record u that does *not* match $r(p)$, cannot match any record in $c(p)$.

THEOREM 6.3. For any package p, if a record u does not match the root $r(p)$, then u does not match any record in $c(p)$.

This fact in turn saves us a lot of work! In our

example, once we wrap up base records r_1, r_2 and r_4 into p_5, we do not have to involve them in any more comparisons. We only use $r(p_5)$ for comparing against other packages. If p_5 matches some other package p_8 (i.e., the roots match), we merge the packages. Otherwise, p_5 and p_8 remain separate since they have nothing in common. That is, nothing in $c(p_5)$ matches anything in $c(p_8)$.

COROLLARY 6.4. Consider two packages p, q that do not match, i.e., $M(r(p), r(q))$ is false. Then no record $s \in c(p)$ matches any record $t \in c(q)$.

As an aside, observe that $r(p)$ has a confidence associated with it. However, this confidence is not meaningful. Since the match function does not use confidences, $r(p).\mathcal{C}$ is not really used. As a matter of fact, there can be a record in $c(p)$ with the same attributes as $r(p)$ but with higher confidence. (In our running example, the root of p_5 was computed as $\langle\langle r_1, r_2\rangle, r_4\rangle$, but this may be different from $\langle\langle r_1, r_4\rangle, r_2\rangle$. The latter record, which is in $c(p)$ may have a higher confidence than $r(p)$.)

6.2 Phase 2

At the end of Phase 1, we have resolved all the base records into a set of independent packages. In Phase 2 we resolve the records in each package, now taking into account confidences. That is, for each package p we compute $ER(b(p))$, using an algorithm like Koosh. Since none of the records from other packages can match a record in $c(p)$, the $ER(b(p))$ computation is completely independent from the other computations. Thus, we save a very large number of comparisons in this phase where we must consider the different order in which records can merge to compute their confidences. The more packages that result from Phase 1, the finer we have partitioned the problem, and the more efficient Phase 2 will be.

Note that Phase 2 correctly computes $ER(R)$ as long as Corollary 6.4 holds: records falling into one package never match records falling into another. Even if Phase 1 combined records that do not actually match (false positives), Phase 2 would still correctly compute $ER(R)$. This fact opens up a wider range of situations in which the Packages algorithm could apply. Even if the match and merge functions do not satisfy Properties P1, P2, and P3, there could exist a more permissive match function, and a corresponding merge function that do satisfy the three properties. In this case, Phase 1 could be run using the more permissive match and merge functions, and Phase 2, using the original match and merge functions would still correctly compute $ER(R)$.

An alternative to Phase 2 is to leave the packages un-expanded, and report them to the user. To the end user, a package may be more understandable than a large number of records with different confidences. If the user finds an interesting package, then she may selectively expand it or query it. For example, say a user sees a package containing records related to a particular person. In the records, the user may observe attributes *name* : Alice, *address* : 123 Main, *passport* : 789, and may then query for the confidence of the record [*name* : Alice, *address* : 123 Main, *passport* : 789]. The answer to the query will tell the user how likely it is there is an entity with these three attributes.

6.3 Example

To illustrate the two phases of our packages algorithm, consider the following example. Symbols a, b, c, d represent label-value pairs. Records match only if they contain the a attribute. The merge function unions all attributes, and the resulting confidence is the minimum of the confidences of the two merging records times 0.5. Initially there are 4 records:

- $r_1 = 0.9[a, b]$
- $r_2 = 0.6[a, c]$
- $r_3 = 0.8[a, d]$
- $r_4 = 0.9[c, d]$

First, the base records are converted to packages. For example, for the first record we create package p_1 with $b(p_1) = \{r_1\}$, $r(p_1) = r_1$. In Phase 1, say we first compare p_1 and p_2, yielding package p_5:

- $b(p_5) = \{r_1, r_2\}$; $r(p_5) = 0.3[a, b, c]$

Later, p_5 matches p_3, yielding p_6:

- $b(p_6) = \{r_1, r_2, r_3\}$; $r(p_5) = 0.15[a, b, c, d]$

Package p_4 does not match anything, so at the end of Phase 1 we have two packages, p_6 and p_4. In Phase 2, when we expand p_6 we obtain

- $c(p_6) = 0.9[a, b], 0.95[a, c], 0.8[a, d]$, $0.45[a, b, c]$, $0.4[a, b, d]$, $0.4[a, c, d]$, $0.2[a, b, c, d]$, $0.15[a, b, c, d]$

Note that the record $[a, b, c, d]$ has two confidences, depending on the order of the merges. The higher confidence $0.2[a, b, c, d]$ is obtained when r_1 and r_3 are merged first.

6.4 Packages-ND

To remove dominated records from the final result, we simply use Koosh-ND in Phase 2 of the Packages algorithm. Note that it is not necessary to explicitly remove dominated packages in Phase 1. To see this, say at some point in Phase 1 we have two packages,

p_1 and p_2 such that $r(p_1) \leq r(p_2)$, and hence $r(p_1) \sqsubseteq r(p_2)$. Then p_1 will match p_2 (by Property P3 and idempotence), and both packages will be merged into a single one, containing the base records of both.

7 Thresholds

Another possibility for reducing the resolution workload lies within the confidences themselves. Some applications may not need to know every record that could possibly be derived from the input set. Instead, they may only need to see the derived records that are above a certain confidence threshold.

Definition 7.1 *Given a threshold value T and a set of base records R, we define the above-threshold entity-resolved set, $TER(R)$ that contains all records in $ER(R)$ with confidences above T. That is, $r \in TER(R)$ if and only if $r \in ER(R)$ and $r.\mathcal{C} \geq T$.*

As we did with domination, we would like to remove below-threshold records, not after completing the resolution process (as suggested by the definition), but as soon as they appear. However, we will only be able to remove below-threshold records if they cannot be used to generate above-threshold records. Whether we can do that depends on the semantics of confidences.

That is, as mentioned earlier, models for the interpretation of confidences vary, and different interpretations will have different properties. Under some interpretations, two records with overlapping information might be considered as independent evidence of a fact, and the merged record will have a higher confidence than either of the two base records.

Other interpretations might see two records, each with their own uncertainty, and a match and merge process which is also uncertain, and conclude that the result of a merge must have lower confidence than either of the base records. For example, one interpretation of $r.\mathcal{C}$ could be that it is the probability that r correctly describes a real-world entity. Using the "possible worlds" metaphor [13], if there are N equally-likely possible worlds, then an entity containing at least the attributes of r will exist in $r.\mathcal{C} \times N$ worlds. With this interpretation, if r_1 correctly describes an entity with probability 0.7, and r_2 describes an entity with probability 0.5, then $\langle r_1, r_2 \rangle$ cannot be true in more worlds than r_2, so its confidence would have to be less than or equal to 0.5.

To be more formal, some interpretations, such as the example above, will have the following property.

- *Threshold Property:* If $r \approx s$ then $\langle r, s \rangle.\mathcal{C} \leq r.\mathcal{C}$ and $\langle r, s \rangle.\mathcal{C} \leq s.\mathcal{C}$.

Given the threshold property, we can compute $TER(R)$ more efficiently. The next lemma (proven in Appendix) tells us that if the threshold property holds, then all results can be obtained from above-threshold records. Thus, below-threshold records can be immediately dropped during the resolution process.

LEMMA 7.1. When the threshold property holds, every record in $TER(R)$ has a well-formed derivation tree consisting exclusively of above-threshold records.

7.1 Algorithms Koosh-T and Koosh-TND

As with removing dominated records, Koosh can be easily modified to drop below-threshold records. First, we add an initial scan to remove all base records that are already below threshold. Then, we simply add the following conjunct to the condition of Line 10 of the algorithm:

$$merged.\mathcal{C} \geq T$$

Thus, merged records are dropped if they are below the confidence threshold.

THEOREM 7.2. When $TER(R)$ is finite, Koosh-T terminates and computes $TER(R)$.

By performing the same modification as above on Koosh-ND, we obtain the algorithm Koosh-TND, which computes the set $NER(R) \cap TER(R)$ of records in $ER(R)$ that are neither dominated nor below threshold.

7.2 Packages-T and Packages-TND

If the threshold property holds, Koosh-T or Koosh-TND can be used for Phase 2 of the Packages algorithm, to obtain algorithm Packages-T or Packages-TND. In that case, below-threshold and/or dominated records are dropped as each package is expanded.

Note that we cannot apply thresholds in Phase 1 of the Packages algorithm. The confidence associated with the root of a package is not an upper bound. So even if $r(p).\mathcal{C}$ is below threshold, there can still be above-threshold records in $c(p)$. For instance, in the example of Section 6.3, $r(p_6).\mathcal{C} = 0.15$, but records $0.2[a, b, c, d]$ and $0.95[a, c]$ are in $c(p_6)$. We can, however, as a minor optimization discard below-threshold records in the base set of a package. Any such records will the thrown away at the beginning of Phase 2 (by Koosh-T), so there is no need to carry them.

8 Experiments

To summarize, we have presented three main algorithms: BFA, Koosh, and Packages. For each of those basic three, there are three variants, adding in thresholds (T), non-domination (ND), or both (TND). In this section, we will compare the three algorithms against each other using both thresholds and non-domination. We will also investigate how performance is affected by varying threshold values, and, independently, by removing dominated records.

To test our algorithms, we ran them on synthetic data. Synthetic data gives us the flexibility to carefully control the distribution of confidences, the probability that two records match, as well as other important parameters. Our goal in generating the data was to emulate a realistic scenario where n records describe various aspects of m real-world entities ($n > m$). If two of our records refer to the same entity, we expect them to match with much higher probability than if they referred to different entities.

To emulate this scenario, we assume that the real-world entities can be represented as points on a number line. Records about a particular entity with value x contain an attribute A with a value "close" to x. (The value is normally distributed with mean x, see below.) Thus, the match function can simply compare the A attribute of records: if the values are close, the records match. Records are also assigned a confidence, as discussed below.

For our experiments we use an "intelligence gathering" merge function as discussed in Section 6, which unions attributes. Thus, as a record merges with others, it accumulates A values and increases its chances of matching other records related to the particular real-world entity.

To be more specific, our synthetic data was generated using the following parameters:

- n, the number of records to generate
- m, the number of entities to simulate
- $margin$, the separation between entities
- σ, the standard deviation of the normal curve around each entity.
- μ_c, the mean of the confidence values

To generate one record r, we proceed as follows: First, pick a uniformly distributed random integer i in the range $[0, m-1]$. This integer represents the value for the real-word entity that r will represent. For the A value of r, generate a random floating point value v from a normal distribution with standard deviation σ and a mean of $margin \cdot i$. To generate r's confidence, compute a uniformly distributed value c in the range $[\mu_c - 0.1, \mu_c + 0.1]$ (with $\mu_c \in [0.1, 0.9]$ so that c stays in $[0, 1]$). Now create a record r with $r.\mathcal{C} = c$ and $r.\mathcal{A} = \{A : v\}$. Repeat all of these steps n times to create n synthetic records.

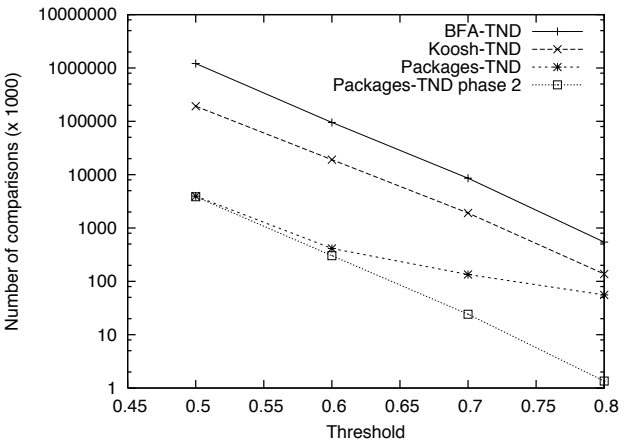

Figure 1: Thresholds vs. Matches

Our merge function takes in the two records r_1 and r_2, and creates a new record r_m, where $r_m.\mathcal{C} = r_1.\mathcal{C} \times r_2.\mathcal{C}$ and $r_m.\mathcal{A} = r_1.\mathcal{A} \cup r_2.\mathcal{A}$. The match function detects a match if for the A attribute, there exists a value v_1 in $r_1.\mathcal{A}$ and a value v_2 in $r_2.\mathcal{A}$ where $|v_1 - v_2| < k$, for a parameter k chosen in advance.

Naturally, our first experiment compares the performance of our three algorithms, BFA-TND, Koosh-TND and Packages-TND, against each other. We varied the threshold values to get a sense of how much faster the algorithms are when a higher threshold causes more records to be discarded. Each algorithm was run at the given threshold value three times, and the resulting number of comparisons was averaged over the three runs to get our final results.

Figure 1 shows the results of this first experiment. The first three lines on the graph represent the performance of our three algorithms. On the horizontal axis, we vary the threshold value. The vertical axis (logarithmic) indicates the number of calls to the match function, which we use as a measure of the work performed by the algorithms. The first thing we notice is that work performed by the algorithms grows exponentially as the threshold is decreased. Thus, clearly thresholds are a very powerful tool: one can get high-confidence results at a relatively modest cost, while computing the lower confidence records gets progressively more expensive! Also interestingly, the BFA-TND and Koosh-TND lines are parallel to each other. This means that they are consistently a constant factor apart. Roughly, BFA does 10 times the number of comparisons that Koosh does.

The Packages-TND algorithm is far more efficient

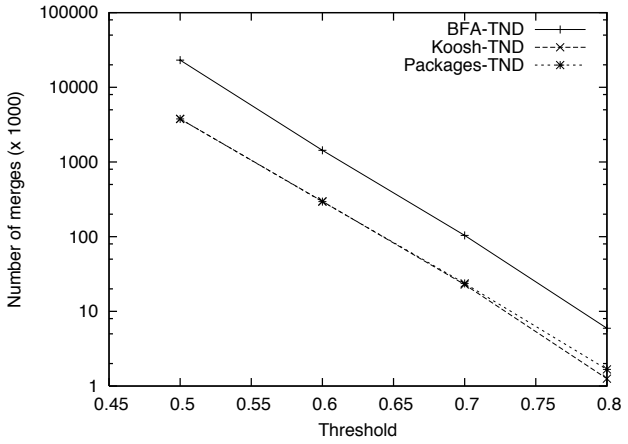

Figure 2: Thresholds vs. Merges

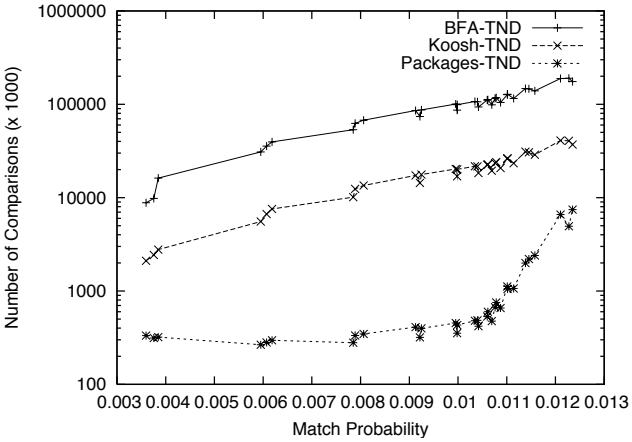

Figure 3: Selectivity vs. Comparisons

than the other two algorithms. Of course, Packages can only be used if Properties P1, P2 and P3 hold, but when they do hold, the savings can be dramatic. We believe that these savings can be a strong incentive for the application expert to design match and merge function that satisfy the properties. Note incidentally that the Packages line is not quite parallel to the other two. This is explained by the fact that the first phase of the Packages algorithm does not consider thresholds at all, and therefore it is a constant amount of work added to each data point. When we consider only the comparisons performed in the second phase of the algorithm, we get the fourth line in the graph, which is indeed parallel to the BFA and Koosh lines. This shows that the work done in the second phase is also a constant factor away from BFA and Koosh. In this case, the constant factor is at least 100.

We also compared our algorithms based on the number of merges performed. In Figure 2, the vertical axis indicates the number of merges that are performed by the algorithms. We can see that Koosh-TND and the Packages-TND are still a great improvement over BFA. BFA performs extra merges because in each iteration of its main loop, it recompares all records and merges any matches found. The extra merges result in duplicate records which are eliminated when they are added to the result set. Packages performs slightly more merges than Koosh, since the second phase of the algorithm does not use any of the merges that occurred in the first phase. If we subtract the Phase 1 merges from Packages (not shown in the figure), Koosh and Packages perform roughly the same number of merges.

In our next experiment, we compare the performance of our algorithms as we vary the probability

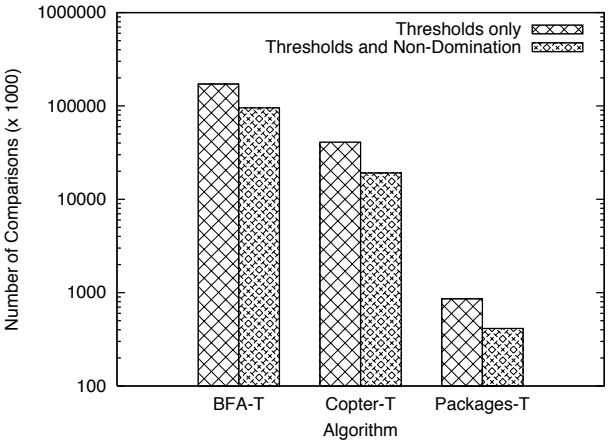

Figure 4: Effects of removing dominated records

that base records match. We can control the match probability by changing parameters k or σ, but instead of reporting our results in terms of those parameter values, we use the resulting match probability as the horizontal axis, which we believe provides more intuition. In particular, to generate Figure 3, we vary parameter k (keeping the threshold value constant at 0.6). During each run, we measure the match probability as the fraction of base record matches that are positive. (We have also computed the match probability over all matches; results are similar.) For each run, we then plot the match probability versus the number of calls to the match function, for our three algorithms.

As expected, the work increases with greater match probability, since more records are produced. Furthermore, we note that the BFA and Koosh lines are roughly parallel, but the Packages line stays level until a quick rise in the amount of work performed once the match probability reaches about 0.011. The Packages optimization takes advantage of the fact that records can be separated into packages that do not merge with one another. As the match probability increases, these packages quickly begin to merge into larger packages, which reduces the effectiveness of the optimization. If this curve were to continue, we would expect it to approach the Koosh line, and slightly surpass it. We expect this because when all records fall into one package, then the second phase of Packages is precisely Koosh on the set of base records.

In practice, we would expect to operate in the range of Figure 3 where the match probability is low and Packages outperforms Koosh. In our scenario with high match probabilities, records that refer to different entities are being merged, which means the match function is not doing its job. One could also get high match probabilities if there were very few entities, so that packages do not partition the problem finely. But again, in practice one would expect records to cover a large number of entities.

We ran a similar test obtaining different match probabilities by varying σ instead of k. This test revealed similar results, indicating that match probability is a good metric for predicting the performance of these algorithms.

Our results so far show the advantages of using thresholds and domination together, but do not illustrate the gains that using domination alone yields. These gains are explicitly shown in Figure 4 (threshold = 0.6). The results show that removing dominated records reduces the amount of work done by each algorithm by a factor of about 2.

9 Related work

Originally introduced by Newcombe et al. [15] under the name of record linkage, and formalized by Fellegi and Sunter [9], the ER problem was studied under a variety of names, such as Merge/Purge [12], deduplication [16], reference reconciliation [8], object identification [19], and others. Most of the work in this area (see [21, 11] for recent surveys) focuses on the "matching" problem, i.e., on deciding which records do represent the same entities and which ones do not. This is generally done in two phases: Computing measures of how similar atomic values are (e.g., using edit-distances [18], TF-IDF [6], or adaptive techniques such as q-grams [4]), then feeding these measures into a model (with parameters), which makes matching decisions for records. Proposed models include unsupervised clustering techniques [12, 5], Bayesian networks [20], decision trees, SVM's, conditional random fields [17]. The parameters of these models are learned either from a labeled training set (possibly with the help of a user, through active learning [16]), or using unsupervised techniques such as the EM algorithm [22].

All the techniques above manipulate and produce numerical values, when comparing atomic values (e.g. TF-IDF scores), as parameters of their internal model (e.g., thresholds, regression parameters, attribute weights), or as their output. But these numbers are often specific to the techniques at hand, and do not have a clear interpretation in terms of "confidence" in the records or the values. On the other hand, representations of uncertain data exist, which soundly model confidence in terms of probabilities (e.g., [1, 10]), or beliefs [14]. However these approaches focus on computing the results and confidences of exact queries, extended with simple "fuzzy" operators for value comparisons (e.g., see [7]), and are not capable of any advanced form of entity resolution. We propose a flexible solution for ER that accommodates any model for confidences, and proposes efficient algorithms based on their properties.

Our generic approach departs from existing techniques in that it interleaves merges with matches. The presence of "custom" merges is an important part of ER, and it makes confidences non-trivial to compute. The need for iterating matches and merges was identified by [3] and is also used in [8], but their record merges are simple aggregations (similar to our "information gathering" merge), and they do not consider the propagation of confidences through merges.

10 Conclusion

Entity Resolution (ER) is at the heart of any information integration process. Often the incoming data is inaccurate, and the match and merge rules for resolution yield uncertain results. While there has been work on ER with confidences, most of the work to date has been couched in terms of a specific application or has not dealt directly with the problems of performance and scalability. In this paper we do look at ER with confidences as a "generic database" problem, where we are given black-boxes that compare and merge records, and we focus on efficient algorithms that reduce the number of calls to these boxes.

The key to reducing work is to exploit generic properties (like the threshold property) than an application may have. If such properties hold we can use the optimizations we have studied (e.g., Koosh-T when the threshold property holds). Of the three optimizations, thresholds is the most flexible one, as it gives us a "knob" (the threshold) that one can control: For a high threshold, we only get high-confidence records, but we get them very efficiently. As we decrease the threshold, we start adding lower-confidence results to our answer, but the computational cost increases. The other two optimizations, domination and packages, can also reduce the cost of ER very substantially but do not provide such a control knob.

References

[1] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[2] O. Benjelloun, H. Garcia-Molina, J. Jonas, Q. Su, and J. Widom. Swoosh: A generic approach to entity resolution. Technical report, Stanford University.

[3] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proc. of the SIGMOD 2004 Workshop on Research Issues on Data Mining and Knowledge Discovery*, June 2004.

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

[5] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, Tokyo, Japan, 2005.

[6] William Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18:288–321, 2000.

[7] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[8] X. Dong, A. Y. Halevy, J. Madhavan, and E. Nemes. Reference reconciliation in complex information spaces. In *Proc. of ACM SIGMOD*, 2005.

[9] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[10] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[11] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. Technical Report 03/83, CSIRO Mathematical and Information Sciences, 2003.

[12] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of ACM SIGMOD*, pages 127–138, 1995.

[13] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[14] Suk Kyoon Lee. An extended relational database model for uncertain and imprecise information. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 211–220. Morgan Kaufmann, 1992.

[15] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[16] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD*, Edmonton, Alberta, 2002.

[17] Parag Singla and Pedro Domingos. Object identification with attribute-mediated dependences. In *Proc. of PKDD*, pages 297 – 308, 2005.

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

[19] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8):635–656, 2001.

[20] Vassilios S. Verykios, George V. Moustakides, and Mohamed G. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12(1):28–40, 2003.

[21] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1999.

[22] W. E. Winkler. Using the EM algorithm for weight computation in the fellegi-sunter model of record linkage. *American Statistical Association, Proceedings of the Section on Survey Research Methods*, pages 667–671, 1988.

A Proofs

LEMMA 3.2. Given a set R, every record in $ER(R)$ is well-formed.

Proof (sketch) Given a set S that satisfies the properties of Definition 3.1 and contains records that are not well-formed, one can show that removing the non-well-formed records from S yields a smaller set that still satisfies the properties of the definition. □

THEOREM 3.3 The solution to the ER problem is unique.

Proof Suppose that S_1 and S_2 are both solutions but different. Consider a record $r \in S_1$ that is not in S_2. Since r is well-formed (previous lemma) and in S_1, it has a derivation tree D with base records as leaves. Since S_2 is also an ER solution, then every internal record in D, as well as the root r, must be in S_2, a contradiction. Thus, every record in S_1 must be in S_2. We can similarly show the converse, and hence S_1 must equal S_2, a contradiction. □

THEOREM 3.4. For any set of records R such that $ER(R)$ is finite, BFA terminates and correctly computes $ER(R)$.

Proof First, note that any record added to R' has a well-formed derivation tree, so at any time in BFA, $R' \subseteq ER(R)$. From this fact we see that BFA must terminate when $ER(R)$ is finite. (If BFA does not terminate, then N is non-empty at the end of every iteration (line 14). This means that at least one new element from $ER(R)$ is added at each iteration, which is impossible if $ER(R)$ is finite.)

Since we know that $R' \subseteq ER(R)$ when BFA terminates, we only need to show that $ER(R) \subseteq R'$ to prove that $R' = ER(R)$ when BFA terminates.

Thus, we next show that every record $r \in ER(R)$ is generated by BFA. Since $r \in ER(R)$, r has a well-formed derivation tree D.

We define the level of a node $x \in D$, $l(x)$, as follows: If x is a base record ($x \in R$), then $l(x) = 0$. Otherwise, $l(x) = 1 + \max(l(c_1), l(c_2))$, where c_1 and c_2 are the children of x in D.

Clearly, all D records at level 0 are in the initial Z set (Step 1 of BFA). All level 1 records will be generated in the first iteration of BFA. Similarly, all level 2 records will be added to Z in the second iteration, and so on. Thus, record r will be added in the j^{th} iteration, where j is the level of r. □

Koosh

LEMMA 4.1. At any point in the Koosh execution, $R' \subseteq ER(R)$. If $ER(R)$ is finite, Koosh terminates.

Proof The records in R' will either be base records, which are well-formed by definition, or they will be new records generated by the algorithm. Koosh only generates new records using the merge function, and therefore all new records will have a derivation tree. Therefore, all records in R' must be in $ER(R)$.

Suppose that Koosh does not terminate. At the end of every iteration (Line 15) a new record is added to R'. This record cannot already be in R', so R' grows by 1 at every iteration. Thus, R' will be infinite. Since $R' \subseteq ER(R)$, then $ER(R)$ must also be infinite, a contradiction. □

LEMMA 4.2. When Koosh terminates, all records in the initial set R are in R'.

Proof Each iteration of the loop removes one record from R and adds it to R'. Records are never removed from R in any other case. Koosh terminates when $R = \emptyset$. Therefore, when Koosh terminates, all records in the initial set R have been moved to R'. □

LEMMA 4.3. Whenever Koosh reaches line 4:

1. all pairs of distinct records in R' have been compared
2. if any pair matches, the merged record is in R or R'.

Proof We prove this by induction. The base case will be before the first iteration, when $R' = \emptyset$, and therefore the condition holds. Each iteration removes one record r from R and adds it to R'. Directly before adding r to R', r is compared to all of the records in R'. This, in combination with the inductive hypothesis, guarantees us the first part of the condition. Furthermore, any record that matches r causes the creation of a merged record. The merged record is added to R unless it is already in R', or if it is the same record as r. In the first two cases, the merged record will end up in either R or R'. In the last case, the merged record is the same as r, and r gets added to R' at the end of the iteration. Therefore, all the new merged records will end up in R or R'. In combination with the inductive hypothesis, we can conclude that at the end of the iteration, all pairs of matching records in R' have their merged records in R or R'. Therefore we have proven the inductive step and can conclude the Lemma holds after each iteration of the loop. □

THEOREM 4.4. For any set of records R such that $ER(R)$ is finite, Koosh terminates and correctly computes $ER(R)$.

Proof By Lemma 4.1 we know that Koosh terminates, so let S be the set returned by Koosh. We know from Lemma 4.2 that $R \subseteq S$. When Koosh terminates, the R set is empty. Therefore, by Lemma 4.3, for all $r_1, r_2 \in S$, if $r_1 \approx r_2$ then $\langle r_1, r_2 \rangle \in S$. Finally, since all records in S are well-formed (Lemma 4.1), we know that S is the smallest set that has those two properties. Therefore, the conditions of Definition 3.1 are satisfied and $S = ER(R)$. □

LEMMA 4.5. For any two records r_1 and r_2, Koosh will never compare them more than once.

Proof When a record r is removed from R, it has not been compared with any records. It is compared once with all of the records in R' and then placed into R'. After this, the r is only compared with records in R as they are removed and placed into R'. Since Koosh never moves records back into R, and it doesn't allow records to be added to R if they are already in R', this guarantees that r will never be compared with the same record twice. Therefore, Koosh will never compare any pair of records more than once. □

THEOREM 4.6. Koosh is optimal, in the sense that no algorithm that computes $ER(R)$ makes fewer comparisons.

Proof Suppose there is an algorithm A which generates $ER(R)$ but performs fewer comparisons than Koosh. Then there exist two records $r_1, r_2 \in ER(R)$ that Koosh compares, but A does not compare. Now we construct new match and merge functions. Functions $match'$ and $merge'$ are the same as the original functions unless the two records are r_1 and r_2. In this case, $match'$ returns *true* and $merge'$ returns a new record k that is not in $ER(R)$ using the original match and merge functions.

It is clear that if *match* and *merge* satisfy our two properties of idempotence and commutativity, then $match'$ and $merge'$ also satisfy those properties. Using $match'$ and $merge'$, $k \in ER(R)$. But algorithm A never compares r_1 and r_2, so it cannot merge them to obtain k. Therefore, algorithm A does not generate $ER(R)$. This is a contradiction, so no algorithm that generates $ER(R)$ can perform fewer comparisons than Koosh. □

Domination

LEMMA 5.1. If the domination property holds, then every record in $NER(R)$ has a well-formed derivation tree with no dominated records.

Proof Consider $r \in NER(R)$. Suppose that r has a well-formed derivation tree D where one of the records x in D is dominated by a different record $x' \in ER(R)$. Consider a new derivation tree D' which is a modified version of D: First, we replace x by x' in D'. Next, we look at the parent of x in

D, say $p = \langle x, x_1 \rangle$, and replace p by $p' = \langle x', x_1 \rangle$ in D'. Note that because of the domination property, $p \leq p'$. Also note that $p' \in ER(R)$ since it has a well-formed derivation tree.

In a similar fashion, we replace the parent of p, and so on, until we replace r by r', where $r \leq r'$, and $r' \in ER(R)$.

If $r \neq r'$, then r is dominated by r', a contradiction since $r \in NER(R)$. Therefore $r = r'$. The same process can be repeated, until all dominated records are eliminated from the derivation tree of r. □

THEOREM 5.2. For any set of records R such that $NER(R)$ is finite, Koosh-ND terminates and computes $NER(R)$.

Proof We know that Koosh computes $ER(R)$ correctly. Koosh-ND is the same as Koosh except that it sometimes discards records that are dominated by another record. Given Lemma 5, we know that we can remove dominated records early, so Koosh-ND computes a superset of $NER(R)$. The final check removes all dominated records from the output set, guaranteeing that the result of Koosh-ND is equal to $NER(R)$. □

Packages

For the following proofs related to the Packages Algorithm, we assume Properties P1, P2, P3.

Notation: We denote by $B(r)$ the set of base records that appear as leaves in a derivation tree that produces r.

LEMMA 6.1. Consider two records r, s with derivation trees, such that such that $B(r) \subseteq B(s)$. Then $r \sqsubseteq s$.

Proof We denote by $D(r)$ the depth of the derivation tree of a record r. Base records have a depth of 0. The proof is by induction on the depth of the (to be proven) smallest record.

Induction hypothesis: Given two records r, s such that $B(r) \subseteq B(s)$ and an integer n, $0 \leq n$: If $D(r) \leq n$ then $r \sqsubseteq s$.

Base: $n = 0$. In this case, r is a base record that belongs to the derivation tree of s. Following the path from s to r in this derivation tree, each record has less or equivalent information than its parent node (due to P1), and therefore, by transitivity of $\sqsubseteq$, we have that $r \sqsubseteq s$.

Induction step: We now show that if the hypothesis holds for $n = k$, then it also holds for $n = k + 1$.

If $D(r) \leq k$, we can directly apply the induction hypothesis. The case to deal with is $D(r) = k + 1$. Consider the children of r in its derivation tree. $r = \langle r_1, r_2 \rangle$. Clearly, $D(r_1) \leq k$ and $D(r_2) \leq k$. Since $B(r_1) \subseteq B(s)$ and $B(r_2) \subseteq B(s)$, we can apply the induction hypothesis to r_1 and r_2. It follows that $r_1 \sqsubseteq s$, and $r_2 \sqsubseteq s$. Applying P2 and P3, we can inject r_2 to obtain that $\langle r_1, r_2 \rangle \sqsubseteq \langle s, r_2 \rangle$. Similarly, we obtain that $\langle r_2, s \rangle \sqsubseteq \langle s, s \rangle$. By merge commutativity and idempotence, and since $\sqsubseteq$ is transitive, it follows that $r \sqsubseteq s$. □

COROLLARY 6.2. If a record u matches a record $s \in c(p)$, then $u \approx r(p)$.

Proof The root $r(p)$ always has a valid derivation tree using all records in $b(p)$, i.e., $B(r(p)) = b(p)$. Since $s \in c(p)$, s has a valid derivation tree, and $B(s) \subseteq b(p)$. Thus, since $B(s) \subseteq B(r(p))$, by Lemma 6.1, $s \sqsubseteq r(p)$. By Property P3, if $u \approx s$, then $u \approx r(p)$. □

THEOREM 6.3. For any package p, if a record u does not match the root $r(p)$, then u does not match any record in $c(p)$.

Proof Suppose that u matches a record $s \in c(p)$. By Corollary 6.2, $u \approx r(p)$, a contradiction. □

COROLLARY 6.4. Consider two packages p, q that do not match, i.e., $M(r(p), r(q))$ is false. Then no record $s \in c(p)$ matches any record $t \in c(q)$.

Proof Suppose that there is an $s \in c(p)$ that matches a $t \in c(q)$. Using Corollary 6.2, we see that s matches $r(q)$. Now, since $r(q)$ matches an element in $c(p)$, we can use the corollary a second time to show that $r(q)$ must match $r(p)$, which is a contradiction. □

Thresholds

LEMMA 7.1. When the threshold property holds, every record in $TER(R)$ has a well-formed derivation tree consisting exclusively of above-threshold records.

Proof Consider $r \in TER(R)$. Since r is also in $ER(R)$, it has a well-formed derivation tree, call it D. Suppose that one of the records x in D is below threshold, $x.\mathcal{C} \leq T$. By the threshold property, the parent of x is below threshold. The parent of the parent of x is also below threshold, and we continue in this fashion until we show that the root itself, r is below threshold, a contradiction. Thus, r has a well-formed derivation tree with above-threshold records. □

THEOREM 7.2. When $TER(R)$ is finite, Koosh-T terminates and computes $TER(R)$.

Proof Koosh computes $ER(R)$, and the only changes made to Koosh-T remove records that are below threshold. Because of Lemma 7.1, we can remove below-threshold records at any point without

missing records in $TER(R)$. Therefore, Koosh-T generates a superset of $TER(R)$. Since below-threshold records are removed from the initial set, and because Koosh-T drops all newly formed records that are below threshold, Koosh-T cannot generate any records that are not in $TER(R)$. Therefore, Koosh-T correctly computes $TER(R)$. □