Joint Entity Resolution

Makoto Tachinaba* Stanford University

Hector Garcia-Molina Stanford University

January 13, 2009

Abstract

Entity resolution (ER) is the process of matching records that represent the same real-world entity and then merging them. We consider the ER problem for two related datasets. In the datasets, a record in one can refer to a record in the other and an ER process running on one set can affect an ER process on the other. We formalize the joint ER model for datasets which reference each other by treating the match and merge functions as black boxes. We identify important properties for match and merge functions that, if satisfied, allow much more efficient ER. We provide four algorithms that run Entity Resolution for a pair of datasets. We show that our parallel algorithms require shorter runtime than naive alternate algorithms. We also introduce improvements for our parallel algorithms which result in fewer feature comparisons.

1 Introduction

Entity Resolution (ER) (sometimes referred to as deduplication) is the process of identifying and merging records judged to represent the same real-world entity. For example, two companies that merge may want to combine their customer records: for a given customer that dealt with the two companies they create a composite record that combines the known information.

Most of the previous work considered ER techniques for a single class dataset. However, in practice, many data integration tasks need to tackle complex information spaces where datasets of multiple classes and relationships between the datasets exist. In this paper, we assume there are two data sets to be resolved. Those data sets are related, which means a record in one data set can refer to a record in the other data set. ER algorithm for those datasets is complex because a ER process for one dataset may affect other ER process.

For concreteness, in this paper we focus on a type of ER processing called generic pair-wise. In this case, a domain expert writes two functions, a match and a merge function. The pair-wise match rule $M(r_1, r_2)$ evaluates to true when two records r_1 and r_2 are determined to represent the same entity. If $M(r_1, r_2)$ is true, then a merge function is used to create the composite record $\langle r_1, r_2 \rangle$. Note that after a merge we may identify new matches with other records. For example, the combined information in $\langle r_1, r_2 \rangle$ may match with a third record r_3, while neither r_1 nor r_2 had enough information to generate a match with r_3.

The alternative to pair-wise ER is generally some type of global *clustering* strategy that groups records that are similar and are deemed to represent the same real-world entity [16, 4]. Briefly, pair-wise may be easier to implement since the domain expert only needs to consider two records at a time, and pair-wise may be more amenable to incremental and distributed processing [1]. Clustering approaches may yield more accurate results.

Several works [14, 15, 7, 5] have addressed algorithms which handles two datasets and they have focused on accuracy of matching and how to compare records. By contrast, we focus on performance and treat the functions for comparing and merging records as black-boxes. We will show efficient parallel algorithms for jointly comparing and merging $\langle r_1, r_2 \rangle$ those data sets.

In summary, in this paper we make the following contributions:

*Visiting from NEC Corporation

- We formalize the generic ER problem for two related dataset (Section 3).
- We identify the ICAR properties of match and merge functions for this problem that lead to efficient strategies (Section 4.2).
- We present parallel ER algorithms for two related dataset (Section 5).
- We experimentally evaluate the algorithms using two sets of related data, one on papers and another on their authors. Our algorithms result in good performance in terms of the number of feature comparisons and runtime (Section 6).

2 Motivating Example

To illustrate joint ER, consider two data sets X and Y, where X is a set of books and Y is a set of authors. Books in X have attributes Title and WrittenBy. Authors in Y have attributes Name, Affiliation and Publication. The two data sets are related because values in the WrittenBy attribute refer to Y author names, and vice versa. For example, X may contain the tuple, $\langle Database\ book, H.Garcia\text{-}Molina\rangle$ and Y may contain the tuple, $\langle H.Garcia\text{-}Molina, Stanford\ University, Database\ book\rangle$.

In this example, the rules for determining when authors (or books) represent the same real world entity are interrelated. For instance, if two records in X have identical title, or similar titles and were written by the same author, we can assume that they refer the same entity. Similarly, if two records in Y have identical name, or similar name and identical affiliation and those authors wrote the same books, we can assume that they refer the same entity. For every record in X we need to refer to Y to find the affiliation of its author. Moreover, for every record in Y we need to refer to X to find the names of books its author wrote.

3 Model

A match rule M_I determines if two records r_1 and r_2 in the record set I refer to the same real-world entity. If the records match, $M_I(r_1, r_2) = true$, we denote this as $r_1 \approx r_2$; otherwise, if $M_I(r_1, r_2) = false$, we denote this $r_1 \neq r_2$. M_I results always true or false; we do not consider approximate matches (with an associated confidence value).

A merge rule μ_I merges two records in the record set I into one. The function is only defined for matching records. The result of $\mu_I(r_1, r_2)$ is denoted $\langle r_1, r_2\rangle$.

Let X and Y be the sets of records to be resolved. We discuss entity resolution from the point of view of the X records; the case for the Y records in symmetrical. All the functions for X that we define (which have an X subscript) have a Y analogue.

A record $r \in X$ has a field F that refers to one or more Y records. We refer to this field as $F_X(r)$. Note that $F_X(r) \subseteq Y$. (We could extend our model to include a confidence value with each link $s \in F_X(r)$, representing how confident we are that s is the Y record associated with r. These confidences could be taken into account by the match function when deciding if two records match. We do not discuss such an extension here.)

When $F_X(r)$ contains a single element we omit the set brackets and simply write $F_X(r) = s$ ($s \in Y$). We use the notation $F_X^{-1}(r)$ to refer to the Y records whose F_Y field contains r. Note that $F_X^{-1}(r)$ can be implemented in a variety of ways. For instance, we can scan Y searching for r references, which would be expensive. We could also keep back pointers in X records, i.e., materialize the $F_X^{-1}(r)$ values. The latter option would involve an update cost when Y records change.

For set X, we have a match function $M_X(r_1, r_2)$ that returns true if records $r_1, r_2 \in X$ represent the same real world entity. We assume that the match function is of the following form:

$M_X(r_1, r_2) = B_X(r_1, r_2) \vee [L_X(r_1, r_2) \wedge C(F_X(r_1), F_X(r_2))]$

The "base" comparison function B_X can determine if r_1 and r_2 match without consulting any Y records. For example, if two products have almost identical names, prices and codes, then we assume they are the same, regardless of their manufacturer (represented by Y records). If the B_X function cannot determine that r_1 and r_2 match on their own, then the second part of the match function checks if they match due to common references to Y records. For example, we may say that two products match if they are manufactured by the same company (the

F_X fields point to a single common manufacturer), and if L_X says the products names are roughly the same.

There are two important issues to discuss regarding the clause $C(F_X(r_1), F_X(r_2))$ in the match function.

First, there are two natural choices for the C comparison function. One is a function C_1 that checks if all elements in the sets match. For example, say a book record r_1 in X refers to two authors in Y, s_1 and s_2. Similarly, a second book r_2 refers to authors, i.e., $F_X(r_2) = \{s_3, s_4\}$. The condition $C_1(F_X(r_1), F_X(r_2))$ is true only if (a) s_1 equals either s_3 or s_4, and (b) s_2 equals the remaining Y author.

$$C_1(F_X(r_1), F_X(r_2)) \equiv F_X(r_1) = F_X(r_2)$$

The second natural condition, C_2 checks that a subset of $F_X(r_1)$ matches a subset of $F_X(r_2)$. In the previous example, if s_1 or s_2 equals either s_3 or s_4 then the condition is true.

$$C_2(F_X(r_1), F_X(r_2)) \equiv F_X(r_1) \cap F_X(r_2) \neq \emptyset$$

The second issue regarding the comparison function is how F values are updated. In particular, during entity resolution we assume that as Y records are merged, the F_X values are updated. For instance, say $F_X(r_1) = s_1$ and $F_X(r_2) = s_2$. If s_1 and s_2 merge into a new record s_3, then $F_X(r_1) = F_X(r_2) = s_3$. This change may then make r_1 and r_2 match in set X. Note that these updates can be implemented in a variety of ways. For example, with an eager strategy, as soon as s_1 and s_2 merge, we can update all X records in $F_Y^{-1}(s_1)$ and $F_Y^{-1}(s_2)$. With a lazy strategy, we do not update the X records but instead keep a *translation table* T that maps records to their current merged record. For instance, before the merge $T(s_1) = s_1$ and after the merge $T(s_1) = s_3$. In this lazy case, to perform the comparison $F_X(r_1) = F_X(r_2)$ in the match function, we actually perform $T(F_X(r_1)) = T(F_X(r_2))$ (where T operates on a set in the obvious way).

When two X records r_1 and r_2 merge, a composite record $r_3 = \langle r_1, r_2 \rangle$ is created by the merge function. The merge function creates an $F_X(r_3)$ value. For now we do not make any assumptions about $F_X(r_3)$. However, the most natural value for $F_X(r_3)$ would be $F_X(r_1) \cup F_X(r_2)$.

Given match and merge functions for X, we can define the resolved set of records $ER_X(X)$ as done in [1]. We also have analogous functions for the Y set and a definition of $ER_Y(Y)$.

We define the joint resolution of X and Y as an iterative process, where we alternate resolving one set and the other until we reach a fixed point.

```
ER(X, Y):
  done := false;
  X' := X;  Y' := Y;
  while not done do
    [ newX := ER_X(X'); newY := ER_Y(Y');
      done := (newX = X') and (newY = Y')
      X' := newX; Y' := newY  ]
   return (X', Y');
```

Note that the above is simply our definition of a correct joint resolution. In what follows we investigate efficient ways to perform such joint resolution (showing that the more efficient strategy yields the same correct results as the above definition). We also study properties of the match and merge functions that may make the resolution more efficient.

4 Properties

4.1 Basic Properties

We assume four basic properties called ICAR properties for M_I and μ_I: idempotence, commutativity, representativity and associativity. (We use I in M_I and μ_I to represent either X or Y.) These properties ware addressed in [2]. Idempotence says that any record matches itself, and merging a record with itself yields the same record. Commutativity says that, if r_1 matches r_2, then r_2 matches r_1. Additionally, the merged results of r_1 and r_2 should be identical regardless of the merge ordering. The meaning of the representativity property is that record r_3 obtained from merging two records r_1 and r_2 "represents" the original records, in the sense that any record r_4 that would have matched r_1 (or r_2 by commutativity) will also match

r_3. Intuitively, this property states that there is no "negative evidence": merging two records r_1 and r_2 cannot create evidence (in the merged record r_3) that would prevent r_3 from matching any other record that would have matched r_1 or r_2.

- Idempotence: $\forall r; r \approx r$ and $\langle r, r \rangle = r$
- Commutativity: $\forall r_1, r_2, r_1 \approx r_2$ iff $r_2 \approx r_1$, and if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$
- Associativity: $\forall r_1, r_2, r_3$ such that $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$ exist, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$
- Representativity: If $r_3 = \langle r_1, r_2 \rangle$ then for any r_4 such that $r_1 \approx r_4$, we also have $r_3 \approx r_4$.

In [2] we argue that many match and merge rules naturally satisfy the ICAR properties. However, now our match and merge function have specific components, so in the next sub-section we study what properties these components (i.e., B, L, C) must satisfy in order to meet the ICAR properties.

4.2 Properties in Two Datasets

Merge Function: To maintain representativity, $F_X(\langle r_1, r_2 \rangle)$ must at least contain $F_X(r_1) \cup F_X(r_2)$; if the merge function removes any records in $F_X(r_1)$ or $F_X(r_2)$, representativity might no longer hold.

The proof of the above property, as well as other properties given in this section, are given in Appendix A.

Since it does not make sense for the merge function to *add* links not already in r_1 or r_2, to satisfy the above property we will assume that $F_X(\langle r_1, r_2 \rangle)$ is equal to $F_X(r_1) \cup F_X(r_2)$.

Idempotence: For M_X to be idempotent, B_X must be idempotent. Note that this is regardless of L_X and C.

Commutativity: If B_X and L_X are commutative, M_X must be commutative. This property holds regardless of whether C is implemented as C_1 or as C_2. (becasue both C_1 and C_2 are commutative.)

Associativity: As in the general case, the merge function μ_I should be associative.

Representativity: The conditions for representativity are different for C_1 and C_2. For C_2, if B_X and L_X are representative and $B_X \Rightarrow L_X$, then M_X must be representative. Note that $B_X \Rightarrow L_X$ means B_X is a stronger condition than L_X. For example, if $B_X(r_1, r_2)$ is $similarity(r_1, r_2) \geq 0.9$, $L_X(r_1, r_2)$ could be $similarity(r_1, r_2) \geq 0.8$. For C_1, in addition to the previous condition, the records must satisfy $B_X \Rightarrow C_1$. It is not possible to satisfy this last property in practice, since the content of the F sets would have to be determined by the outcome of the B_X function. Thus, the ICAR properties cannot be expected to hold if C_1 is used.

To summarize, for the ICAR properties to hold, the following conditions must be met:

- Merge: $F_X(\langle r_1, r_2 \rangle) = F_X(r_1) \cup F_X(r_2)$
- Idempotence: $\forall r; B_X(r, r) = \texttt{true}$, $\langle r, r \rangle = r$
- Commutativity: $\forall r_1, r_2, B_X(r_1, r_2) = B_X(r_2, r_1)$, and $L_X(r_1, r_2) = L_X(r_2, r_1)$, and if $M_X(r_1, r_2) = \texttt{true}$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
- Associativity: $\forall r_1, r_2, r_3$ such that $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$ exist, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$
- Representativity: If $r_3 = \langle r_1, r_2 \rangle$ then
  - For any r_4 such that $B_X(r_1, r_4) = \texttt{true}$, we also have $B_X(r_3, r_4) = \texttt{true}$
  - For any r_4 such that $L_X(r_1, r_4) = \texttt{true}$, we also have $L_X(r_3, r_4) = \texttt{true}$.
  - For any r_1 and r_4 such that $B_X(r_1, r_4) = \texttt{true}$, we also have $L_X(r_1, r_4) = \texttt{true}$
  - In case C_1 is used, if $B_X(r_1, r_4) = \texttt{true}$, then we mst have $C_1(F_X(r_1), F_X(r_4)) = \texttt{true}$.

5 Resolution

5.1 Problem of Naive ER Algorithm

In section 3, we showed an algorithm which alternates running Entity Resolution on two data sets. That algorithm has two performance problems.

First, the algorithm of section 3 may need a large amount of time; each ER process must run to completion

before the next may begin. To improve response time, we will show the following algorithm, which runs Entity Resolution on X and Y concurrently.

Second, the algorithm may perform redundant comparisons. For example, say there are a records where $r_1, r_2 \in X$, $s_1, s_2 \in Y$, $F_X(r_1) = s_1$, $F_X(r_2) = s_2$, $B_X(r_1, r_2) = \texttt{false}$, $L_X(r_1, r_2) = \texttt{true}$, $M_Y(s_1, s_2) = \texttt{false}$. At the comparison in the first R-Swoosh execution on X, records r_1 and r_2 do not match, because $C_2(F_X(r_1), F_X(r_2))$ is false. At the comparison in the next R-Swoosh execution on X after doing so on Y, records r_1 and r_2 will not match because both $F_X(r_1)$ and $F_X(r_2)$ are not changed and $C_2(F_X(r_1), F_X(r_2))$ will be false. In the example, $M_X(r_1, r_2)$ is computed twice, and the computation is redundant. Section 5.4 shows a modification that eliminates redundancy.

5.2 Preliminaries

To explain the new algorithms, consider records r_1, r_2, s_1, s_2 where $r_1, r_2 \in X$ and $s_1, s_2 \in Y$. Furthermore, say $F_X(r_1) = \{s_1\}$, $F_X(r_2) = \{s_2\}$. Say we want to run ER on X and Y.

We make the standard assumptions about our parallel computing model: we assume that no messages exchanged between processors are lost.

5.3 Running R-Swoosh in Parallel

Our Simple Parallel ER (SPER) algorithm consists of three code segments, shown as Algorithms 1, 2, 3.

Each processor for X and Y runs procedure $ER(I)$ detailed in Algorithms 1, where I is the input set of records (either X or Y). Except for lines 6, 25, 26, 29-31, $ER(I)$ is the Swoosh algorithm of [2], which assumes the ICAR properties. In this code, set I' contains the records that are *tentatively* in the result. Each record $currentRecord$ in I is considered in turn, and if it does not merge with any I' records, $currentRecord$ is moved to I'.

However, activity in the concurrent ER process may lead to a new local match (because the F sets changed), and hence a record already in I' may have to be moved back to I so it can be reconsidered in light of new information. In particular, it is possible for a match result $M_I(r_1, r_2)$ to change from `false` to `true` when: $B_I(r_1, r_2) = \texttt{false}$, $L_I(r_1, r_2) = \texttt{true}$, $C_2(F_X(r_1), F_X(r_2)) = \texttt{false}$.

To inform the concurrent ER process of merges, the ER procedure sends UPDATE messages in line 25. Then, after each I record is processed (line 29), procedure ER checks for received messages from its concurrent process by calling procedure UPDATE (Algorithm 2).

To check for changes in local F sets, procedure ER calls function UPDATE after each I record is processed (line 29). To illustrate function UPDATE, say in our running example P_Y merges s_1 and s_2 to $s_3 = \langle s_1, s_2 \rangle$, so it sends P_X an update messages which includes $F_Y^{-1}(\langle s_1, s_2 \rangle)$ (=r_1,r_2)and the identity of s_3. When that message is received (line 3 of UPDATE) at P_X, if either r_1 or r_2 happen to be in I' (= X'), they are moved back to I (=X) since these records need to be compared again, in light of the new information. Also, in line XX the F sets are updated locally using a lazy strategy.

The concurrent ER processes finish when neither one has new merged records to report and all outstanding UPDATE messages have been fully processed. The termination function (Algorithm 3) checks these conditions. When P_X finished processing all records in X (line 5), it sends a termination message with the number of messages it sent and retrieved. When P_Y receives a termination message, it examines the contents and if it knows that all those messages were processed then it completes execution.

5.4 Enhanced Parallel ER Algorithm

We will show how the joint ER algorithm can be modified to improve efficiency by explaining how an ER process sends and reacts to messages.

First, we consider the possibility of reducing the number of record comparisons. When ER processes apply a Simple Parallel algorithm to the example given above, P_X compares records in X ,r_1, r_2, to all records in X' ,r_3, and moves the records in X to X'. Then, after receiving messages, P_X moves r_1 and r_2 from X' back to X. Finally, P_X compares r_1 and r_2 to r_3 again. However, only the comparison of r_1 and r_2 could match. All other comparisons between r_1 and r_3 will not match, and are redundant.

To avoid this redundant comparison, we will show how a process can react to messages more efficiently. Just be-

fore moving X' records back to X, P_X runs R-Swoosh for the records which refer to merged Y records. If by doing so, P_X produces new records, then P_X adds them to X. If instead P_X deletes records, then it deletes them from X'. In the process, P_X will not retrieve messages. This algorithm is shown in Algorithm 4. In the example given above, after P_X pops the update messages, P_X runs R-Swoosh for r_1, r_2, and produces $\langle r_1, r_2 \rangle$. P_X then adds $\langle r_1, r_2 \rangle$ to X, and deletes r_1 and r_2 from X'. Note that P_X does not compare r_1 and r_3 again.

Second, we consider the possibility of reducing the number of feature comparisons in a record comparison. We use a cache to reduce the number of feature comparisons. When records do not match, they may be compared again. However, once an ER process compares the records, the evaluation of B_I and L_I will not change so the process will not need to be repeated. In our algorithm, if L_I is false, an ER process memoizes the combination of records in a table. Before comparing records, if an ER process finds the combination of records in the table, it knows the evaluation is false without computing B_I and L_I. Algorithm 5 is called as a match function from Algorithm 1 and update the cache named $NOMATCH_I$. Algorithm 6 is called as a match function from Algorithm 4 and use the cache.

6 Experiments

We implemented the Alternate ER algorithm given in section 3 and the Simple Parallel ER algorithm, the Enhanced Parallel ER Algorithm, and the Enhanced Parallel ER algorithm with cache given in section 5. We used the *cora* citation data set. We compared the number of merges performed and messages sent by the four algorithms while varying the thresholds of B_I and L_I. We also compared the number of comparisons by and runtime of the algorithms while varying the thresholds of B_I and L_I. Finally, we conducted scalability tests.

6.1 Experimental Setting

We ran our experiments on a citation dataset. We retrieved two data sets from the *cora* dataset: author and paper. The author dataset includes an ID, an author name, an institution, and a paper title, which refers to paper records. The paper dataset includes an ID, a title, a venue, and an author name, which refers to author records. B_I and L_I evaluate the lexical similarity of each attribute other than ID and compare the similarity to a threshold. The B_I threshold is larger than L_I threshold so that B_I and L_I meet the representativity condition, $B_I \rightarrow L_I$, then they satisfy ICAR properties. We fixed $k = 2$, $C_2(F_X(r_1), F_X(r_2)) \equiv F_X(r_1) \cap F_X(r_2) \neq \emptyset$. We implemented F_X for author records as a set of IDs of paper records whose title is the exactly same as the title in the author record. An approximate match between the titles in the author and paper records might be required, however we do not talk in this paper. This may be a future work. We also implemented F_X for paper records as a set of IDs of author records whose author name is the exactly same as the author name in the paper record.

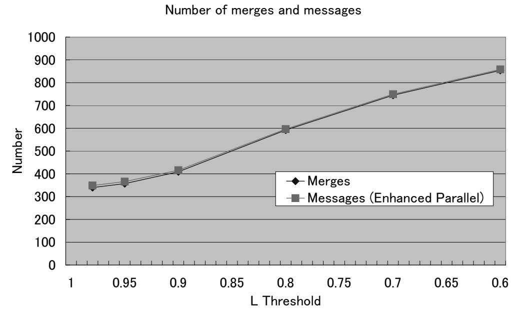

Figure 1: Merges and Messages Plotted Against L_I for the Paper Dataset

The algorithms were implemented in Java, and our experiments were run on four 2.4GHz Intel Core processor with 4GB of memory.

6.2 Merges and Messages

We measured the number of merges performed in and messages sent by a ER process by varying the thresholds of either B_I or L_I.

Figure 1 and 2 shows the number of merges and messages while varying the L_I threshold. When we varied the L_I threshold, we fixed the B_I threshold to 1.0. Figure 3 shows the number of merges while varying the B_I threshold. When we varied the B_I threshold, we fixed the L_I threshold to 0.7. Lower thresholds of B_I and L_I resulted

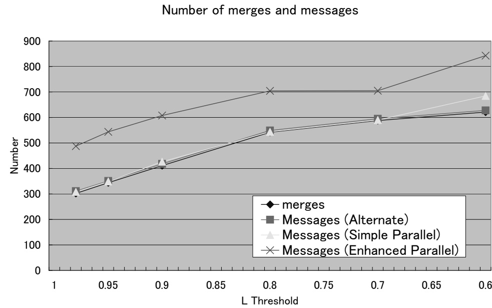

Figure 2: Merges and Messages Plotted Against L_I for the Author Dataset

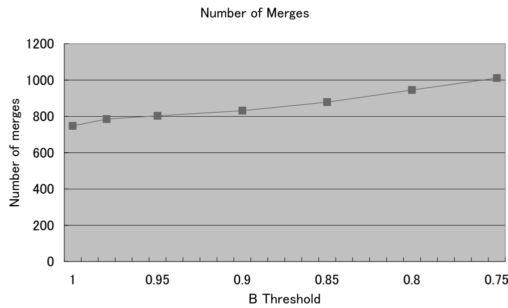

Figure 3: Merges and Messages Plotted Against B_I for the Paper Dataset

in a higher number of merges because processes produced more new records. The number of messages corresponded to the number of merges because when records were merged processes sent messages. Relatively few termination messages were observed. The fact that the Enhanced Parallel algorithm sent more messages than the alternate algorithm seems inefficient however it resulted in a shorter runtime. We will explain why below.

6.3 Comparisons and Runtimes

We measured the number of feature value comparisons for each algorithm and runtimes of the four algorithms by varying the L_I thresholds for both the author and paper datasets. We started measuring the runtime when a ER process start R-Swoosh algorithm, and we finished measuring it when a ER process end by termination protocol

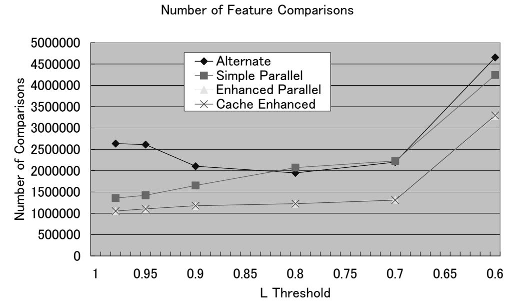

Figure 4: Feature Comparisons for the Paper Dataset

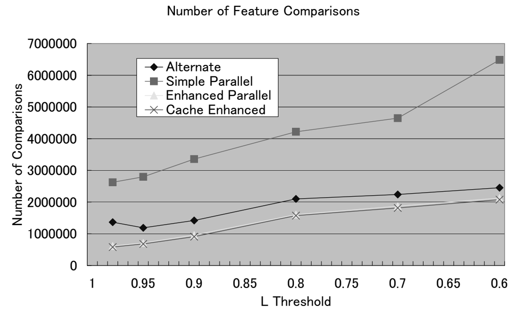

Figure 5: Feature Comparisons for the Author Dataset

which we showed. We fixed B_I threshold to 1.0 in this experiment.

Figure 4 shows the number of feature comparisons for paper records. In general, the number increased for lower thresholds because lower thresholds produced more new composite records. However, when the threshold was high, more than 0.8, the number of feature comparisons decreased for lower thresholds in the alternative algorithm. The reason why is that processes run R-Swoosh repeatedly for larger record sets. The Enhanced Parallel ER algorithm performed 40.1% of the comparisons of the alternate algorithm for a high threshold (0.98). For lower thresholds, the relative advantage over the alternate algorithm decreased. However it still performed only 70.3% of the comparisons of the alternate algorithm for a low threshold (0.6).

Figure 5 shows the number of feature comparisons for author records. Notice that the number of feature com-

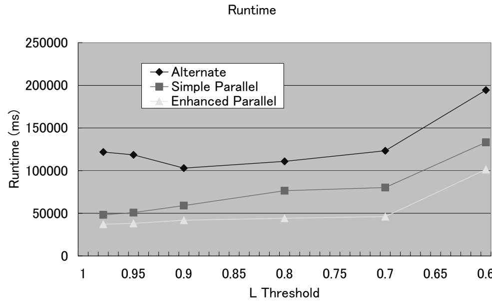

Figure 6: Runtime for the Paper Dataset

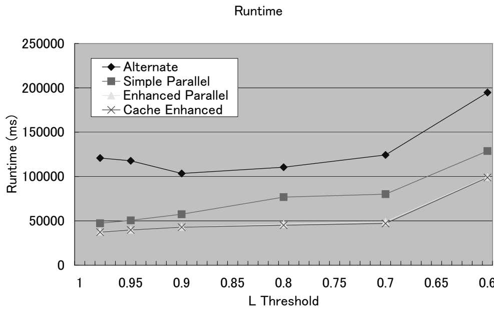

Figure 7: Runtime for the Author Dataset

parisons for the Simple Parallel ER algorithm was much larger than for the other algorithms. The reason why is that processes repeated comparisons every time records matched. The plot also shows the Enhanced Parallel ER algorithm reduced redundant comparisons; the number of comparisons was between 23.5 and 40.3% of the Simple Parallel algorithm's.

These plots also shows that the use of a cache did not affect the number of feature comparisons. The difference between the Enhanced Parallel ER algorithm with and without cache was between 95 and 100%.

Figures 6 and 7 show runtime. The graphs are nearly identical because our termination protocol required both ER processes to terminate at the same time. For large thresholds (0.98), the Enhanced Parallel ER algorithm required 30.9% of the runtime of the alternate algorithm. For thresholds as low as 0.6, the runtime was still almost 50.8%. Our Enhanced Parallel algorithm was sig-

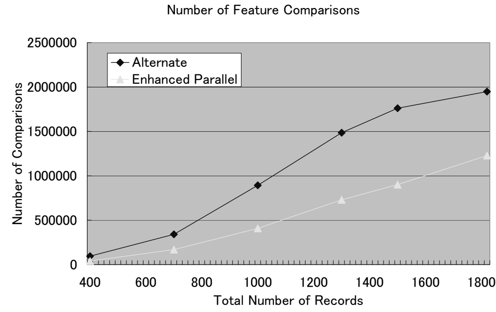

Figure 8: Scalability Test of Feature Comparisons for the Paper Dataset

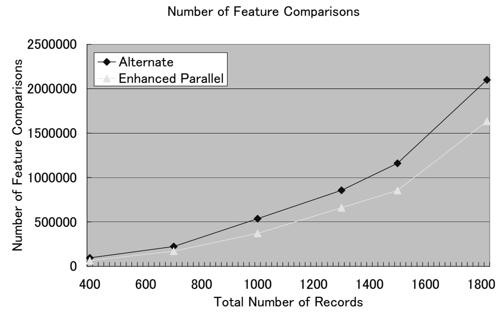

Figure 9: Scalability Test of Feature Comparisons for the Author Dataset

nificantly faster in terms of runtime.

6.4 Scalability

We conducted scalability tests for the alternate algorithm and for the Enhanced Parallel ER Algorithm. We measured the number of feature comparisons and runtimes by varying the number of input record both for paper and author records. Input records were selected randomly from a paper dataset containing 1037 records and an author dataset containing 780 records. The B_I and L_I thresholds were fixed to 1.0 and 0.8 respectively.

Figure 8 and 9 shows the number of comparisons made by each algorithm. The plot shows that the Enhanced Parallel ER algorithm performed fewer comparisons than the alternate algorithm. For very few number of input records near 400, the Enhanced Parallel ER algorithm

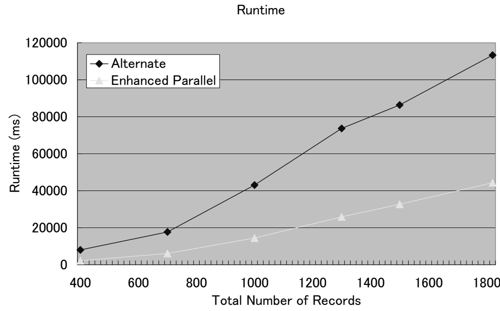

Figure 10: Scalability Test of Runtime for the Paper Dataset

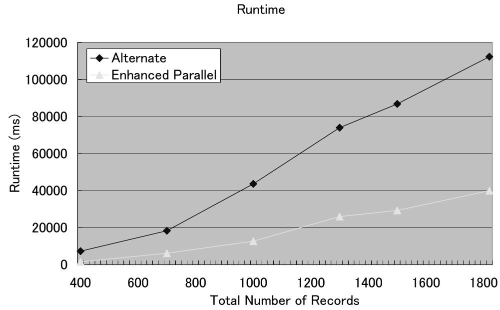

Figure 11: Scalability Test of Runtime for the Author Dataset

performed very fewer comparisons than the alternate algorithm, 42.6% for paper records and 66.8% for author records. For larger number of record more than 700, the improvement rates were almost same around 50% for paper records and 70% for author records.

Figure 10 and 11 shows rumtimes of each algorithm. The plots shows that the Enhanced Parallel ER algorithm performed faster than the alternate algorithm. For very few number of input records near 400, the Enhanced Parallel ER algorithm performed very faster than the alternate algorithm, 27.2% for paper records and 21.3% for author records. For larger number of record more than 700, the improvement rates were almost same around 35% for both records.

7 Related Work

Entity Resolution has been studied under various names including record linkage [17], merge/purge [12], deduplication [18], reference reconciliation [8], object identification [19], and more (see [21, 11] for recent surveys).

Most approaches focus on matching: accurately finding records that represent the same entity, using a variety of techniques such as Fellegi and Sunter's probabilistic linkage rules [9], Bayesian networks [20], or clustering [16] [4]. Our approach encapsulates the behavior of such complex decision processes into a Boolean match function that decides whether two records represent the same entity or not. Iterative approaches [3] [8] identify the need for a feedback loop that compares merged records in order to discover more matches. Our ER algorithm provides a general framework where match and merge are black-box functions.

In the parallel computing literature, [1, 13] introduced a parallel algorithm for single dataset based on the Swoosh algorithm. [6, 10] used parallel algorithms as well.

Several works [14, 15, 7, 5] have addressed algorithms which handles two datasets. [14] introduced a parallel algorithm for two datasets. However the datasets it handles belong to same class. By contrast, our target datasets belong to different classes. [15, 7, 5] presented collective models for different datasets. However their research focused on accuracy rather than runtime. Although they demonstrated that making multiple ER decisions collectively can provide better accuracy than historical approaches, however it is believed that the approach is expensive.

8 Future Work

Future work might proceed in several directions. First, one might consider how to run ER for three or more datasets. Second, one might consider more flexible implementation of $C(F_X(r_1), F_X(r_2))$ which can handle similarity of confidence values, because $C_1(F_X(r_1), F_X(r_2)) \equiv F_X(r_1) = F_X(r_2)$ may be too strict and $C_2(F_X(r_1), F_X(r_2)) \equiv F_X(r_1) \cap F_X(r_2) \neq \emptyset$ may be too lax. Finally, one might investigate approximate reference between two datasets.

9 Conclusion

In this paper, we formalized the problem of Entity Resolution for two related different datasets. In the datasets, a record in one can refer to a record in the other and an ER process affect to other ER process.

We also provided four algorithms that run ER jointly for two datasets. The alternate algorithm is the most straightforward, but it results in long runtimes. The Simple Parallel algorithm is efficient in terms of runtime, but it performs more redundant comparisons than the alternate algorithm. The Enhanced Parallel algorithm is an efficient way to reduce the number of comparisons and runtime. In our experiments, it required 31% of the runtime of the alternate algorithm. Adding a cache to the Enhanced Parallel algorithm did not affect its performance.

Finally, we presented four important properties for match and merge functions for two datasets that lead to significantly more efficient ER. We argued that these properties should guide the development of match and merge functions.

Acknowledgments

We acknowledge the supports made by Steven Euijong Whang and David Menestrina. We would also like to thank the other InfoLab members at Stanford University and many members of NEC Corporation and NEC Laboratories America and NEC Corporation of America who supported me. We also thank Eric Schkufza.

A Appendix: Proofs

Proposition 1. *Idempotence: If B_X and μ_X are idempotent, then idempotence holds.*

Proof. If we know $B_X(r,r) = \texttt{true}$, then we know $M_X(r,r) = \texttt{true}$. $\langle F_X(r), F_X(r)\rangle = F_X(r) \cup F_X(r) = F_X(r)$. Therefore, because $M_X(r,r) = \texttt{true}$ and $\langle r, r\rangle = r$ by definition, the idempotence holds. □

Proposition 2. *Commutativity: If B_X and L_X and μ_X are commutative, then commutativity holds.*

Proof. For C_1, $(F_X(r_1) = F_X(r_2)) = (F_X(r_2) = F_X(r_1))$. Similarly, for C_2, $F_X(r_1) \cap F_X(r_2) = F_X(r_2) \cap F_X(r_1)$, so C_1 and C_2 are always commutative, then M_X are commutative. In terms of merge, F_X is commutative because $F_X(r_1) \cup F_X(r_2) = F_X(r_2) \cup F_X(r_1)$. Therefore the records are always commutative. □

Proposition 3. *Associativity: If μ_X is associative, the records are associative.*

Proof. $\{\{F_X(r_1) \cup F_X(r_2)\} \cup F_X(r_3)\} = \{F_X(r_1) \cup \{F_X(r_2) \cup F_X(r_3)\}\}$, then F_X is associative, so if the μ_X is associative, the records are always associative. □

Proposition 4. *Representativity: For C_2, if B_X and L_X are representative and $B_X \Rightarrow L_X$, then representativity holds. For C_1, if B_X and L_X are representative and $B_X \Rightarrow L_X$ and $B_X \Rightarrow C_1$, then representativity holds.*

If $r_3 = \langle r_1, r_2\rangle$ then for any r_4 such that $B_X(r_1, r_4) = \texttt{true}$, we also have $B_X(r_3, r_4) = \texttt{true}$, and for any r_4 such that $L_X(r_1, r_4) = \texttt{true}$, we also have $L_X(r_3, r_4) = \texttt{true}$, and for any r_1 and r_4 such that $B_X(r_1, r_4) = \texttt{true}$, we also have $L_X(r_1, r_4) = \texttt{true}$ and in the case of C_1, we also have $C_1(F_X(r_1), F_X(r_4)) = \texttt{true}$.

Proof. We begin by showing C is representative. To do that, we must consider two cases. For C_2, if $C_2(F_X(r_1), F_X(r_4)) = \texttt{true}$, $F_X(r_1) \cap F_X(r_4) \neq \emptyset$ and $F_X(r_1) \cap F_X(r_4) \subseteq F_X(r_1)$. $F_X(r_3) = F_X(r_1) \cup F_X(r_2)$, then $F_X(r_1) \subseteq F_X(r_3)$, so $F_X(r_1) \cap F_X(r_4) \subseteq F_X(r_3)$. Then $F_X(r_4) \cap F_X(r_3) \neq \emptyset$ and $C_2(F_X(r_3), F_X(r_4)) = \texttt{true}$. We will consider for C_1 later.

We now consider four cases in turn.

If $B_X(r_1, r_2)$ and $B_X(r_1, r_4)$ equal to $\texttt{true}$, then $B_X(r_3, r_4)$ equal to $\texttt{true}$, because B_X is representative. Then, $r_3 \approx r4$.

If $L_X(r_1, r_2)$ and $C_2(F_X(r_1), F_X(r_2))$ and $L_X(r_1, r_4)$ and $C_2(F_X(r_1), F_X(r_4))$ equal to $\texttt{true}$, then $L_X(r_3, r_4) = \texttt{true}$, because L_X is representative. $C_2(F_X(r_3), F_X(r_4))$ also equal to $\texttt{true}$, as we explain above. Then, $r_3 \approx r4$.

If $B_X(r_1, r_2)$ and $L_X(r_1, r_4)$ and $C_2(F_X(r_1), F_X(r_4))$ equal to $\texttt{true}$, then $L_X(r_1, r_2) = \texttt{true}$, because $B_X \Rightarrow L_X$. $L_X(r_3, r_4)$ and $C_2(F_X(r_3), F_X(r_4))$ equal to $\texttt{true}$ because of same reason as previous explanation. Then, $r_3 \approx r4$.

If $L_X(r_1, r_2)$ and $C_2(F_X(r_1), F_X(r_2))$ and $B_X(r_1, r_4)$ equal to `true`, then $L_X(r_1, r_4) =$ `true`, because $B_X \Rightarrow L_X$. $L_X(r_3, r_4)$ and $C_2(F_X(r_3), F_X(r_4))$ equal to `true` because of same reason as previous explanation. Then, $r_3 \approx r4$.

We now consider four cases for C_1.

If $B_X(r_1, r_2)$ and $B_X(r_1, r_4)$ equal to `true`, then $B_X(r_3, r_4)$ equal to `true`, because B_X is representative. Then, $r_3 \approx r4$.

If $L_X(r_1, r_2)$ and $C_1(F_X(r_1), F_X(r_2))$ and $L_X(r_1, r_4)$ and $C_1(F_X(r_1), F_X(r_4))$ equal to `true`, then $L_X(r_3, r_4) =$ `true`, because L_X is representative. $C_1(F_X(r_3), F_X(r_4))$ also equal to `true`, because $F_X(r_3) = F_X(r_1) \cap F_X(r_2) = F_X(r_1) = F_X(r_4)$. Then, $r_3 \approx r4$.

If $B_X(r_1, r_2)$ and $L_X(r_1, r_4)$ and $C_1(F_X(r_1), F_X(r_4))$ equal to `true`, then $L_X(r_1, r_2) =$ `true`, because $B_X \Rightarrow L_X$, and then $C_1(F_X(r_1), F_X(r_2)) =$ `true`, because $B_X \Rightarrow C_1$. $L_X(r_3, r_4)$ and $C_1(F_X(r_3), F_X(r_4))$ equal to `true` because of same reason as previous explanation. Then, $r_3 \approx r4$.

If $L_X(r_1, r_2)$ and $C_2(F_X(r_1), F_X(r_2))$ and $B_X(r_1, r_4)$ equal to `true`, then $L_X(r_1, r_4) =$ `true`, because $B_X \Rightarrow L_X$, and then $C_1(F_X(r_1), F_X(r_4)) =$ `true`, because $B_X \Rightarrow C_1$. $L_X(r_3, r_4)$ and $C_1(F_X(r_3), F_X(r_4))$ equal to `true` because of same reason as previous explanation. Then, $r_3 \approx r4$.

As we mentioned above, in all case where $r_3 = \langle r_1, r_2 \rangle$ and $r_1 \approx r_4$, we always have $r_3 \approx r_4$, because we showed in all cases, $r_1 \approx r_4$.

□

A.1 Reference

References

[1] O. Benjelloun, H. Garcia-Molina, H. Kawai, TE Larson, D. Menestrina, and S. Thavisomboon. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. 2007.

[2] O. Benjelloun, H. Garcia-Molina, Q. Su, and J. Widom. Swoosh: A generic approach to entity resolution. *VLDB Journal*, 2008.

[3] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 11–18. ACM New York, NY, USA, 2004.

[4] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 865–876, 2005.

[5] A. Culotta, A. McCallum, and MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER SCIENCE. A Conditional Model of Deduplication for Multi-Type Relational Data, 2005.

[6] EW Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), 1965.

[7] P. Domingos and P. Domingos. Multi-Relational Record Linkage. 2004.

[8] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 85–96. ACM New York, NY, USA, 2005.

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

[10] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the Association for Computing Machinery*, 32(4):841–855, 1985.

[11] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions.

[12] M.A. Hernandez and S.J. Stolfo. The Merge/Purge Problem for Large Databases. In *SIGMOD Conference*, pages 127–138, 1995.

[13] H. Kawai, H. Garcia-Molina, O. Benjelloun, D. Menestrina, E. Whang, and H. Gong. P-Swoosh: Parallel Algorithm for Generic Entity Resolution.

Technical report, Technical report, Stanford University, 2006.

[14] H. Kim and D. Lee. Parallel linkage. pages 283–292, 2007.

[15] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. *Advances in Neural Information Processing Systems*, 17:905–912, 2005.

[16] A.E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. *Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[17] HB Newcombe, JM Kennedy, SJ Axford, and AP James. Automatic linkage of vital records. *Science*, 130:954–9, 1959.

[18] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–278. ACM New York, NY, USA, 2002.

[19] S. Tejada, C.A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems*, 26(8):607–633, 2001.

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

[21] William E Winkler and Nov P. Overview of record linkage and current research directions. Technical report, Bureau of the Census, 2006.

```
1:  input: record set X and Y
2:  output: a set X' and Y' of records, X' = ER(X), Y' = ER(Y)
3:  call ER(X) and ER(Y) together
4:
5:  ER(I):
6:  I' ← Ø, sentCount_I, recievedCount_I ← 0
7:  loop
8:    if I ≠ Ø then
9:      currentRecord ← a record from I
10:     remove currentRecord from I
11:     buddy ← null
12:     for all records r' in I' do
13:       result ← M_I(currentRecord, r')
14:       if result = true then
15:         buddy ← r'
16:         exit for
17:       end if
18:     end for
19:     if buddy = null then
20:       add currentRecord to I'
21:     else
22:       r'' ← ⟨currentRecord, buddy⟩
23:       remove buddy from I'
24:       add r'' to I
25:       SEND (UPDATE, F_I^{-1}(r''), (currentRecord, buddy, r''))
26:       sentCount_I ++
27:     end if
28:   end if
29:   UPDATE()
30:   if TERMINATION() = true then
31:     return I'
32:   end if
33: end loop
```

Algorithm 1: Parallel Joint ER Algorithm

```
1: UPDATE():
2: while UPDATE messages exist do
3:    pop UPDATE message → I'', (oldRecord_1, oldRecord_2, newRecord)
4:    T(oldRecord_1) ← newRecord
5:    T(oldRecord_2) ← newRecord
6:    recievedCount_I ++
7:    add I'' ∩ I' to I
8:    remove I'' ∩ I' from I'
9: end while
```

Algorithm 2: Simple Update Algorithm

```
1: UPDATE():
2: while UPDATE messages exist do
3:    pop UPDATE message → I'', (oldRecord_1, oldRecord_2, newRecord)
4:    T(oldRecord_1) ← newRecord
5:    T(oldRecord_2) ← newRecord
6:    recievedCount_I ++
7:    ER(I'' ∩ I') → I''', with no UPDATE
8:    add I''' − I' to I
9:    remove (I'' ∩ I') − I''' from I'
10: end while
```

Algorithm 4: Enhanced Update Algorithm

```
1: TERMINATE():
2: if I ≠ Ø ∨ UPDATE messages exist then
3:    return false
4: end if
5: SEND (TERMINATE, sentCount_I, receivedCount_I, P_J)
6: loop
7:    Wait for messages
8:    if UPDATE messages exist then
9:       return false
10:   end if
11:   if TERMINATE messages exist then
12:      pop TERMINATE message → sentCount_J, receivedCount_J
13:      if sentCount_J = receivedCount_I ∧ receivedCount_J = sentCount_I then
14:         return true
15:      end if
16:   end if
17: end loop
```

Algorithm 3: Termination Protocol

```
1: M_I(r_1, r_2):
2: if B_I(r_1, r_2) = true then
3:    return true
4: else if L_I(r_1, r_2) = false then
5:    add r_1 to NOMATCH_I(r_2)
6:    return false
7: else if C_k(F_I(r_1), F_I(r_2)) = true then
8:    return true
9: else
10:   return false
11: end if
```

Algorithm 5: Update Cache

```
1: M_I(r_1, r_2):
2: if r_1 ∈ NOMATCH_I(r_2) | r_2 ∈ NOMATCH_I(r_1) then
3:    return false
4: else
5:    return M_I(r_1, r_2) without NOMATCH_I
6: end if
```

Algorithm 6: Match Function with Cache