The VLDB Journal manuscript No.
(will be inserted by the editor)

Omar Benjelloun · Hector Garcia-Molina · David Menestrina · Qi Su · Steven Euijong Whang · Jennifer Widom

Swoosh: a generic approach to entity resolution

Received:

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. We formalize the generic ER problem, treating the functions for comparing and merging records as black-boxes, which permits expressive and extensible ER solutions. We identify four important properties that, if satisfied by the match and merge functions, enable much more efficient ER algorithms. We develop three efficient ER algorithms: G-Swoosh for the case where the four properties do not hold, and R-Swoosh and F-Swoosh that exploit the 4 properties. F-Swoosh in addition assumes knowledge of the "features" (e.g., attributes) used by the match function. We experimentally evaluate the algorithms using comparison shopping data from Yahoo! Shopping and hotel information data from Yahoo! Travel. We also show that R-Swoosh (and F-Swoosh) can be used even when the four match and merge properties do not hold, if an "approximate" result is acceptable.

Keywords Entity resolution · Generic entity resolution · Data cleaning

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. ER is a well-known problem that arises in many applications. For example, mailing lists may contain multiple entries representing the same physical address, but each record may be slightly different, e.g., containing different spellings or missing some information. As a second example, consider a company that has different customer databases (e.g., one for each subsidiary), and would like to consolidate them. Identifying matching records is challenging because there are no unique identifiers across databases. A given customer may appear in different ways in each database, and there is a fair amount of guesswork in determining which customers match.

Deciding if records match is often computationally *expensive* and *application specific*. For instance, a customer information management solution from a company [1] we have been interacting with uses a combination of nickname algorithms, edit distance algorithms, fuzzy logic algorithms, and trainable engines to match customer records. On the latest hardware, the speeding of matching records ranges from 10M to 100M comparisons per hour (single threaded), depending on the parsing and data cleansing options executed. A record comparison can thus take up to about 0.36ms, greatly exceeding the runtime of any simple string/numeric value comparison. How to match and combine records is also application specific. For instance, the functions used by that company to match customers are different from those used by others to match say products or DNA sequences.

In this paper we take a "generic approach" for solving ER, i.e., we do not study the internal details of the functions used to compare and merge records. Rather, we view these functions as "black-boxes" to be invoked by the ER engine. (Incidentally, there has been a lot of work done on the design of effective comparison and merge functions; see Section 6.) Given such black-boxes, we study algorithms for efficiently performing ER, i.e., we develop strategies that minimize the number of invocations to these potentially expensive black-boxes. In a way, our work is analogous to the design of efficient join algorithms, except that the operator we study is the ER operator. An important component of our work is that we identify a set of properties that, if satisfied by the match and merge functions, lead to significantly more efficient ER. For example, associativity of merges is one such important property: If merges are not associative, the order in which records are merged may impact the final result. An-

O. Benjelloun
Google Inc., Mountain View, CA 94043
E-mail: benjello@google.com

H. Garcia-Molina · D. Menestrina · Q. Su · S. E. Whang · J. Widom
Stanford University Computer Science Department, Stanford, CA 94305
E-mail: {hector, dmenest, qisu, euijong, widom}@cs.stanford.edu

[1] This company wishes to remain anonymous so that the performance numbers we give here are not associated with their product specifically.

other notable feature is that we do not perform the matching and merging separately, but tightly integrate them into a single process.

In this paper we focus on "pairwise ER," a common way to resolve records in the commercial world. In particular, the following assumptions are made:

- *Pairwise decisions.* Our black-box functions to match and merge records operate on two records at a time. Their operation depends solely on the data in these records, and not on the evidence in other records. In general, it is easier for application specialists to write pairwise record comparison and merge functions, as opposed to, say, functions that determine when a group of records may represent the same entity. Note that this requirement needs only be true at the time ER is performed, and does not preclude a prior training phase that considers the whole dataset, or a representative sample. (For example, a first phase can compute term frequencies for say all product descriptions, and the frequencies can then be used in comparing pairs of descriptions.) Thus, approaches based on machine learning can be leveraged to match or merge records.
- *No confidences.* We do not work with numeric similarity values or confidences. Record comparison functions may indeed compute numeric similarities (e.g., how close is this name to that name), but in the end they make a yes-no decision as to whether records match. Carrying confidences in the ER computations could in principle lead to more accurate decisions, but complicates processing significantly. For instance, one must decide how to combine confidences when records are merged. Also, confidences may decrease upon merges, which makes it more challenging to compare the information in merged records to that of base records. In a technical report [26], we study generic ER with confidences, in an extension of the framework presented here, where confidences are also handled by the black-box match and merge functions.
- *No relationships.* In our model, records contain all the information that pertains to each entity (See Figure 1 for an example). We do not consider a *separate* class of records that describe relationships between entities. Of course, some relationships can be represented in our model: for example, say Fred is Bill's brother. Then the record for Fred may contain the value "brother: {Bill}".
- *Consistent labels.* We assume that the input data has gone through a schema-level integration phase, where incoming data is mapped to a common set of well-defined labels. For instance, we assume that a "salary" label means the same thing, no matter what the source of the information is. However, we do not impose a rigid structure on records: we allow missing or multiple values for each label.

The particular variant of the ER problem that we study in this paper may not be the most sophisticated, but is used frequently in practice, at least in the commercial world. Indeed, IBM's recently introduced "DB2 Entity Analytic Solutions" [21] (formerly SRD) provides an exact, order insensitive solution to the ER problem (applied to human identities), which abstracts away from the particular functions used to compare values. Another leading commercial offering from Fair Isaac Corp. also encapsulates the match process as pairwise Boolean functions [10]. The customer information management company uses a pairwise matching framework to which a combination of comparison algorithms can be applied. Although these products have extra features, the core of their approach is the same as ours. In fact, their commercial success originally motivated our study of this particular approach to entity resolution (see Section 6 for an overview of alternative techniques).

	Name	**Phone**	**E-mail**
r_1	{John Doe}	{235-2635}	{jdoe@yahoo}
r_2	{J. Doe}	{234-4358}	
r_3	{John D.}	{234-4358}	{jdoe@yahoo}

Fig. 1 An instance of records representing persons

In summary, the ER variant we address here is relatively simple, but as we will see, can still be very expensive to compute. One fundamental cause of this complexity in ER is that record merges can lead to new matches. To illustrate, consider the records of Figure 1. Suppose that our black-box record match function works as follows: The function compares the name, phone and email values of the two records. If the names are very similar (above some threshold), the records are said to match. The records also match if the phone and email are identical. For matching records, the black-box merge function combines the names into a "normalized" representative, and performs a set-union on the e-mails and phone numbers. Note that phone and e-mail are being treated as a unit for comparison purposes. We call such a unit a *feature* (defined formally in Section 4). Thus, in this example, there are two features: one is "name" and the other is the pair "phone+ email".

For our example, the black-box comparison function determines that r_1 and r_2 match, but r_3 does not match either r_1 or r_2. For instance, the function finds that "John Doe" and "J. Doe" are similar, but finds "John D." not similar to anything (e.g., because John is a frequent first name). Thus, r_1 and r_2 merge into a new record r_4:

r_4	{John Doe}	{234-4358, 235-2635}	{jdoe@yahoo}

Now notice that r_4 now matches r_3 since the same phone and e-mail appear in both records. The combination of the information in r_1 and r_2 led us to discover a new match with r_3, therefore yielding an initially unforeseen merge. Thus, every time two records are merged, the combined record needs to be re-compared with "everything else".

Because record matching is inherently expensive, large sets of input records are often divided into "buckets" using application knowledge, and then ER is run on each bucket. For instance, if we are resolving products, we may be able to divide them using a "category" field. Thus, camera records will only be matched against other cameras, CDs will only

be matched against other CDs, and so on. If a record may match records in more than one category, then typically copies of the record are placed in multiple buckets. For example, a cell phone with a camera may be placed in the camera and the telephone buckets. (In our related work section we briefly mention other ways in which domain knowledge can be used to prune the search space.) In this paper we focus on resolving records within one bucket, that is, we study algorithms that must exhaustively consider all (within bucket) possible record matches. This type of exhaustive algorithm is invoked by a higher-level process that divides the data and decides what buckets need to be resolved. And since buckets can be quite large, it is still important to have as efficient an algorithm as possible for exhaustive ER. (Note that if the semantic function that divides records is imprecise, then overall matches may be missed, e.g., two wet-suits may be incorrectly placed in different buckets, say clothing and sporting goods. In this paper we do not consider the accuracy of the semantic function that partitions records.)

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

- We formalize the generic ER problem (Section 2). Unlike other works that focus only on identifying matching records (see related work in Section 6), we also include the process of merging records and how it may lead to new matches.
- We identify the ICAR properties (see Section 2.2) of match and merge functions that lead to efficient strategies.
- We present ER algorithms for three scenarios:
  - *G-Swoosh:* The most general ER algorithm, for the case where the 4 properties of match and merge functions do not hold (Section 3.1).
  - *R-Swoosh:* An algorithm that exploits the 4 properties of match and merge functions, and that performs comparisons at the granularity of records (Section 3.2).
  - *F-Swoosh:* An algorithm that also exploits the 4 properties, and uses feature-level comparison functions (Section 4.1). F-Swoosh avoids repeated feature comparisons and can be significantly more efficient than R-Swoosh.

  For each algorithm, we show that it computes the correct ER result and that it is "optimal" in terms of the number of comparisons performed. (What we mean by "optimal" varies by scenario and is precisely defined in each section.)
- We experimentally evaluate the algorithms using actual comparison shopping data from Yahoo! Shopping and hotel information data from Yahoo! Travel. Our results show that G-Swoosh can only be used on relatively small data sets when merges occur frequently, while R-Swoosh and F-Swoosh can handle substantially more data. Furthermore, when we know the features used for comparisons, we can use F-Swoosh and achieve between 1.1 and 11.4 performance improvement.
- Since G-Swoosh is so expensive, we investigate using R-Swoosh even when the ICAR properties of match and merge functions do *not* hold. In this case R-Swoosh does not produce the correct answer, but we show that what R-Swoosh produces is close to what G-Swoosh produces. Thus, if the application can tolerate an approximate answer, R-Swoosh and F-Swoosh are viable algorithms for all scenarios.

2 Fundamentals of Generic ER

We first consider entity resolution at the granularity of records. Our approach is very generic, since no assumption is made on the form or data model used for records. Finer granularity ER will be considered in Section 4.

2.1 Basic Model

We assume an infinite domain of records $\mathcal{R}$. An *instance* $I = \{r_1, \ldots, r_n\}$ is a finite set of records from $\mathcal{R}$.

A *match function* M is a Boolean function over $\mathcal{R} \times \mathcal{R}$, used to determine if two records r_1 and r_2 represent the same real-world entity (in which case $M(r_1, r_2) = \texttt{true}$). Such a match function reflects the restrictions we are making that (i) matching decisions depend solely on the two records being compared, and (ii) that such decisions are Boolean, and not associated with any kind of numeric confidence. In practice, such functions are easier to write than functions that consider multiple records.

A *merge function* μ is a partial function from $\mathcal{R} \times \mathcal{R}$ into $\mathcal{R}$, that captures the computation of merged records. Function μ is only defined for pairs of matching records (i.e., for r_1, r_2 s.t. $M(r_1, r_2) = \texttt{true}$).

When M and μ are understood from the context, $M(r_1, r_2) = \texttt{true}$ (resp. $M(r_1, r_2) = \texttt{false}$) is denoted by $r_1 \approx r_2$ (resp. $r_1 \not\approx r_2$), and $\mu(r_1, r_2)$ is denoted by $\langle r_1, r_2 \rangle$.

In order to define ER, we need to introduce two key intermediary notions: the merge closure of an instance, and record domination.

Merge closure Intuitively, given an instance I, we would like to find all pairs of matching records in I and merge them, using the match and merge functions defined above. The notion of extending I with all the records that can be derived this way is called the *merge closure* of I:

Definition 2.1 Given an instance I, the *merge closure* of I, denoted $\bar{I}$ is the smallest set of records S such that:

- $I \subseteq S$
- For any records $r_1, r_2 \in S$, if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle \in S$.

For any instance I, the merge closure of I clearly exists and is unique. It can be obtained as the fixpoint of adding to I merges of matching records.

Note that the merge closure of a (finite) instance I may be infinite. Intuitively, arbitrarily long chains of matches and merges may keep producing new records. However, the match and merge functions used in practice for ER do not exhibit such a behavior. We will give in Section 2.2 some simple properties, often satisfied by match and merge functions, which guarantee that the merge closure is finite.

Domination The merge closure is only a first step towards defining ER. The goal of ER is to determine the set of records that best represent some real-life entities. Intuitively, if two records r and r' are about the same entity but r holds more information than r', then r' is useless for representing this entity. In this case, we say that r' is *dominated* by r, denoted $r' \preceq r$. For instance, in our example, it is natural to consider that $r_1 \preceq r_4$, as r_4 contains all the values of r_1, and maybe also that $r_2 \preceq r_4$. Even though the name "J. Doe" does not appear in r_4 it can be considered as subsumed by "John Doe".

Formally, domination is defined to be any partial order relation on records (i.e., a reflexive, transitive and anti-symmetric binary relation). The choice of a specific partial order depends on the particular data and application at hand. Just like the match and merge functions, we view domination as a "black-box". Hence, our focus is not on the accuracy of the domination test. We will see in Section 2.2 that when the match and merge function have some simple and natural properties, then a canonical domination order can be defined using them.

Domination on records can be naturally extended to instances as follows:

Definition 2.2 Given two instances I_1, I_2, we say that I_1 *is dominated by* I_2, denoted $I_1 \preceq I_2$ if $\forall r_1 \in I_1, \exists r_2 \in I_2, \text{ such that } r_1 \preceq r_2$.

It is straightforward to verify that instance domination is a partial pre-order, i.e., that it is a reflexive and transitive relation. Instance domination is not a partial order because it is not anti-symmetric. Indeed, if $r_1 \preceq r_2$, the instances $\{r_2\}$ and $\{r_1, r_2\}$ are distinct yet dominate each other.

Entity Resolution We are now ready to define entity resolution formally:

Definition 2.3 Given an instance I, recall that $\bar{I}$ is the merge closure of I. An *entity resolution* of I is a set of records I' that satisfies the following conditions:

1. $I' \subseteq \bar{I}$,
2. $\bar{I} \preceq I'$,
3. No strict subset of I' satisfies conditions 1 and 2

The following property establishes that ER is well-defined. Proofs for this result and subsequent ones can be found in Appendixes A and B.

Proposition 2.1 *For any instance I, the entity resolution of I exists and is unique. We denote it $ER(I)$.*

Although ER is well defined, just like the merge closure it may be infinite, and therefore not computable. Even when it is finite, its computation may be very expensive. Intuitively, any finite sequence of merges may produce a different record, and dominated records can only be removed after all matches have been found. We will give in Section 3.1 an algorithm that computes ER when the merge closure is finite, which is optimal in terms of the number of record comparisons it performs. Before that, we introduce in the next section some natural properties often satisfied by the match and merge functions, which ensure the ER computation is tractable.

2.2 ICAR Match and Merge Properties

In practice, some M and μ functions have some desirable properties that lead to efficient ER. We have identified the following four such properties, which are quite intuitive.

1. *Idempotence:* $\forall r,\ r \approx r$ and $\langle r, r \rangle = r$. A record always matches itself, and merging it with itself still yields the same record.
2. *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 $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
3. *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, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.
4. *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 $r_3 \approx r_4$.

We call these the ICAR properties. We stress that not all match and merge functions will satisfy these properties, but it is nevertheless important to study the special case where they hold.

Commutativity and idempotence are fairly natural properties to expect from match and merge functions. Associativity is also a reasonable property to expect from a merge function. Note that if associativity does not hold, then it becomes harder to interpret a result record, since it not only depends of the source records, but on the order in which they were merged.

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.

Note also that we do not assume the match function to be transitive (i.e. $r_1 \approx r_2$ and $r_2 \approx r_3$ does not necessarily imply $r_1 \approx r_3$). Transitive match functions were considered by [27]. In practice, designing transitive match functions is difficult.

Merge domination When the match and merge functions satisfy the ICAR properties, there is a natural domination order that can be defined based on them, which we call the *merge domination*:

Definition 2.4 Given two records, r_1 and r_2, we say that r_1 *is merge dominated by* r_2, denoted $r_1 \leqslant r_2$, if $r_1 \approx r_2$ and $\langle r_1, r_2 \rangle = r_2$.

The properties of the match and merge functions defined in the previous section guarantee that merge domination is a valid domination partial order on records:

Proposition 2.2 *Merge domination is a valid domination order.*

Note that all the properties we required for match and merge functions are necessary to ensure that domination is a partial order relation.

The merge domination order on records is useful to understand how records relate to each other. For instance one can easily check that the following *monotonicity conditions* hold:

(A) for any records r_1, r_2 such that $r_1 \approx r_2$, it holds that $r_1 \leqslant \langle r_1, r_2 \rangle$ and $r_2 \leqslant \langle r_1, r_2 \rangle$, i.e., a merge record always dominates the records it was derived from,
(B) if $r_1 \leqslant r_2$ and $r_1 \approx r$, then $r_2 \approx r$, i.e., the match function is monotonic,
(C) if $r_1 \leqslant r_2$ and $r_1 \approx r$, then $\langle r_1, r \rangle \leqslant \langle r_2, r \rangle$, i.e., the merge function is monotonic,
(D) if $r_1 \leqslant s$, $r_2 \leqslant s$ and $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle \leqslant s$.

Interestingly, merge domination is a canonical domination order in the sense that it is the only one for which the match and merge functions "behave well", i.e., satisfy the above monotonicity conditions:

Proposition 2.3 *Given match and merge functions such that the match function is reflexive and commutative, if a domination order $\preceq$ exists such that the four monotonicity conditions (A)-(D) above are satisfied with $\leqslant$ replaced by $\preceq$, then the ICAR properties of Section 2.2 are also satisfied, and $\preceq$ coincides with the merge domination order $\leqslant$.*

In some sense, the above proposition justifies the properties we required from match and merge functions, as they capture the requirements needed to make entity resolution a monotonic process. We believe that checking our simple properties on match and merge functions is more practical than looking for an order for which the monotonicity conditions (A)-(D) are satisfied. In the rest of the paper, whenever the match and merge function satisfy the ICAR properties of Section 2.2, we consider merge domination to be our default domination order.

ER with ICAR properties When the match and merge functions satisfy the ICAR properties above, then the ER process itself has interesting computational properties: it is guaranteed to be finite, records can be matched and merged in any order, and dominated records can be discarded anytime. We next define the notion of maximal derivation sequence, and then use it to state these properties precisely.

Definition 2.5 Given an instance I, a *derivation step* $I \rightarrow I'$ is a transformation of instance I into instance I' obtained by applying one of the following two operations:

- **Merge step:** Given two records r_1 and r_2 of I s.t. $r_1 \approx r_2$, and $r_3 = \langle r_1, r_2 \rangle \notin I$, $I' = I \cup \{r_3\}$,
- **Purge step:** Given two records r_1 and r_2 of I s.t. $r_1 \leqslant r_2$, $I' = I - \{r_1\}$.

A *derivation sequence* $I \stackrel{*}{\rightarrow} I_n$ is any non-empty sequence of derivation steps $I \rightarrow I_1 \rightarrow \ldots \rightarrow I_n$. A derivation sequence $I \stackrel{*}{\rightarrow} I_n$ is *maximal* if there exists no instance I_{n+1} s.t. $I_n \rightarrow I_{n+1}$ is a valid derivation step.

The following theorem (proven in appendix) states the properties of ER:

Theorem 2.1 *Given match and merge functions that are idempotent, commutative, associative and representative, for any instance I, $ER(I)$ is finite, and any maximal derivation sequence starting from I computes $ER(I)$.*

Union Class of Match and Merge Functions There is a broad class of match and merge functions that satisfy the ICAR properties because they are based on union of values. We call this class the *Union Class*. The key idea is that each record maintains all the values seen in its base records. For example, if a record with name {John Doe} is merged with a record with name {J. Doe}, the result would have the name {John Doe, J. Doe}. Unioning values is convenient in practice since we record all the variants seen for a person's name, a hotel's name, a company's phone number, and so on. Keeping the "lineage" of our records is important in many applications, and furthermore ensures we do not miss future potential matches. Notice that the actual presentation of this merged record to the *user* does not have to be a set, but can be any string operation result on the possible values (e.g., {John Doe}). Such a strategy is perfectly fine as long as the records only use the "underlying" set values for matching and merging. Two records match if there exists a pair of values from the records that match. In our example, say the match function compares a third record with name {Johnny Doe} to the merged record obtained earlier. If the function compares names, then it would declare a match if Johnny Doe matches either one of the two names. The match and merge functions in this Union Class satisfy the ICAR properties as long as the match function is reflexive and commutative (two properties that most functions have):

Proposition 2.4 *Given match and merge functions such that the match function is reflexive and commutative, if the match and merge functions are in the Union Class, the ICAR properties are satisfied.*

Beyond the Union Class, there are other functions that while not strictly in this class, also record in some way all the values they have encountered. For example, a record may represent the range of prices that have been seen. If the record is merged with another record with a price outside the range, the range is expanded to cover the new value. Thus, the range covers all previously encountered values. Instead of checking if the prices in the records match exactly, the match function checks if price ranges overlap. It can be shown that match and merge functions that keep all values explicitly or in ranges also satisfy the ICAR properties.

In this section, we proposed four simple and natural conditions on merge and match functions for records: commutativity, idempotence, representativity, and associativity. We showed that under these conditions, records and instances can be meaningfully ordered through merge domination, and that ER is finite and independent from the order in which records are processed. We believe that the ICAR properties above are important in practice, for two main reasons:

(a) There are many applications where these properties hold. For example, in some intelligence gathering applications, values are unioned during merges, to accumulate all evidence. One can show that such "additive" applications use Union Class match and merge functions, satisfying the properties. The properties also hold if values can be combined (when two record are merged) into a "representative value" that captures all matches with values it represents.

(b) By understanding the huge performance advantages that the properties give us we believe that application designers will be strongly incentivized to develop functions that have the properties. In some cases, achieving the properties involves small changes. For example, in one application we ran across a match function that was not idempotent. However, it was easy to make the function idempotent by adding an explicit check for the case where both input records had identical content. In other cases, obtaining good functions may involve more complex changes. But without knowing what efficient algorithms exist for the case where the properties hold, the designer may never put the effort into developing good functions.

In the next two sections, we propose actual algorithms to compute ER for both the cases when the properties do not hold and when they do. The performance advantage of having the properties satisfied will be illustrated by our experiments in Section 5.

3 Record-Level ER Algorithms

We start by presenting G-Swoosh, an algorithm that does not require the match and merge functions to satisfy any particular properties. As ER may be infinite, G-Swoosh may not terminate, and in general is expensive, but we show that it is cost optimal for this very general scenario. We then present R-Swoosh, an algorithm that applies when the match and merge functions satisfy the ICAR properties, and which is also optimal for that situation.

```
1: input: a set I of records
2: output: a set I' of records, I' = ER(I)
3: I' ← I; N ← ∅
4: repeat
5:    I' ← I' ∪ N; N ← ∅
6:    for all pairs (r, r') of records in I' do
7:       if r ≈ r' then
8:          merged ← ⟨r, r'⟩
9:          if merged ∉ I' then
10:            add merged to N
11:         end if
12:      end if
13:   end for
14: until N = ∅
15: for all pairs (r, r') of records in I' where r ≠ r' do
16:   if r' ⪯ r then
17:      Remove r' from I'
18:   end if
19: end for
```

Alg. 1: The BFA algorithm for ER(I)

3.1 The G-Swoosh Algorithm

To motivate G-Swoosh, we first present a simple, naive algorithm that makes no assumptions about the match and merge functions. As defined in Section 2.1, ER(I) is the set of all non-dominated records that can be derived from the records in I, or from records derived from them. Algorithm 1 presents a "brute force" algorithm, BFA, that performs ER. The proposition that follows states the correctness of BFA. Its proof is given in Appendix.

Proposition 3.1 *For any instance I such that $\bar{I}$ is finite, BFA terminates and correctly computes $ER(I)$.*

To illustrate how BFA works, consider the instance of Figure 2. The initial instance I is represented by the records in the left column. Matching (similar) records are enclosed by a rectangle, and the arrow points to the resulting merged record. The horizontal order corresponds to the progression of the algorithm.

In the first iteration, BFA compares all possible pairs of records in the initial I, generating the new records r_{12} and r_{23}. Since new records were generated, the algorithm continues with a second iteration, in which 7 records are compared (the 5 original ones plus the two new ones). Thus, in this second iteration, the new record r_{123} is generated. Again, since a new record was found, we iterate with I' now containing 8 records generating r_{1235}. The fourth and last iteration finds no matches. Finally, BFA eliminates all dominated records and terminates.

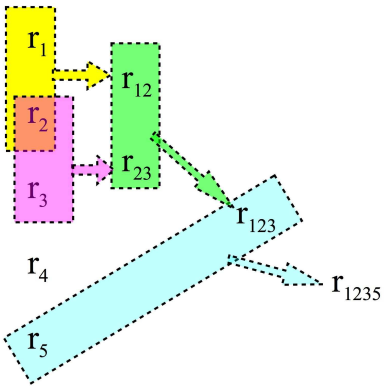

Fig. 2 "Brute force" ER on a simple instance

It is clear that BFA is performing a large number of match calls, and that many of them are unnecessary. For example, note that records r_4 and r_5 are compared a total of four times. The three redundant comparisons could be avoided by "remembering" results of previous match calls. The G-Swoosh algorithm (Algorithm 2) avoids all these unnecessary comparisons, not by explicitly remembering calls, but by intelligently ordering the match and merge calls.

```
1:  input: a set I of records
2:  output: a set I' of records, I' = ER(I)
3:  I' ← ∅
4:  while I ≠ ∅ do
5:      r ← a record from I
6:      remove r from I
7:      for all records r' in I' ∪ {r} do
8:          if r ≈ r' (resp. r' ≈ r) then
9:              merged ← ⟨r, r'⟩ (resp. ⟨r', r⟩)
10:             if merged ∉ I ∪ I' ∪ {r} then
11:                 add merged to I
12:             end if
13:         end if
14:     end for
15:     add r to I'
16: end while
17: Remove dominated records from I' (See lines 15-18 in BFA)
18: return I'
```

Alg. 2: The G-Swoosh algorithm for ER(I)

G-Swoosh works by maintaining two sets. I is the set of records that have not been compared yet, and I' is a set of records that have all been compared with each other. The algorithm iteratively takes a record r out of I, compares it to every record in I', and then adds it to I'. For each record r' that matches r, the record $\langle r, r' \rangle$ is added to I.

Returning to the example of Figure 2, r_1 is first added to I' (there is nothing yet to compare against). Next, r_2 is compared against all records in I', i.e. against r_1. This step generates r_{12}, which is placed in I. At the same time, r_2 is added to I'. Next, r_3 is compared against $I' = \{r_1, r_2\}$, adding r_{23} to I and r_3 to I'. Records r_4, r_5 and r_{12} generate no matches, so at this point we have $I = \{r_{23}\}$ and $I' = \{r_1, r_2, r_3, r_4, r_5, r_{12}\}$. When we compare r_{23} against I' we add r_{123} to I and r_{23} to I'. We continue in this fashion until I is empty and I' contains $\bar{I}$ (i.e., all the records shown in Figure 2), then dominated records are removed and the algorithm terminates. It is easy to see that in this example G-Swoosh performs many fewer comparisons than BFA.

Note incidentally that if the commutativity and idempotence properties hold, we can eliminate many comparisons in G-Swoosh (as well as in BFA). Idempotence and commutativity of the match and merge functions are easy to satisfy in most applications. If they hold, in G-Swoosh we can eliminate one of the two match calls in line 8, and one of the two merge calls in line 9. Furthermore, we do not need to match r against itself (line 7).

Proposition 3.2 *For any instance I such that $\bar{I}$ is finite, G-Swoosh terminates and computes $ER(I)$.*

Even though G-Swoosh may not terminate (because the match and merge functions are so general), it is an optimal algorithm, in terms of the number of record comparisons, our main cost metric.

Theorem 3.1 *G-Swoosh is optimal, in the sense that no algorithm that computes $ER(I)$ makes fewer comparisons in the worst case.*

Notice that in our evaluation of BFA and G-Swoosh we have used the number of calls to the match function as the main metric. We believe this metric is the right one. Each record comparison may be quite complex, taking into account several data values and using costly techniques. Moreover, the number of record comparisons is roughly quadratic in the number of records in the original instance (see Section 5). (As an aside, note that the quadratic cost is not specific of our approach; for instance, machine learning approaches, overviewed in Section 6, need to compute similarities for at least all pairs of records.) By contrast, merging records is generally less costly, as it often relies on simple syntactic rules. It is also less of a discriminating factor between algorithms, since for a given instance they will all roughly perform the same merges.

Another cost factor in G-Swoosh is the elimination of dominated records at the end. Depending on how domination is defined, this step can also be quite expensive, but is similar for both BFA and G-Swoosh algorithms.

If domination is a "black-box" partial order, then we can only eliminate dominated records after we generate the merge closure $\bar{I}$ (Definition 2.1). However, if we know how domination is checked, we may be able to perform dominated record elimination more efficiently. In particular, if we know that dominated records can never generate undominated records, then we can eliminate dominated records as soon as they are found. Note that this informal property exactly corresponds to monotonicity properties B and C of Proposition 2.2. There are actually several ways to exploit this property to improve G-Swoosh by eliminating dominated records early, but we do not discuss them here.

```
 1: input: a set I of records  /* Initialization */
 2: output: a set I' of records, I' = ER(I)
 3: I' ← ∅
 4: while I ≠ ∅ do  /* Main loop */
 5:    currentRecord ← a record from I
 6:    remove currentRecord from I
 7:    buddy ← null
 8:    for all records r' in I' do
 9:       if M(currentRecord, r') = true then
10:          buddy ← r'
11:          exitfor
12:       end if
13:    end for
14:    if buddy = null then
15:       add currentRecord to I'
16:    else
17:       r'' ←< currentRecord, buddy >
18:       remove buddy from I'
19:       add r'' to I
20:    end if
21: end while
22: return I'
```

Alg. 3: The R-Swoosh algorithm for ER(I)

3.2 The R-Swoosh Algorithm

In this section we assume that the ICAR properties defined in Section 2.2 hold. Furthermore, we assume that merge domination (Definition 2.4) is used as the definition of domination. As we argued earlier, these properties hold naturally in some applications. In other applications the match and merge properties may not initially satisfy these properties, but with small changes to the functions we may achieve the properties.

The properties simplify ER processing in two significant ways:

1. When two records r_1 and r_2 match to yield r_{12}, we are able to immediately discard the source records r_1 and r_2, since whatever records can be derived from r_1 or r_2 can now be derived from r_{12}.
2. If we eliminate records used in a merge, we do not need to explicitly eliminate dominated records. To see this fact, say we run ER without explicitly eliminating dominated records at the end. In particular, say two records r_1 and r_2 appear in the final answer, and $r_1 \leqslant r_2$. By definition of merge domination, $r_1 \approx r_2$ and $\langle r_1, r_2 \rangle = r_2$. Thus, the comparison of r_1 and r_2 should have generated merged record r_2, and r_1 should have been eliminated.

We use these two ideas in the R-Swoosh algorithm, given in Algorithm 3. To illustrate the operation of R-Swoosh, we revisit the example of Figure 2. Processing is similar to that of G-Swoosh, except when we find a match, we immediately discard both source records. In particular, when we find that r in I matches r' in I', we do not need to compare r to any other I' records: we simply remove r from I and r_2 from I' and add the new records to I. For example, after r_1 and r_2 are processed, I' is empty (in G-Swoosh it contained $\{r_1, r_2\}$) and $I = \{r_3, r_4, r_5, r_{12}\}$. When we next compare r_3 against I' we do not perform any comparisons and just add r_3 to I'. The final result is $I' = \{r_4, r_{1235}\}$. At the end, there is no need to remove dominated records.

With R-Swoosh we clearly avoid many comparisons that G-Swoosh would have performed. For instance, once r_1 is merged into r_{12}, we do not need to compare r_1 to any other records. Furthermore, we avoid generating some intermediate merged records. For example, R-Swoosh never generates r_{23}; r_3 merges directly with r_{12} to generate r_{123}.

The following proposition establishes the correctness of the R-Swoosh algorithm.

Proposition 3.3 *Given an instance I, the R-Swoosh algorithm computes $ER(I)$.*

As R-Swoosh randomly picks the next record from the set I, this leaves room for improvement. In some cases, additional knowledge can be used to influence the order in which records are picked (e.g. through sorting the records, in the style of [19]), so that the number of comparisons is reduced on average. However, if we have no knowledge of what order is best, then R-Swoosh is "optimal" in the sense that even on the most unfavorable instances, R-Swoosh performs at least as well as any other possible algorithm.

Proposition 3.4 *For any instance I of n records such that entity resolution yields j records, R-Swoosh performs at most $(n-1)^2 - \frac{(j-1)(j-2)}{2}$ record comparisons. There exists an instance (with n records, yielding j records) on which any algorithm performs at least as many record comparisons.*

4 Feature-Level ER

Although R-Swoosh is optimal in terms of record comparisons, it may still perform redundant comparisons of the underlying values. To see why, recall the example we used in the introduction, corresponding to the instance of Figure 1. The names "John D." and "John Doe" are first compared when records r_1 and r_3 are compared, and then recompared when r_4 (obtained from merging r_1 and r_2) and r_3 are compared. More generally, different records may share common values, therefore the same comparisons may be performed redundantly.

We classify value comparisons based on their outcome: *Positive comparisons* are the ones that succeed (e.g., the names "John Doe" and "J. Doe" are similar), while *negative comparisons* fail (e.g., "John D." and "John Doe" do not match in our example). Our goal is to avoid repeating both positive and negative value comparisons.

To avoid these redundant comparisons, we refine the granularity of the match function, to take into account the *contents* of records. We break down the process of comparing records into several fine-grained comparisons on features (data subsets) of the compared records. In the previous example, the name is such a feature, while the combination of e-mail and phone number forms another feature. For each feature, a specific comparison function is used. Two records match if one or more of their features match. In a nutshell,

the F-Swoosh algorithm will improve performance by taking into account these feature comparisons, and by keeping track of encountered values in order to avoid positive and negative redundant comparisons.

More formally, we consider a finite set of *features* $f_1, \ldots, f_m$. Each feature f_i is a function on records that returns a set of feature values from some domain $\mathcal{D}_{f_i}$. For example, since the second feature f_2 of our example above is "phone+email," $f_2(r_4)$ = {{234-4358, jdoe@yahoo}, {235-2635, jdoe@yahoo}}. Each feature f_i comes with a boolean match function M_{f_i} defined over $\mathcal{D}_{f_i} \times \mathcal{D}_{f_i}$. Two records r_1, r_2 match iff there exists a feature f_i and feature values v_1, v_2 s.t. $v_1 \in f_i(r_1),\ v_2 \in f_i(r_2)$ and $M_{f_i}(v_1, v_2) = \texttt{true}$.

Thus, record matching is defined as an existentially quantified disjunction over feature matches. One could think of the scenario where record matching can be done by any one of several match functions. Suppose two records match if they are similar according to either an edit distance algorithm or a fuzzy logic algorithm. In this case, the entire matching is a disjunction of the two match functions. On the other hand, if the individual match functions by themselves are not powerful enough to determine matches, one may want to consider more complex combinations of features, e.g., involving conjunction, universal quantification, or negation. However, the disjunctive case we consider here leads to simple bookkeeping, since one can determine if records match by comparing one feature at a time. We believe that with more complex conditions, bookkeeping will be significantly more complex, and more storage will be necessary, slowing down the performance of F-Swoosh. Bookkeeping becomes more complex because we can no longer store each feature separately as we do in F-Swoosh based on the assumption that a single feature match implies an entire record match. Managing several features together also requires larger data structures, which take longer to access.

Just as for R-Swoosh, we still require that the ICAR properties of Section 2.2 be satisfied. We need to make sure that the feature-level match functions M_{f_i} are such that their combination yields a record-level match function that satisfies these properties. A simple sufficient condition is to have an idempotent, commutative and associative merge function, and have each of the M_{f_i} be idempotent, commutative and representative for this merge function.

4.1 The F-Swoosh Algorithm

We now present the F-Swoosh algorithm. As its name suggests, F-Swoosh has a similar structure to that of R-Swoosh. The set I' is here also used to incrementally build a set of non-dominated, non-matching records. The main difference is that for each feature, a hash table and a set are used to keep track of previously seen feature values and save redundant positive and negative comparisons, respectively. An important point is that these data structures have a size which is only linear in the size of the data. Simply recording the outcome of all previously performed match comparisons would occupy a quadratic space, which is unacceptable for large datasets. The F-Swoosh algorithm is given in Algorithm 4. We first introduce the data structures used by F-Swoosh, before discussing the algorithm itself and its properties.

For each feature f_i, we maintain a data structure P_{f_i} that avoids repeating positive value comparisons for f_i. P_{f_i} is a hash table that records all previously seen values of f_i, and associates with each feature value v the record r [2] that currently "represents" v. The record r is either the first record where feature value v appeared for feature f_i, or one that was derived from it through a sequence of merge steps. If there is no such record, i.e., feature value v is seen for the first time, $P_{f_i}(v)$ returns null. Note that there can be at most one record associated with the feature value v for f_i; if more than one record has been seen, the records have been merged into the record returned by $P_{f_i}(v)$. The hash table is updated by a command of the form $P_{f_i}(v) \leftarrow r$. If the feature value v (for f_i) had not been recorded earlier, then this command adds the pair (v, r) to the table. If v had been seen (for f_i), then the command replaces the (v, r') pair by (v, r), indicating that the old record r' has been merged into r.

For each feature f_i, we also maintain a data structure N_{f_i} aimed at avoiding redundant negative value comparisons. N_{f_i} is a set that records the feature values of f_i that were compared against all the feature values of records in I' and did not match any of them (line 31). By representativity, this implies that if the record currently processed by the algorithm has an f_i value that appears in N_{f_i}, then this value need not be further compared (line 23).

Algorithm When a new record is processed by F-Swoosh, the algorithm first registers any new feature values (lines 12-14), then checks if any of the values of the record already appeared in a different record (lines 15-20). If this is the case, the record will be merged with the one pointed by the corresponding entry in the P_{f_i} hash table. If not, the feature values of the record are compared to those of the records in I' (lines 21-34), and if no match is found, the record is inserted in I'. As for R-Swoosh, when a match is found, the old records buddy and currentRecord are purged, while the merged record is placed in I for processing. Additionally, the P_{f_i} hash tables are updated so that feature values that previously pointed to buddy or currentRecord now point to the new merged record (lines 42-44).

As an optimization, and to avoid scanning the hash tables in this last step, one can keep an "inverted" hash table that maintains, for each record, a list of (features, feature value) pairs that point to it. This data structure costs space linear in the size of the instance, and its maintenance is straightforward. This optimization was used in the code that ran our experiments.

To illustrate the operation of F-Swoosh, suppose we have the three records r_1 = {name: John Doe, phone: 235-2635, email: jdoe@yahoo}, r_2 = {name: Fred, phone: 678-1253, email: fred@yahoo}, and r_3 = {name: John Doe, phone:

[2] In fact, we slightly abuse notation, as this is a pointer to the corresponding record, and not the record itself.

235-2635, email: jdoe@microsoft}. Suppose that there are two features "name" and "phone+email," and that two records match if their names are similar or if both their phones and emails are the same. We first add r_1 to I' and then compare r_2 to r_1. Since r_2 does not match with r_1, r_2 is also added to I'. Unlike R-Swoosh, however, r_3 is then directly merged with r_1 (without running the match function) because the feature value {John Doe} is found in P_{name}. The merged record r_{13} = {name: John Doe, phone: 235-2635, email: {jdoe@yahoo, jdoe@microsoft}} is now the current record. This time, we do not need to compare the names of r_{13} and r_2 (unlike R-Swoosh) because {John Doe} is found in N_{name} (which means that we know {John Doe} has already been compared with all the feature values of "name" in I' and thus does not match with {Fred}). After we compare the "phone+email" values of r_{13} and r_2, r_{13} is added to I'. As a result, F-Swoosh performs fewer feature value comparisons than R-Swoosh.

The correctness of F-Swoosh is established by the following proposition.

Proposition 4.1 *Given an instance I, the F-Swoosh algorithm computes the maximal derivation of I, and therefore solves the ER problem.*

F-Swoosh exercises a lot of care not to perform redundant or unnecessary feature value comparisons. The P_{f_i} hash tables records all previously seen feature values, (including those that may have disappeared from I' because of merges) and keep track of records that represent them, to immediately merge any records where these feature values may appear again (lines 15-20). Pairs of feature values that match immediately lead to a merge, and are never recompared again, while pairs of feature values that do not match (or feature values that represents them) are added to the sets N_{f_i}, and once this happens, are guaranteed to never be recompared again.

Some feature value comparisons may still be carried out multiple times by F-Swoosh. In particular, pairs of feature values that do not match may be recompared at a later time if a merge happens, and at least one of the feature values hasn't been recorded in N_{f_i}. Avoiding such redundancies would require to store the outcome of all previous unsuccessful feature value comparisons, which would have an unacceptable storage cost. Instead, our algorithm tries to minimize the windows where such redundant comparisons may occur, by constraining the order in which records are processed. Whenever a match is found, the merged record will be set as the next record to be processed (line 45), and no new record will be processed before the merged record, or one derived from it is added to I'. At this time, encountered feature values have been added to N_{f_i} and will not be recompared against each other.

The benefits of F-Swoosh are further illustrated by our experimental evaluation, presented in Section 5.

```
 1: input: a set I of records
 2: output: a set I' of records, I' = ER(I)
 3: P_f ← empty hash table, for each feature f  /* Initialization */
 4: N_f ← empty set, for each feature f
 5: I' ← ∅, currentRecord ← null
 6: while I ≠ ∅ or currentRecord ≠ null do  /* Main loop */
 7:    if currentRecord = null then
 8:       currentRecord ← a record from I
 9:       remove currentRecord from I
10:    end if
11:    buddy ← null
12:    for all (f, v) of currentRecord do  /* Keep track of any new
       values in the record */
13:       if P_f(v) = null then P_f(v) ← currentRecord
14:    end for
15:    for all (f, v) of currentRecord do  /* Was any value previ-
       ously encountered? */
16:       if P_f(v) ≠ currentRecord then
17:          buddy ← P_f(v)
18:          exitfor
19:       end if
20:    end for
21:    if buddy = null then  /* If not, look for matches */
22:       for all (f, v) of currentRecord do
23:          if v ∉ N_f  then  /* If a value never made it to N_f... */
24:             for all value v' of each r' ∈ I' do  /* ... compare it to
                the values of I' */
25:                if M_f(v, v') then
26:                   buddy ← r'
27:                   exitfor
28:                end if
29:             end for
30:             if buddy ≠ null then exitfor
31:             add v to N_f
32:          end if
33:       end for
34:    end if
35:    if buddy = null then
36:       add currentRecord to I'
37:       currentRecord ← null
38:    else
39:       r'' ←< currentRecord, buddy >
40:       remove buddy from I'  /* Update P_f's to point to the
          merged record */
41:       for all (f, v) where P_f(v) ∈ {currentRecord, buddy}
          do
42:          P_f(v) ← r''
43:       end for
44:       currentRecord ← r''
45:    end if
46: end while
47: return I'
```

Alg. 4: The F-Swoosh algorithm for ER(I)

Incremental F-Swoosh One important advantage of F-Swoosh is that it is very easy to adapt to an *incremental* scenario where new data or new features are added. For example, suppose we have performed ER on a set of records S and have obtained S' using F-Swoosh. Next, new evidence arrives, in the form of new records ΔS. We do *not* need to run F-Swoosh on $S \cup \Delta S$. Instead, we run F-Swoosh with $I' \leftarrow S'$ and $I \leftarrow \Delta S$, and initialize the hash tables P_{f_i} and the sets N_{f_i} to the state they had at the end of the original run. This setup will avoid unnecessary comparisons between S' records, which we already know have no matches

among themselves. In fact, this will avoid any comparison of feature values that was performed during the original run. Note, incidentally, that R-Swoosh can also be made incremental, in the same fashion. Since R-Swoosh does not use additional data structures, no internal state needs to be saved or restored.

An analogous setup can be used if a new feature f_{m+1} is defined after F-Swoosh ran and generated S'. We again initialize each P_{f_i} and N_{f_i} to the state they had after the original run, and add a new hash table $P_{f_{m+1}}$ and a new set $N_{f_{m+1}}$, both empty. We take $I \leftarrow S'$ (and $I' \leftarrow \emptyset$) and run F-Swoosh with the full set of features (old ones plus f_{m+1}). Because the data structures already hold the feature values for old features, no unnecessary comparisons will be performed during the new run.

5 Experiments

We implemented G-Swoosh (assuming commutativity and idempotence; see Section 3.1), R-Swoosh, and F-Swoosh as described in the paper and conducted extensive experiments on a comparison shopping dataset from Yahoo! Shopping and a hotel information dataset from Yahoo! Travel. To evaluate the performance of an algorithm, we counted the number of feature value comparisons done by the match function. As confirmed by practitioners (see some of the companies we interacted with in the Introduction), black-box match functions are very expensive and hence the number of times they are invoked is the natural metric here. We compared the performance of the three algorithms, while varying the selectivity of the match function, to cover a large spectrum of ER situations. We also compared their scalability as the size of the input dataset grows. Finally, we quantitatively investigated whether R-Swoosh and F-Swoosh could be used even when some of the properties do not hold.

Notice that in our context it does not make sense to evaluate the run-time of the match functions themselves, nor the accuracy of the results. We have only studied when and how to invoke match and merge functions, not how to build efficient functions nor how to build ones that are good at identifying records that truly correspond to the same real-world entity (see Section 6).

5.1 Experimental Setting

We ran our experiments on a comparison shopping dataset provided by Yahoo! Shopping. In this application, hundreds of thousands of records arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Because of the volume of data, records cannot be exhaustively compared to each other, and must first be partitioned into independent clusters using some semantic knowledge, e.g., by product category, a technique commonly known as "blocking." Exhaustive algorithms such as those proposed in this paper are then used to resolve similar records within a partition or bucket. In our experiments, we used a partition of size 5,000 containing records with the sub-string "iPod" in their titles; we will call these iPod-related records from now on (As mentioned in the Introduction, when we partition the data, we miss inter-partition matches).

In addition, we ran our experiments on a hotel dataset from Yahoo! Travel. This time, many records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. The hotels are located in different countries including the United States, United Kingdom, Germany, etc. Thus, a natural way to partition the hotels is by their countries. In our experiments, we used a partition of size 14,574 containing hotels in the United States (called U.S. hotels from now on).

The ER code was implemented in Java, and our experiments were run on an 1.8GHz AMD Opteron Dual Core processor with 24.5GB of RAM. Though our server had multiple processors, we did not exploit parallelism. This is a topic we are addressing in a separate paper [7].

5.2 Match and Merge Strategies

We used for the two datasets different match and merge strategies – called MM_{shop} and MM_{trav} – that satisfy all the ICAR properties of Section 2.2. MM_{shop} uses three attributes – title, price, and category – for matching and merging shopping data records while MM_{trav} uses eight – name, street address, city, state, zip, country, latitude, and longitude – for hotel records. Each strategy will be explained in detail. Since all the ICAR properties are satisfied, G-Swoosh may use the merge domination of Definition 2.4 as its domination function.

MM_{shop} The MM_{shop} strategy uses a union approach for titles and categories, and a range approach for prices. Titles are compared using the Jaro-Winkler similarity measure [3] [22], to which a threshold t from 0 to 1 is applied to get a yes/no answer. Categories are compared using exact string matches. When two records merge, the titles and categories are unioned. (Note that, since we do exact matches for categories, each record has one category.) Prices are represented by ranges of possible values and are matched when their ranges overlap. Initially, a single price x has a range that includes all the prices that match within percentage α, i.e., $[x–\alpha x, x+\alpha x]$. When two prices merge, the two ranges are merged into a new range.

MM_{shop} uses two features: F^1_{sh}, which consists of the attributes title and price, and F^2_{sh}, which consists of title, price, and category. Although the attributes of F^1_{sh} are a subset of the attributes of F^2_{sh}, using F^1_{sh} and F^2_{sh} makes sense by giving higher thresholds to F^1_{sh} for matching. The feature values of a record for a certain feature is the combination

[3] The Jaro-Winkler similarity measure returns a similarity score in the 0 to 1 range base on many factors, including the number of characters in common and the longest common substring.

of all possible values of the attributes in the feature. Two records match if at least one of their feature values match for F^1_{sh} or F^2_{sh}. Two feature values match if each of the corresponding attribute values match, as described in the previous paragraph.

MM_{shop} satisfies the ICAR properties because each record keeps all the values explicitly for titles and categories, and a range that covers all encountered prices (see our discussion in Section 2.2).

We experimented with MM_{shop} using various thresholds. We fixed the price threshold α to 0.1 (10%). Also, we always set the title threshold of F^1_{sh} halfway between the title threshold of F^2_{sh} and 1.0 to make it stricter than F^2_{sh}'s title threshold. Thus, for the experiments we report on here we only varied the title threshold of F^2_{sh}.

There are obviously many other strategies we can use on the shopping dataset. In our technical report [6], we show more experimental results comparing additional strategies. However, we think that the MM_{shop} is representative of the behavior of all those strategies.

MM_{trav} The MM_{trav} strategy uses match and merge functions in the Union Class (see Section 2.2). Each attribute of a record retains all the distinct attribute values of the base records. When comparing two records, we do pairwise comparisons between all the feature values from each record and look for an existing match. The names and street addresses are compared using the Jaro-Winkler similarity measure to which a threshold is applied. The attributes city, state, zip, country are compared using exact string matches. Finally, the latitude and longitude are compared by checking if the absolute difference of the numeric values is less than a threshold. Since the match and merge functions are inside the Union Class, MM_{trav} naturally satisfies the ICAR properties.

MM_{trav} uses three features for matching records: F^1_{tr} consists of the attributes name, street address, city, state, and country; F^2_{tr} consists of name, street address, zip, and country; and F^3_{tr} consists of name, street address, latitude, and longitude.

We also experimented with MM_{tr} using various thresholds. We used the same threshold for comparing the names and street addresses because they had similar string lengths. This threshold was used by all three features. We also fixed the latitude and longitude thresholds to 0.1 degree (which corresponds to about 11.1 km). Thus, for our experiments we only vary the name threshold of F^1_{tr}.

In summary, we tested a variety of match and merge functions, each representing different "application semantics." For each scenario we may get fewer or more matches, and the runtime may vary. As mentioned in the Introduction, our focus is on how the different scenarios perform, not on how well the match and merge functions capture application semantics.

5.3 Match Selectivity Impact

We measured the performances of G-Swoosh, R-Swoosh, and F-Swoosh using the MM_{shop} strategy by varying the selectivity of the match function on 3,000 random iPod-related records. We did not test on the entire block of 5,000 records because there were too many matching records, making G-Swoosh extremely slow. Varying the threshold of the match function affects its selectivity. In our experiments, we varied the title threshold of F^2_{sh} to capture a variety of match criteria. However, instead of plotting all graphs with the title threshold of F^2_{sh} as the horizontal axis, we plotted against the number of actual merges that occurred, i.e., the number of records in the initial set minus that in the result set. We think that the number of merged records is a more intuitive parameter, one that captures the "selectivity" of an application. The higher the number of merged records, the more selective the match function is (allowing more records to match and merge).

Figure 3 shows the number of feature value comparisons for each algorithm as the number of merges increases for the MM_{shop} scenario. The comparisons of G-Swoosh rapidly increase even if a moderate number of records merge. This is because the complexity of G-Swoosh is exponential on the number of records that match each other. R-Swoosh also increases rapidly compared to F-Swoosh because, as merged records become larger, their numbers of feature values increase linearly for both F^1_{sh} and F^2_{sh}. As a result, R-Swoosh does many feature value comparisons when comparing large records. F-Swoosh, on the other hand, saves many of these comparisons by using the P_{f_i} and N_{f_i} hash tables. Another noticeable trend is that the comparisons of R-Swoosh start to increase rapidly after a certain point where many titles start to match each other and result in a burst of new matches.

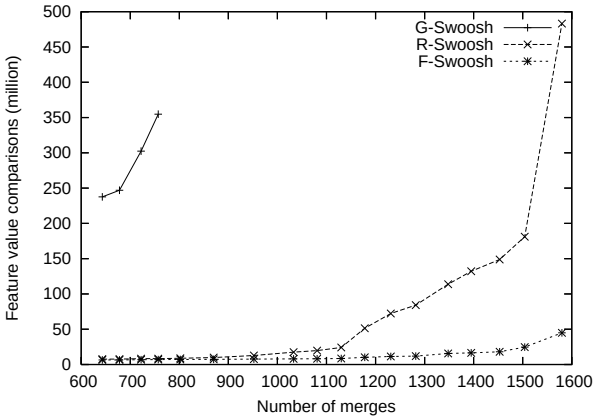

Fig. 3 MM_{shop} feature value comparisons

To illustrate how the number of feature value comparisons relates to the actual performance, Figure 4 shows the runtime for each algorithm (same MM_{sh} scenario). The runtime results mainly depend on the number of feature value comparisons. The most dominant factor of the runtime for

any algorithm in our application turns out to be the total time for matching titles during the feature value comparisons. This is because title matching involves string similarity measuring, which takes much longer than number comparing (used for matching prices) and exact string matching (used for matching categories). The total merge time is negligible because the number of merges is much less than the number of matches done.

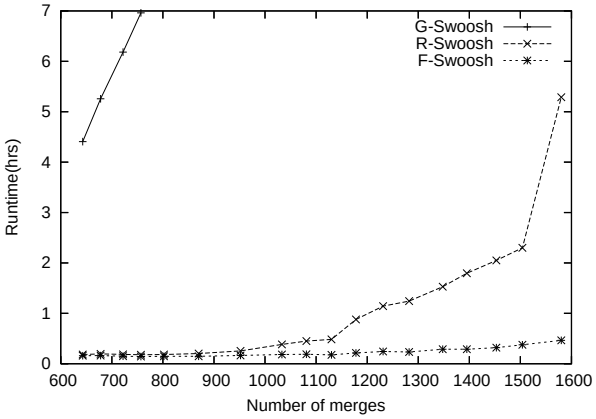

Fig. 4 MM_{shop} runtime

Next, we compared G-Swoosh, R-Swoosh, and F-Swoosh using the MM_{trav} strategy on the block of 14,574 U.S. hotel records. This time, we varied the name threshold of F_{tr}^1 to capture different selectivities of the application. One notable feature of the hotel dataset was that most of the merged records consisted of exactly two base records. The reason is twofold. First, the hotel records came from four different data sources where each data source did not contain duplicates in itself. Hence, the only way for duplicates to occur was for different sources to share the same hotels. Second, each hotel usually came from at most two different sources. Since the merged records were so small, F-Swoosh was not significantly better than R-Swoosh whereas G-Swoosh actually performed reasonably well.

Figure 5 shows the number of feature value comparisons for each algorithm as the number of merges increases for the MM_{trav} scenario. Although the differences among the algorithms are minor compared to Figure 3, the performance gap between algorithms steadily increases as the number of merges grows.

The reason is that, as we lower the comparison threshold, some merged records do get larger and have many feature values to compare.

Figure 6 compares the actual performance of the three algorithms using MM_{trav}. F-Swoosh is slightly faster than R-Swoosh while G-Swoosh has a reasonable performance and is only about twice as slow as F-Swoosh. Compared to Figure 5, there is a larger gap between G-Swoosh and the other algorithms. This additional runtime is used in the last stage of removing dominated records.

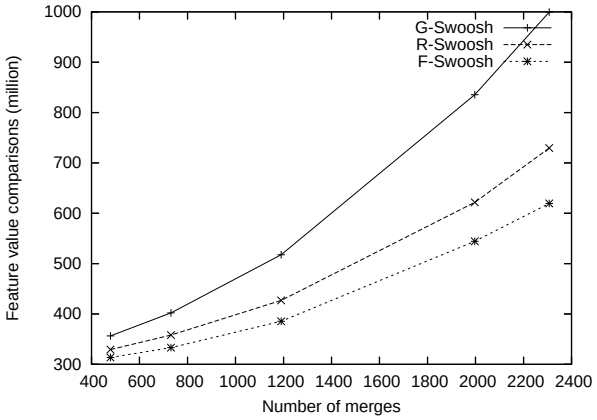

Fig. 5 MM_{trav} feature value comparisons

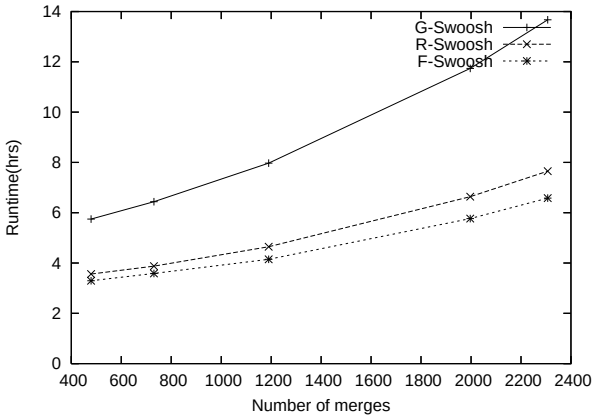

Fig. 6 MM_{trav} runtime

In summary, the actual number of comparisons and runtime depend on the "application semantics," i.e., on how selective the match function is, how merged records "grow" (e.g., by storing all the feature values of the base records), and how efficient the value comparisons are. However, in general:

- F-Swoosh is 1.1 to 11.4 times faster than R-Swoosh in the scenarios we considered.
- G-Swoosh is extremely expensive and not practical (even when we assume commutativity and idempotence) when many merges occur, but performs reasonably well when most of the merged records are very small.
- Of course, even when G-Swoosh is impractical, it is an important algorithm to understand, as it represents the cost of exact ER when the ICAR properties do not hold.

5.4 Scalability

We conducted scalability tests for G-Swoosh, R-Swoosh and F-Swoosh using MM_{shop} and MM_{trav}. We first tested MM_{shop} on randomly-selected shopping data (regardless of product types) ranging from 2,000 to 16,000 records. We then tested MM_{trav} on randomly-selected hotel data (regardless of the

countries) ranging from 2,000 to 20,000 records. The string comparison thresholds for MM_{shop} and MM_{trav} were set to values (0.9 and 0.85, respectively) that give good results.

Figure 7 shows the runtime scalability test results for each algorithm. Both R-Swoosh and F-Swoosh illustrate the quadratic cost of ER. As the dataset gets larger, F-Swoosh outperforms R-Swoosh by up to 49% (for the 16,000 shopping records). G-Swoosh increases more rapidly than the other algorithms. While G-Swoosh cannot handle more than 3,900 shopping records in a reasonable amount of time, it scales better on the hotel data.

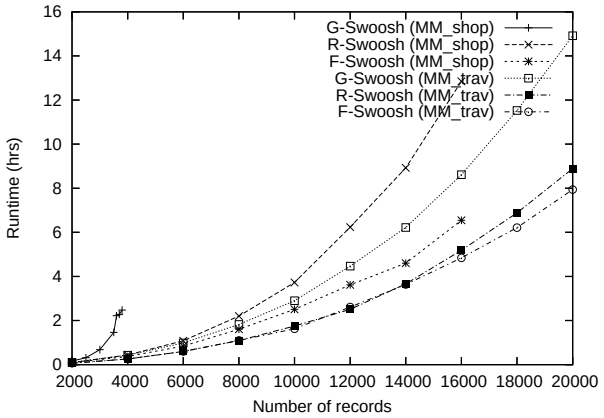

Fig. 7 Runtime scalability

In summary, ER is an inherently expensive process, so that only relatively small sets of records can be handled. Thus, large data sets need to be partitioned into smaller sets that can be resolved in detail. How large a data set can be exhaustively resolved depends on the application. It is also possible to distribute the ER computations across multiple processors, in order to handle larger data sets. In [7] we study various strategies for distributing the work done by R-Swoosh.

5.5 Without The Properties

So far, we have only studied scenarios where the ICAR properties of Section 2.2 hold. We now consider a scenario where the properties do not hold. In this case, we need to run G-Swoosh to get a correct answer. From our previous results, however, we know that G-Swoosh can be very expensive, especially when there are many large merges. The alternatives are (i) to modify the match and merge functions so that the ICAR properties hold and that they still capture reasonably what the application intends or (ii) to run R-Swoosh and F-Swoosh even though we will not get correct answers. It would be interesting to see what results we get for the second alternative.

We used a variant of MM_{shop} for our strategy. Instead of saving all the titles when merging records, we simply choose the longer string. If two strings have the same length, however, we choose the string that lexicographically precedes the other in order to satisfy commutativity. For the price attribute, we choose the numerically larger value. We did not take the average of the values because, otherwise, G-Swoosh would have produced an infinite number of new records with slightly differing prices. We used the same match function as that of MM_{shop}. In order to provide a partial order domination, we used a variation of Definition 2.4 where r_1 is dominated by r_2 if r_1's base records are included in r_2's base records. This new strategy does not satisfy the ICAR properties because the ER result now depends on the order of record merges.

Our results show that, at least for our product resolution application, the answer produced by R-Swoosh is very similar to the answer G-Swoosh generates. Since R-Swoosh is so much more efficient, it is thus attractive to use R-Swoosh even when the ICAR properties do not hold. Note that even the G-Swoosh answer may not be 100% accurate (some resolved records may not represent true real-world entities) because the match and merge functions themselves may not be not perfect. Thus, the difference between the G-Swoosh and R-Swoosh answers can also be viewed as "additional" error beyond whatever error that may occur in the G-Swoosh answer.

Before presenting our results, we give a simple example that helps interpret the results. Consider an initial instance with three records $I = \{r_1, r_2, r_3\}$. Suppose that r_1 and r_2 match, yielding $\langle r_1, r_2 \rangle = r_{12}$. Similarly, r_2 and r_3 match and $\langle r_2, r_3 \rangle = r_{23}$. However, there are no other matches. (Representativity does not hold. If it did, for instance, r_3 would match r_{12} yielding r_{123}.) In this case, G-Swoosh computes the set $I' = \{r_{12}, r_{23}\}$, assuming that r_1, r_2, r_3 are dominated. On the other hand, R-Swoosh computes $I' = \{r_{12}, r_3\}$, assuming the r_1, r_2 comparison is done first. (After r_{12} is found, r_1 and r_2 are discarded, so r_3 has nothing to match with.) If r_2 and r_3 are compared first, then R-Swoosh computes $I' = \{r_{23}, r_1\}$. Thus, we see that R-Swoosh may miss some records in the correct answer, and can generate records not in the G-Swoosh answer.

Figure 8 shows the result size comparison between G-Swoosh and R-Swoosh tested on 2,000 random iPod records varying the title threshold of F^2_{sh}. (Figure 8 shows the results for a particular record ordering, based on the order of records in the input file. Results for other orderings are similar). Again, we did not test on the entire 5,000 records became G-Swoosh was too expensive for low thresholds. Surprisingly, the result size of G-Swoosh is slightly smaller than that of R-Swoosh. This fact is because G-Swoosh considers all possible merges and tends to produce large records that dominate smaller records (like r_3 in our example) that are also part of the result of R-Swoosh.

Figure 9 shows the intersection between G-Swoosh and R-Swoosh. On average, 99.49% of all the records produced by R-Swoosh are also produced by G-Swoosh. The remaining 0.51% are those that have been discarded by domination in G-Swoosh. The result is also similar the other way: on

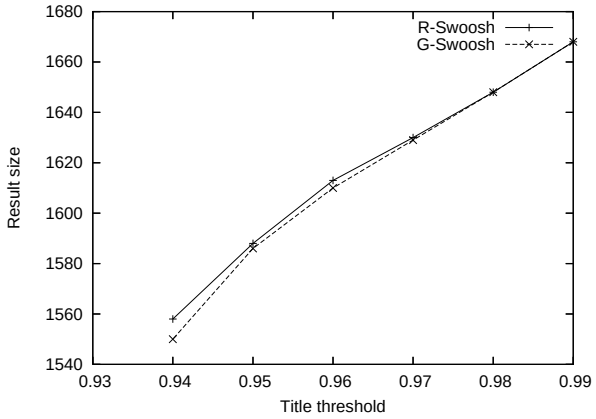

Fig. 8 Result Sizes of G-Swoosh and R-Swoosh

average, 99.64% of the records produced by G-Swoosh are also produced by R-Swoosh. This time, the missing records are the ones that R-Swoosh was not rigorous enough to come up with (like r_{23} in our example).

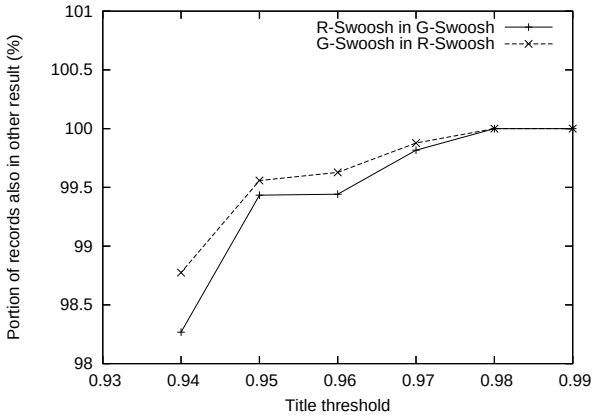

Fig. 9 Intersection between G-Swoosh and R-Swoosh

Returning to our example, G-Swoosh computes the set $\{r_{12}, r_{23}\}$ while R-Swoosh computes $\{r_{12}, r_3\}$. In this case, only 50% of the R-Swoosh records are in the G-Swoosh answer. One can argue that this low number is misleading because the "incorrect" r_3 record is closely related to the r_{23} record in the G-Swoosh set, and hence is not "completely wrong." To capture this intuition, we define a precision metric that takes into account similarity between records. For each record r in the R-Swoosh answer, we find the maximum value of the Jaccard similarity coefficients between r and each record in the G-Swoosh answer. The Jaccard similarity coefficient between two records is defined as the size of the intersection set of base records divided by the size of the union set of base records (e.g., records r_{12} and r_{23} have the Jaccard similarity coefficient of $\frac{|\{r_2\}|}{|\{r_1,r_2,r_3\}|} = \frac{1}{3}$). The precision of R-Swoosh is then the sum of the maximum Jaccard similarity coefficients for all r's divided by the size of the R-Swoosh answer. In our example, the sum of the maximum Jaccard coefficients for $\{r_{23}, r_1\}$ is $\frac{2}{2} + \frac{1}{2} = 1.5$, and the precision is $\frac{1.5}{2} = 75\%$. Figure 10 shows the precision of R-Swoosh compared to G-Swoosh. The average precision is 99.77%, showing that the R-Swoosh answer is very similar to the G-Swoosh answer.

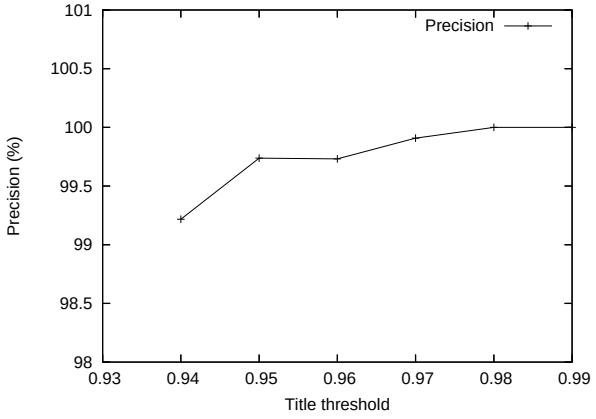

Fig. 10 Precision of R-Swoosh compared to G-Swoosh

We have also done similar experiments on the hotel data using a variant of MM_{trav} for our strategy. When merging names and street addresses, we choose the longer strings. If two strings have the same length, we choose the string that lexicographically precedes the other. For numerical attributes such as latitude and longitude, we choose the numerically larger values. We used the same match function as that of MM_{trav}. Finally, we used the same domination order as the one used in our MM_{shop} variant.

Figure 11 shows the result size comparison between G-Swoosh and R-Swoosh tested on the block of 14,574 U.S. hotel records varying the name threshold of F^1_{tr}. We show the result sizes in a table because the results of G-Swoosh and R-Swoosh are almost identical for all the thresholds we tested on, and the sizes are hard to distinguish using a graph presentation. Indeed, even in the worst case, the sizes differ only by 0.35%.

Figure 11 also shows the precision of R-Swoosh using the Jaccard similarity coefficient. The average precision is 99.88%, making the R-Swoosh result almost the same as that of G-Swoosh.

Name Threshold	G-Swoosh	R-Swoosh	Precision
0.99	14095	14095	100.0
0.95	13843	13843	100.0
0.90	13385	13385	99.996
0.85	12579	12580	99.984
0.80	11316	11324	99.890
0.75	10327	10363	99.389

Fig. 11 Result Size and Precision on the Hotel Data

The experiments show that R-Swoosh and F-Swoosh are indeed reasonable ways to do ER even if the match and

merge functions do not satisfy all the ICAR properties. They solve the scalability problem of G-Swoosh while producing most of the records G-Swoosh does.

6 Related Work

Originally introduced by Newcombe et al. [29] as "record linkage", entity resolution was then studied under various names, such as merge/purge [19], deduplication [31], reference reconciliation [14], object identification [36], and others.

Several works have addressed the issue of performance for ER algorithms. However, most of them make strong assumptions on the data and/or the comparison functions to make their algorithms efficient. For example, references [19, 20] assume that records can be represented by one or multiple alphanumeric keys, and that most matches occur between records whose keys are lexicographically close. A "blocking key" can be used to split records into buckets [22] or canopies [25]. Reference [23] proposed mapping the records values into a multi-dimensional Euclidean space, then performing a similarity join. An overview of such "blocking" methods can be found in [4]. Since they do not compare all records, such techniques make ER algorithms approximate, to an extent that depends on properties of the data. More recently, reference [2] proposed efficient algorithms for set similarity joins using string similarity functions. In contrast, we view the match and merge functions as black-boxes and provide exact ER algorithms that are optimal in the number of black-box invocations.

Reference [27] proposes a generic and exact approach, where the ER problem is viewed as an instance of the classical set-union problem [35], for which efficient data structures and algorithms were extensively studied. However, their work requires the record comparison function to be transitive, a property we believe is constraining and difficult to satisfy.

Iterative approaches [8, 14] identified the need to transitively compare merged records to discover more matches, for merges that are simple groupings of the data in merged records. Our approach allows richer, "custom" merges. More importantly, it eliminates redundant comparisons by tightly integrating merges and comparisons, and naturally yields incremental algorithms.

Our match functions are specified as logical formulas of smaller feature-level match functions, in the style of the "equational theory" of [19], and similar in spirit to works on declarative data cleaning (e.g., [16]). Such a specification has the benefits of (1) having clear semantics, (2) allowing the kinds of optimizations we perform.

While our work focuses on performance, there has also been a significant amount of work on enhancing the precision and recall of the ER process. The first formalization, by Fellegi and Sunter [15] optimizes the relative importance of numerical similarity functions between records, in a probabilistic setting. In this paper and most follow-ups (see [38, 18] for recent surveys), the assessment of ER is in terms of precision and recall of the obtained classification. Many string comparison techniques based on edit-distances [34], TF-IDF [13], or adaptive techniques such as q-grams [11, 17] are used for matching records. Reference [28] removes attribute level conflicts of matching records by comparing the quality of their data sources. Reference [32] provides user-defined grouping as part of an SQL extension. As domain-independent techniques may not be suitable for some domains, one may need domain-specific value comparison functions [1]. Any of these techniques can fill in the black-boxes, which we decouple from our match and merge process.

Finally, there has also been a great amount of research on non-pairwise ER, including clustering techniques [27, 3, 12], classifiers such as Bayesian networks [37], decision trees, SVM's, or conditional random fields [33]. The parameters of these models are learned either from a (hopefully representatitive) set of labeled example, possibly with the help of a user [31], or in an unsupervised way [39, 12]. A recent line of works focuses on the relationships among records [14, 30, 24, 5]. Reference [9] proposed a technique to resolve entities collectively based on the relationship graph among records. Such techniques are not pairwise because they generally examine all or part of the dataset to learn match decisions. In contrast, our focus is on pairwise ER because of its practical values such as easier coding and efficiency. As we mentioned in the introduction, however, we can use non-pairwise techniques during a prior training phase.

7 Conclusion

Entity Resolution (ER) is an important information integration problem arising when data from diverse sources is combined. We have divided the problem into two aspects: the black-box functions that match and merge records, and the ER algorithm that invokes these functions. In our opinion, this division has two important advantages: (i) it yields generic ER algorithms that can be used, with well-defined semantics, in many applications, and (ii) it lets us focus on our performance measure, the number of black-box invocations. While there may be other factors that affect the overall runtime performance, we assume that the black-box invocations are potentially expensive and thus are the critical runtime factors. We have presented three algorithms, G-Swoosh, R-Swoosh, and F-Swoosh, that make as few calls as possible, thus yielding significant performance improvements over naive algorithms. We have also presented four important yet natural ICAR properties for match and merge functions, that lead to the significantly more efficient R-Swoosh and F-Swoosh. We have argued that these properties should guide the development of merge and match functions. If the properties do not hold because of application semantics, the designers will know that ER will be inherently expensive.

Acknowledgements We acknowledge the useful comments made by Jeff Jonas and Tanton Gibbs.

A Proofs

A.1 Basic Model

Proposition 2.1 *For any instance I, the entity resolution of I exists and is unique. We denote it $ER(I)$.*

Proof Existence: Starting from $\bar{I}$, we can remove dominated records one at a time, until no more dominated records exist. The obtained set satisfies the conditions for being an $ER(I)$ solution.

Unicity: Suppose there are two $ER(I)$ solutions: I'_1 and I'_2. Say, some record r_1 is in I'_1 but not I'_2. There exists r_2 in I'_2 s.t. $r_1 \preceq r_2$, and r_3 in I'_1 s.t. $r_2 \preceq r_3$. Hence $r_1 \preceq r_3$, and they are distinct since $r_1 \neq r_2$. $I'_1 - \{r_1\}$, a strict subset of I'_1 satisfies conditions 1 and 2, a contradiction. □

A.2 Match and Merge Properties

Proposition 2.2 *Merge domination is a partial order on records.*

Proof We show that merge domination is reflexive, transitive and anti-symmetric:

- *Reflexivity:* $r \leqslant r$, follows from idempotence.
- *Transitivity:* Suppose $r_1 \leqslant r_2$ and $r_2 \leqslant r_3$. In particular, $\langle r_2, r_3\rangle = r_3$ and $r_1 \approx r_2$ hold, which implies, by representativity, that $r_1 \approx r_3$. This ensures the existence of $\langle r_1,\ r_3\rangle$. $\langle r_1, r_3\rangle = \langle r_1, \langle r_2, r_3\rangle\rangle$ also equals, by associativity, $\langle\langle r_1, r_2\rangle, r_3\rangle = \langle r_2, r_3\rangle = r_3$. Therefore $r_1 \leqslant r_3$ holds.
- *Anti-symmetry:* Suppose $r_1 \leqslant r_2$ and $r_2 \leqslant r_1$ hold. This means that $\langle r_1, r_2\rangle = r_2$ and $\langle r_2, r_1\rangle = r_1$. Commutativity ensures that the two left terms are equal, and therefore that $r_1 = r_2$.

□

Proposition 2.3 *Given match and merge functions such that the match function is reflexive and commutative, if a domination order $\preceq$ exists such that the four monotonicity conditions [that follow proposition 2.2] are satisfied with $\leqslant$ replaced by $\preceq$, then the ICAR properties of Section 2.2 are also satisfied, and $\preceq$ coincides with the merge domination order $\leqslant$.*

Proof We first prove that all four properties of Section 2.2 hold, and then that the orders coincide.

- *Idempotence:* The match function is reflexive, i.e., $r \approx r$ always holds. Since $r \preceq r$, by the first monotonicity property, $r \preceq \langle r, r\rangle$, and by the fourth $\langle r, r\rangle \preceq r$, and hence $r = \langle r, r\rangle$ follows.
- *Commutativity:* Suppose $r_1 \approx r_2$. The match function is commutative so $r_2 \approx r_1$. Hence both $\langle r_1, r_2\rangle$ and $\langle r_2, r_1\rangle$ are defined. By applying the fourth monotonicity property, it follows that $\langle r_1, r_2\rangle \preceq \langle r_2, r_1\rangle$ and symmetrically that $\langle r_2, r_1\rangle \preceq \langle r_1, r_2\rangle$, hence $\langle r_1, r_2\rangle = \langle r_2, r_1\rangle$.
- *Representativity:* Suppose $\langle r_1, r_2\rangle = r_3$ and $r_1 \approx r_4$. Since $r_1 \preceq r_3$, it follows from the second monotonicity property that $r_3 \approx r_4$.
- *Associativity:* Suppose $\langle\langle r_1, r_2\rangle, r_3\rangle$ and $\langle r_2, r_3\rangle$ are defined. Since $r_3 \preceq \langle r_2, r_3\rangle$, we have that $\langle\langle r_1, r_2\rangle, r_3\rangle \preceq \langle\langle r_1, r_2\rangle, \langle r_2, r_3\rangle\rangle$ by the third monotonicity property. By the same argument, $\langle r_2, r_3\rangle \preceq \langle\langle r_1, r_2\rangle, r_3\rangle$. Therefore, $\langle\langle r_1, r_2\rangle, \langle r_2, r_3\rangle\rangle \preceq \langle\langle r_1, r_2\rangle, r_3\rangle$, and since $\preceq$ is anti-symmetric, $\langle\langle r_1, r_2\rangle, \langle r_2, r_3\rangle\rangle$ is equal to $\langle\langle r_1, r_2\rangle, r_3\rangle$. By a symmetrical argument, we obtain that $\langle\langle r_1, r_2\rangle, \langle r_2, r_3\rangle\rangle$ equals $\langle r_1, \langle r_2, r_3\rangle\rangle$.

We now show that $r \leqslant s$ iff $r \preceq s$. Recall that $r \leqslant s$ means that $\langle r, s\rangle = s$. The forward implication directly follows from the property $r \preceq \langle r, s\rangle$. For the backward implication, suppose $r \preceq s$. Since $r \approx r$, it follows that $r \approx s$, and that $\langle r, s\rangle \preceq \langle s, s\rangle$, i.e, that $\langle r, s\rangle \preceq s$. We also know that $s \preceq \langle r, s\rangle$, and since $\preceq$ is anti-symmetric, we have that $\langle r, s\rangle = s$, which is the definition of $r \leqslant s$. □

A.3 ER with ICAR Properties

In order to prove the results of Theorem 2.1, we need first to define some terminology: the notions of a derivation tree, and the base and depth of a record.

Definition A.1 Given an instance I, for any record $r \in \bar{I}$, a *derivation tree* d_r of r represents a hierarchy of records in $\bar{I} - \{r\}$ that were merged to produce r. The *base* of r for derivation d_r, denoted $B(r, d_r)$, is the set of I records at the leaves of d_r . The *depth* of r for derivation d_r, denoted $D(r, d_r)$ is the size of the longest path in d_r. The depth of r, denoted $D(r)$ is its smallest depth across derivation trees of r in $\bar{I}$.

We can now show an important intermediary result:

Proposition A.1 *For any two records r, s if $B(r, d_r) \subseteq B(s, d_s)$ for some derivation trees d_r, d_s of r, s respectively, then $r \leqslant s$.*

Proof The proof is by induction on the depth of the record with the smallest base.

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

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 is dominated by its parent node and therefore, by transitivity of $\leqslant$, we have that $r \leqslant s$.

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

If $D(r, d_r) \leq k$, we can directly apply the induction hypothesis. The case to deal with is $D(r, d_r) = k + 1$. Consider the children of r in its d_r derivation tree: $r = \langle r_1, r_2\rangle$. Clearly, $D(r_1, d_{r_1}) \leq k$ and $D(r_2, d_{r_2}) \leq k$ (with d_{r_1}, d_{r_2} being the derivation trees of r_1, r_2 in d_r) . Since $B(r_1, d_{r_1}) \subseteq B(s, d_s)$ and $B(r_2, d_{r_2}) \subseteq B(s, d_s)$, we can apply the induction hypothesis to r_1 and r_2. It follows that $r_1 \leqslant s$, and $r_2 \leqslant s$. By the monotonicity properties of $\leqslant$, we can inject r_2 to obtain that $\langle r_1, r_2\rangle \leqslant \langle s, r_2\rangle$. Similarly, by injecting s we obtain that $\langle r_2, s\rangle \leqslant \langle s, s\rangle$. By merge commutativity and idempotence, and since $\leqslant$ is transitive, it follows that $r \leqslant s$. □

The finiteness of ER when the properties hold is a direct corollary of the above result:

Corollary A.1 *When the match and merge properties are satisfied, for any instance I, $\bar{I}$ (hence $ER(I)$) is finite.*

Proof A direct consequence of the previous theorem is that any two records with the same base are equal. The possible bases for records derivable from I are the elements of the powerset of I, a finite set. Therefore $\bar{I}$ is finite. Since $ER(I) \subseteq \bar{I}$, $ER(I)$ is finite as well. □

To prove the second part of Theorem 2.1, we first need to show a confluence result on derivation sequences:

Proposition A.2 ***(Confluence)*** *Given an instance I and a derivation sequence $I \xrightarrow{*} I'$, for any other derivation sequence $I \xrightarrow{*} I''$, there exists a continuation $I'' \xrightarrow{*} I'''$ such that $I' \leqslant I'''$.*

Proof (sketch) The key observation is that derivation steps are monotonic, i.e., if $I \rightarrow I'$ then $I \leqslant I'$: In the case of a merge step, clearly $I \subseteq I'$, and in the case of a purge step, the only record in I which is not in I' is dominated by another record in I (and therefore in I'). Now, consider I' and I''. The only records of I' that may not be dominated by records in I'' are those that are not in the initial instance I, and therefore were produced through merge steps. Since $I \leqslant I''$, $\forall r_i \in I$, $\exists r''_i \in I''$ s.t. $r_i \leqslant r''_i$. The idea is to continue I'' by applying the same merge steps as those that lead to I', but using instead of each r_i an $r''_i \in I''$ that dominates it (skipping the steps that do not modify the current instance). The obtained records, because merges are also monotonic, dominate the corresponding records in I', hence $I' \leqslant I'''$. □

We can now prove the second part of Theorem 2.1:

Proposition A.3 *Every derivation sequence is finite. Every maximal derivation sequence starting from an instance I computes $ER(I)$.*

Proof For finiteness, observe that for each derivation step $I \rightarrow I'$, $I \neq I'$. Since derivation sequences are monotonic (w/r to instance domination), all instances involved in a derivation sequence are distinct. Moreover, all these instances are subsets of $\bar{I}$. There is a finite number of them, since we showed $\bar{I}$ to be finite. Therefore every derivation sequence is finite.

We now construct a maximal derivation sequence that computes $ER(I)$: Starting from I, perform all necessary merge steps to produce the records in $ER(I)$. This is possible since all the needed records are in $\bar{I}$. Then, perform purge steps to remove all records which are not in $ER(I)$. Each step is a valid purge step, since $\bar{I} \leqslant ER(I)$. No additional purge step are possible, since $ER(I)$ does not contain dominated records, and no additional merge steps are possible, since $\bar{I} \leqslant ER(I)$. Therefore the derivation is maximal.

To conclude the proof, we show that all maximal derivations of an instance I compute the same result. By contradiction, suppose an instance I has two maximal derivations $I \xrightarrow{*} I_1$ and $I \xrightarrow{*} I_2$. Then, by Proposition A.2, there exists I_3 s.t. $I_1 \xrightarrow{*} I_3$ and $I_2 \leqslant I_3$. Since I_1 is maximal, it has no derivation but itself, and therefore we have that $I_3 = I_1$, hence $I_2 \leqslant I_1$. Symmetrically, we obtain that $I_1 \leqslant I_2$. Now, if $I_1 \neq I_2$, some record in one of the instances (say, $r_1 \in I_1$) is not present in the other instance (here, I_2). Since $I_1 \leqslant I_2$, r_1 is dominated by some record $r_2 \in I_2$, and we know that $r_2 \neq r_1$. Similarly, since $I_2 \leqslant I_1$, r_2 is dominated by some record r_3 in I_1, and by transitivity $r_1 \leqslant r_3$ (with $r_1 \neq r_3$). Removing r_1 from I_1 would therefore be a valid purge step, which contradicts the fact that I_1 is a maximal derivation. □

Proposition 2.4 *Given match and merge functions such that the match function is reflexive and commutative, if the match and merge functions are in the Union Class, the ICAR properties are satisfied.*

Proof Since the match and merge functions are in the Union Class, each record can be represented as a set of base records (records with single values), i.e., $r = \{b_1, b_2, ..., b_n\}$. The merge of two records $\mu(r_1, r_2) = r_1 \cup r_2$ (i.e., the set union of records). Given a match function for base records BM, the match of two records $M(r_1, r_2)$ = `true` if $\exists b_1 \in r_1, b_2 \in r_2$ s.t. $BM(b_1, b_2)$ = `true`.

We now prove that all four properties of Section 2.2 hold.

- *Idempotence:* The match function is reflexive, i.e., $r \approx r$ always holds. Since set union is idempotent, the merge function guarantees $r = \langle r, r \rangle$.
- *Commutativity:* Suppose $r_1 \approx r_2$. The match function is commutative, so $r_2 \approx r_1$. Hence, both $\langle r_1, r_2 \rangle$ and $\langle r_2, r_1 \rangle$ are defined. Since set union is commutative, the merge function guarantees $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
- *Associativity:* Suppose that for r_1, r_2, r_3, there exist $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$. Since set union is associative, the merge function guarantees $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.
- *Representativity:* Let $r_3 = \langle r_1, r_2 \rangle$. Suppose we have r_4 such that $r_1 \approx r_4$. Since we use an existential match, there exists a pair of values from r_1 and r_4 that match. Since the merge function guarantees r_3 to have all the possible values of r_1, there also exists a pair of values from r_3 and r_4 that match. Hence, according to the match function, $r_3 \approx r_4$.

□

B Record-Level Algorithms

Proposition 3.1 *For any instance I such that $\bar{I}$ is finite, BFA terminates and correctly computes $ER(I)$.*

Proof (Sketch) BFA essentially computes $\bar{I}$ by recursively adding to the set I merges of all matching records, until a fixpoint is reached, then removes dominated records. The set I increases by at least one record at each iteration, therefore BFA terminates if $\bar{I}$ is finite, and returns $ER(I)$. □

Proposition 3.2 *For any instance I such that $\bar{I}$ is finite, G-Swoosh terminates and computes $ER(I)$.*

Proof Similarly to BFA, G-Swoosh also first computes $\bar{I}$, then removes dominated records (line 17). Observe that all records added to I' are either records initially in I or records derived from them by successive merges, therefore $I' \subseteq \bar{I}$. I' increases by at least one record at each iteration, therefore G-Swoosh necessarily terminates if $\bar{I}$ is finite.

To prove that G-Swoosh computes $ER(I)$, all we need to show is that $\bar{I} \subseteq I'$ before dominated records are removed. We prove this inclusion by recursion on the depth of records in $\bar{I}$. (Recall that the depth of a record was introduced in Definition A.1.) More precisely, our induction hypothesis is that all records of $\bar{I}$ of depth less or equal than k ($k \geq 0$) appear in the working set I at some point in the algorithm. Since all records that appear in I are added to I' later on, this will establish that $\bar{I} \subseteq I'$ before dominated records are removed.

Records of depth 0 are the records of the initial dataset, which are present in I at the start of the algorithm. Suppose the hypothesis holds for all records of $\bar{I}$ of depth less or equal than k ($k \geq 0$), and let us show that the hypothesis is also verified for any record $r \in \bar{I}$ of depth $k+1$. Since $r \in \bar{I}$, r is derived either (1) from merging a record r' of depth k with itself, or (2) from merging two records r_1 and r_2 of depth less than or equal to k. For case (1), by our induction hypothesis, r' appears in I, therefore when r' is processed (line 7), r' matches itself and $r = \langle r', r' \rangle$ is added to I (line 11). For case (2), both r_1 and r_2 are of depth $\leq k$, and therefore appear in I. G-Swoosh picks these two records (line 5) in some order. W.l.o.g., say r_1 is picked before r_2. When r_1 is processed, it is added to I' (line 15), and when r_2's turn comes, a match is found and $r = \langle r_1, r_2 \rangle$ is added to I (line 11). All records in I are added to I', therefore $\bar{I} \subseteq I'$ when the algorithm reaches line 17. At that point, dominated records are removed, hence G-Swoosh computes $ER(I)$. □

Theorem 3.1 *G-Swoosh is optimal, in the sense that no algorithm that computes $ER(I)$ makes fewer comparisons in the worst case.*

Proof It is fairly immediate to see that for any pair of records r_1, r_2 (resp. for any single record r) in $\bar{I}$, G-Swoosh checks whether $r_1 \approx r_2$ (resp. whether $r \approx r$) exactly once. Suppose there exists an algorithm A that generates $ER(I)$ but performs fewer comparisons than G-Swoosh. Then for some run of A, there must exist two records r_1, r_2 in $\bar{I}$ such that A does not check whether $r_1 \approx r_2$. (The case of a single record r can be represented by taking $r_1 = r_2 = r$). Now, we construct new match and merge functions. Functions M' and μ' are the same as the original functions M and μ, unless the two records are r_1 and r_2. In this case, $M'(r_1, r_2)$ returns $true$ and $\mu'(r_1, r_2)$ returns a new record r_3 that is not dominated, and is not in $ER(I)$ using the original match and merge functions.

Using M' and μ', $r_3 \in ER(I)$. But the run of algorithm A never checks whether $r_1 \approx r_2$, so it cannot merge them to obtain r_3. Therefore, algorithm A does not generate $ER(I)$. This is a contradiction, so no algorithm that generates $ER(I)$ can perform fewer comparisons than G-Swoosh. □

Proposition 3.3 *Given an instance I, the R-Swoosh algorithm computes $ER(I)$.*

Proof Consider the instance $J = I \cup I' \cup \{currentRecord\}$. What the algorithm computes is indeed a derivation sequence of J. Whenever two records r and r' match (line 9), a merge step is performed by adding $r'' = \langle r, r' \rangle$ to I (an hence to J) (line 19). Immediately after that, two purge steps are performed: record r (removed from I at line 6) is not added to I', while records r' is removed from I' (line 19).

Therefore, r, r' are removed from J). These two purge steps are valid, since both r and r' are dominated by r''. When a record r from I has no match in I', it is added to I', leaving J unchanged.

Moreover, observe that the following is an invariant for I': for any two records $r, r' \in I', r \not\approx r'$. This is because a record is added to I' iff it does not match any record already there. The algorithm stops when I is empty. Since no two records in I' match, the derivation sequence is maximal. □

Proposition 3.4 *For any instance I of n records such that entity resolution yields j records, R-Swoosh performs at most $(n-1)^2 - \frac{(j-1)(j-2)}{2}$ record comparisons. There exists an instance (with n records, yielding j records) on which any algorithm performs at least as many record comparisons.*

Proof We first count the maximal number of record comparisons performed by Swoosh. For each record removed from I, at most $|I'|$ comparisons are conducted. When no merges occur, at each iteration $|I|$ decreases by 1, while $|I'|$ increases by 1. If the original size of $|I|$ is n, this gives us at most $\frac{n(n-1)}{2}$ comparisons. Whenever a merge occurs, $|I'|$ is decreased by 1, while $|I|$ is not decreased. Therefore one extra round of comparisons is added, but the number of records to compare to is decreased. For the first merge, the number of added comparisons is at most the maximal size of I' minus 1, i.e. $(n-2)$, and for k merges, we obtain that the maximal number of comparisons performed by R-Swoosh is $n*(n-1)/2+(n-2)+\ldots+(n-k-1)$. For R-Swoosh, it holds that $j = n - k$. Therefore, at most $(n-1)^2 - \frac{(j-1)(j-2)}{2}$ comparisons are performed.

We now prove that any exhaustive algorithm will do the same number of comparisons in the worst case. We consider a dataset consisting of n distinct records, and construct "adversarial" match and merge functions (satisfying the four properties of Section 2.2) that behave as follows: The match function returns true only after it was asked to compare all pairwise combinations of the original records, thus effectively forcing the algorithm to perform $n*(n-1)/2$ comparisons. For these two matching records, the merge function creates a new record, and the match function waits for this record to be compared against all the original records, (except the two that were merged) before declaring a match, thus forcing the algorithm to perform $n-2$ additional record comparisons. The merge function will again create a new record for the new match that was found. At the next step, $n-3$ comparisons are incurred, and so on. After k merges, $j = n - k$ records remain, and the algorithm was forced to perform at least $(n-1)^2 - \frac{(j-1)(j-2)}{2}$ record comparisons. □

C Feature-Level Algorithms

Proposition 4.1 *Given an instance I, the F-Swoosh algorithm computes the maximal derivation of I, and therefore solves the ER problem.*

Proof (sketch) The algorithm essentially computes a derivation of the instance $J = I \cup I' \cup \{\mathsf{currentRecord}\}$. It is easy to see that the derivation is correct: records are merged only if one of their feature values for one feature match. The tricky part is to show that the derivation is maximal, because the feature values of currentRecord which already appear in some N_{f_i} are not compared to the values of the records in I'. However, as we mentioned earlier, the feature values in N_{f_i} do not match each other (except when they are from the same record), and therefore if one of them appears in the current record, it cannot match another feature value of a different record in N_{f_i}. Since the feature values of the records in I' are a subset of the values in N_{f_i}, it follows that no match can be missed. Just like in R-Swoosh, records in I' are non dominated, and the set $I \cup \mathsf{currentRecord}$ is empty at the end of the algorithm, therefore the derivation is maximal. □

References

1. Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: Proc. of VLDB, pp. 586–597 (2002)
2. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)
3. Bansal, N., Blum, A., Chawla, S.: Correlation clustering. In: FOCS, pp. 238– (2002)
4. Baxter, R., Christen, P., Churches, T.: A comparison of fast blocking methods for record linkage. In: Proc. of ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation (2003). URL citeseer.ist.psu.edu/article/baxter03comparison.html
5. Bekkerman, R., McCallum, A.: Disambiguating web appearances of people in a social network. In: WWW, pp. 463–470 (2005)
6. Benjelloun, O., Garcia-Molina, H., Jonas, J., Menestrina, D., Whang, S., Su, Q., Widom, J.: Swoosh: a generic approach to entity resolution. Tech. rep., Stanford University (2006). Available at http://dbpubs.stanford.edu/pub/2005-5
7. Benjelloun, O., Garcia-Molina, H., Kawai, H., Larson, T.E., Menestrina, D., Thavisomboon, S.: D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. In: ICDCS (2007)
8. Bhattacharya, I., Getoor, L.: Iterative record linkage for cleaning and integration. In: Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (2004)
9. Bhattacharya, I., Getoor, L.: A latent dirichlet model for unsupervised entity resolution. In: 6th SIAM Conference on Data Mining (2006)
10. Blume, M.: Automatic Entity Disambiguation: Benefits to NER, Relation Extraction, Link Analysis, and Inference. In: Int. Conf. on Intelligence Analysis (2005). Available from: https://analysis.mitre.org/
11. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: Proc. of ACM SIGMOD, pp. 313–324 (2003)
12. Chaudhuri, S., Ganti, V., Motwani, R.: Robust identification of fuzzy duplicates. In: Proc. of ICDE. Tokyo, Japan (2005)
13. Cohen, W.: Data integration using similarity joins and a word-based information representation language. ACM Transactions on Information Systems **18**, 288–321 (2000)
14. Dong, X., Halevy, A.Y., Madhavan, J.: Reference reconciliation in complex information spaces. In: Proc. of ACM SIGMOD (2005)
15. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. Journal of the American Statistical Association **64**(328), 1183–1210 (1969)
16. Galhardas, H., Florescu, D., Shasha, D., Simon, E., Saita, C.A.: Declarative data cleaning: Language, model, and algorithms. In: Proc. of VLDB, pp. 371–380 (2001)
17. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
18. Gu, L., Baxter, R., Vickers, D., Rainsford, C.: Record linkage: Current practice and future directions. Tech. Rep. 03/83, CSIRO Mathematical and Information Sciences (2003)
19. Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: Proc. of ACM SIGMOD, pp. 127–138 (1995)
20. Hernández, M.A., Stolfo, S.J.: Real-world data is dirty: Data cleansing and the merge/purge problem. Data Min. Knowl. Discov. **2**(1), 9–37 (1998)
21. IBM: DB2 Entity Analytic Solutions. Http://www-306.ibm.com/software/data/db2/eas/
22. Jaro, M.A.: Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association **84**(406), 414–420 (1989)
23. Jin, L., Li, C., Mehrotra, S.: Efficient record linkage in large data sets. In: Proc. of Intl. Conf. on Database Systems for Advanced Applications, pp. 137– (2003)
24. Kalashnikov, D.V., Mehrotra, S., Chen, Z.: Exploiting relationships for domain-independent data cleaning. In: Proc. of the SIAM International Conference on Data Mining. Newport Beach, CA (2005)

25. McCallum, A.K., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets with application to reference matching. In: Proc. of KDD, pp. 169–178. Boston, MA (2000)
26. Menestrina, D., Benjelloun, O., Garcia-Molina, H.: Generic entity resolution with data confidences. In: CleanDB (2006)
27. Monge, A.E., Elkan, C.: An efficient domain-independent algorithm for detecting approximately duplicate database records. In: Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 23–29 (1997)
28. Motro, A., Anokhin, P.: Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. Information Fusion **7**(2), 176–196 (2006)
29. Newcombe, H.B., Kennedy, J.M., Axford, S.J., James, A.P.: Automatic linkage of vital records. Science **130**(3381), 954–959 (1959)
30. Parag, Domingos, P.: Multi-relational record linkage. In: Proc. of the KDD-2004 Workshop on Multi-Relational Data Mining, pp. 31–48 (2004)
31. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: Proc. of ACM SIGKDD. Edmonton, Alberta (2002)
32. Schallehn, E., Sattler, K.U., Saake, G.: Extensible and similarity-based grouping for data integratio. In: ICDE, p. 277 (2002)
33. Singla, P., Domingos, P.: Object identification with attribute-mediated dependences. In: Proc. of PKDD, pp. 297 – 308 (2005)
34. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology **147**, 195–197 (1981)
35. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
36. Tejada, S., Knoblock, C.A., Minton, S.: Learning object identification rules for information integration. Information Systems Journal **26**(8), 635–656 (2001)
37. Verykios, V.S., Moustakides, G.V., Elfeky, M.G.: A bayesian decision model for cost optimal record matching. The VLDB Journal **12**(1), 28–40 (2003). URL http://www.cs.purdue.edu/homes/mgelfeky/Papers/vldbj12(1).pdf
38. Winkler, W.: Overview of record linkage and current research directions. Tech. rep., Statistical Research Division, U.S. Bureau of the Census, Washington, DC (2006)
39. Winkler, W.E.: 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 pp. 667–671 (1988)