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

Steven Euijong Whang · Omar Benjelloun · Hector Garcia-Molina

Generic Entity Resolution with Negative Rules

Received:

Abstract Entity Resolution (also known as deduplication or merge-purge) is a process of identifying records that refer to the same real-world entity and merging them together. In practice, ER results may contain "inconsistencies," either due to mistakes by the match and merge function writers or changes in the application semantics. To remove the inconsistencies, we introduce "negative rules" that disallow inconsistencies in the ER solution (ER-N). A consistent solution is then derived based on the guidance from a domain expert. The inconsistencies can be resolved in several ways, leading to accurate solutions. We formalize ER-N, treating the match, merge, and negative rules as black boxes, which permits expressive and extensible ER-N solutions. We identify important properties for the rules that, if satisfied, enable much more efficient ER-N. We develop and evaluate two algorithms that find an ER-N solution based on guidance from the domain expert: the GNR algorithm that does not assume the properties and the ENR algorithm that exploits the properties.

Keywords Generic entity resolution, inconsistency, negative rule, data cleaning

1 Introduction

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

The process for matching and merging records is most often application-specific, complex, and error-prone. The input records may contain ambiguous and not-fully specified data, and it may be impossible to capture all the application nuances and subtleties in whatever logic is used to decide when records match and how they should be merged. Thus, the set of resolved records (after ER) may contain "errors" that would be apparent to a domain specialist. For example, we may have a customer record with an address in a country we do not do business with. Or two different company records where the expert happens to know that one company recently acquired the other, so they are now the same entity.

A common approach to handle "application errors" is to define *integrity constraints* that should be satisfied (locally and globally) by the data [18,12]. The constraints are typically written by people different from the application writers, to avoid making the same mistake twice. After (or while) the application runs, the constraints are independently checked, and inconsistencies flagged. Of course, in an ideal world, the application writers would enforce all integrity constraints perfectly, and integrity checking would be unnecessary. However, we do not live in an ideal world and integrity checking represents a useful "sanity check."

Integrity constraints tell us what data states are invalid but do not tell us how to arrive at valid state. In this paper we study how to modify the ER process, in light of some integrity constraints that we call *negative rules*, so that we arrive at a set of resolved records that satisfy the constraints. Furthermore, since in general there can be more than one valid resolved set, we also discuss how a domain expert can "guide" the ER process to arrive at a "desirable" and valid set of records using various methods for resolving records. We also explore properties of the negative rules that make this directed ER process more efficient.

For concreteness, in this paper we focus on a type of ER processing called *generic pair-wise*. In this case, a domain expert writes two functions, a match and a merge function. (Machine learning techniques could also be used to develop these functions.) The pair-wise match rule, $M(r, s)$ evaluates to true when two records r and s are determined to represent the same entity. If $M(r, s)$ is true, then a merge function is used to create the composite record $\langle r, s\rangle$. Note that after a merge we may identify new matches with other

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

S. E. Whang · H. Garcia-Molina
Stanford University Computer Science Department, Stanford, CA 94305
E-mail: {swhang, hector}@cs.stanford.edu

records. For example, the combined information in $\langle r, s\rangle$ may match with a third record t, while neither r nor s had enough information to generate a match with t.

The alternative to pair-wise ER is generally some type of global "clustering" strategy that groups records that are similar and are deemed to represent the same real-world entity [23,5]. Both pair-wise and clustering ER are used in practice [2], and each approach has its advantages (and its passionate supporters). Briefly (and open to debate), pair-wise may be easier to implement (and debug) since the domain expert only needs to consider two records at a time, and pair-wise may be more amenable to incremental and distributed processing [1]. Clustering approaches may yield more accurate results since decisions are global and have received much more attention in the academic literature.

Note that both pair-wise and clustering approaches are prone to errors and both schemes can benefit from integrity checking. Indeed, there has been prior work on clustering with constraints [3,6]. However, the integrity checking techniques used in those works are not sufficient for integrity checking in pair-wise ER. For example, our framework considers record merges (which are not directly supported by clustering approaches) and imposes constraints on merged records. A more detailed comparison between our and other work can be found in Section 8.

1.1 Motivating Example

We conclude our introduction with a motivating example. Consider the three people records shown in Figure 1, that are to be resolved. We would like to merge records that actually refer to the same person. Suppose the match rule compares r_1 and r_2 first and returns a match because they have similar names. Records r_1 and r_2 are thus merged into a new record r_{12}:

r_{12} | Pat, Patricia | 999-04-1234 | F

Now suppose that r_{12} matches with r_3 since they have similar names and an identical social security number. The result is a new record r_{123}:

r_{123} | Pat, Patricia | 999-04-1234 | M, F

In this case, r_{123} is the answer of the ER process.

However, it is easy to see there are "problems" with this solution. These problems can be identified by "negative rules," i.e., constraints that define inconsistent states. In this example, say we have a rule that states that one person cannot have two genders, and hence record r_{123} violates the constraint. The reader may of course wonder why this constraint was not enforced by the merge function that combined r_{12} with r_3. There are two reasons. One reason is that the person writing the merge rule may be unaware of this gender constraint or enforced it incorrectly. Keep in mind that the constraints in practice will be much more complex than what our simple example shows. For instance, the merge rule (or the negative rule) may be a complex computer program that considers many factors in making a decision. It may have numerous "patches" added over time by different people. Furthermore, the match and negative rules are typically written by different people (as mentioned earlier), so it is not surprising that the rules can reach conflicting decisions.

	Name	SSN	Gender
r_1	Pat	999-04-1234	
r_2	Patricia		F
r_3	Pat	999-04-1234	M

Fig. 1 A list of people

A second reason why the gender constraint was not enforced by the merge rule may be that the constraint is "fixable". In this application it may be acceptable to have a record with two genders *during* the resolution (as opposed to in the final answer), because future merges may resolve the gender. For example, say r_{123} were to merge with another record that indicated that Pat was Male. Then the merge rule may eliminate the Female gender because there is now more evidence that Pat is male. In this scenario it is OK to temporarily generate r_{123} since it is useful in constructing a valid final record. However, it is not OK to leave r_{123} in the final answer.

To resolve the gender inconsistency, say we unmerge r_{123} back into $\{r_{12}, r_3\}$. In our example, the set $\{r_{12}, r_3\}$ may still not be a valid ER answer: We may have a negative rules stating that no two final records should have the same social security number. In our case, the problem occurred because r_1 was initially merged with r_2 instead of r_3.

The reader may wonder why the ER process did not first merge r_1 and r_3 since they are "clearly" a better match than r_1, r_2. First, our example is deceivingly simple, and in practice there may be no obvious ordering to the merges. Furthermore, the person coding the match rule may not be aware of the SSN check that will be performed by the negative rule. Second, an inherent "feature" (some would say weakness) of pair-wise matching is that merge decisions are done without global analysis, a pair of records at a time. This feature is what makes the approach simple and appealing to some applications, but is also the feature that can introduce problems like the one illustrated by our example. Our approach here will be to fix these problems via the definition of negative rules.

In our simple example, we can arrive at two possible solutions that satisfy the negative rules presented above. One solution occurs when we unmerge r_{12} and re-merge r_1 and r_3, resulting in $\{r_{13}, r_2\}$. The other is when we simply discard r_3, resulting in $\{r_{12}\}$. Note that $\{r_1, r_2\}$ is not a good solution because it is not "maximal," i.e., r_1 and r_2 could have been merged without problems. The precise definition of a valid solution will be given in the next section.

Interestingly, many inconsistencies in real-world data can be captured with negative rules that examine one or two records at a time. For example, we can easily apply our rules to hotel data saying that no hotel can have two different street numbers on the same street and that no two hotels with different names can have the same street name, street number, and phone number.

In this paper we address precisely the identification and handling of inconsistent ER answers. We start by summarizing the ER model of this paper (Section 2.1), which has been introduced in our previous work [2], but does not use negative rules to handle inconsistencies. We then define the concept of negative rules (Section 2.2), both *unary negative rules* that detect internal inconsistencies within one record, and *binary negative rules* that detect problems involving a pair of records (as in our example). We formally define what is the correct ER answer in the presence of such negative rules (Section 2.3). We define simple properties of the match, merge, and negative rules that make it easier to find the correct solutions (Section 4), and we present algorithms that find a solution based on guidance from a domain expert (Sections 3, 5). We experimentally evaluate our algorithms using actual comparison shopping data from Yahoo! Shopping and hotel information data from Yahoo! Travel (Sections 6,7). We discuss related work in Section 8 and conclude in Section 9.

2 ER-N Model

2.1 ER

We start with an instance $I = \{r_1, \ldots, r_n\}$, which is a set of records.

Match and Merge Rules A match rule M determines if two records r_1 and r_2 refer to the same real-world entity. If the records match, $M(r_1, r_2)$ = `true`. We denote this as $r_1 \approx r_2$. Otherwise, $M(r_1, r_2)$ = `false` ($r_1 \not\approx r_2$).

A merge rule μ merges two records into one. The function is only defined for matching records. The result of $\mu(r_1, r_2)$ is denoted as $\langle r_1, r_2 \rangle$.

We assume two basic properties for M and μ – idempotence and commutativity. Idempotence says that any record matches itself, and merging a record with itself yields the same record. Commutativity says that, if r_1 matches r_2, then r_2 matches r_1. Additionally, the merged results of r_1 and r_2 should be identical regardless of the merge ordering.

– *Idempotence:* $\forall r,\ r \approx r$ and $\langle r, r \rangle = r$.
– *Commutativity:* $\forall r_1, r_2,\ r_1 \approx r_2$ iff $r_2 \approx r_1$, and if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.

We believe that most match and merge rules will naturally satisfy these properties. Even if they do not, they can easily be modified to satisfy the properties. To illustrate the second point, suppose that idempotence does not hold because the records have very little information (e.g., a person named John isn't necessarily identical to another person named John when no other data is available). In that case, we can be more strict in determining if two records are the same by conducting a bitwise comparison between the records or comparing the sources from which the records originated.

```
input: a set $I$ of records
output: the merge closure of $I$, $\bar{I}$
$\bar{I} \leftarrow \emptyset$
while $I \neq \emptyset$ do
    $r \leftarrow$ a record from $I$
    remove $r$ from $I$
    for all records $r'$ in $\bar{I}$ do
        if $r \approx r'$ then
            $merged \leftarrow \langle r, r' \rangle$
            if $merged \notin I \cup \bar{I} \cup \{r\}$ then
                $I \leftarrow I \cup \{merged\}$
            end if
        end if
    end for
    $\bar{I} \leftarrow \bar{I} \cup \{r\}$
end while
return $\bar{I}$
```

meaning: if the merged record looks different (not part of current workspace) then we add it to I to be compared later

Algorithm 1: Computing the merge closure ($\bar{I}$)

Merge Closure A merge closure $\bar{I}$ contains all the possible records that can be generated from I using M and μ. The formal definition is given below.

Definition 2.1 *The* merge closure *$\bar{I}$ of I satisfies the following conditions:*

1. *$I \subseteq \bar{I}$*
2. *$\forall r_1, r_2 \in \bar{I}$ s.t. $r_1 \approx r_2$, $\langle r_1, r_2 \rangle \in \bar{I}$.*
3. *No strict subset of $\bar{I}$ satisfies conditions 1,2.*

We present an algorithm for computing $\bar{I}$ in Algorithm 1. (It is shown in [2] that Algorithm 1 is optimal in a sense that no algorithm makes fewer record comparisons in the worst case.) Note that the merge closure can possibly be infinite if a chain of merges produces new records indefinitely. In Section 4, we will present some additional properties for M and μ that prevent this case.

Domination Record r_1 is dominated by r_2 if both records refer to the same entity, but r_2's information "includes" that of r_1. That is, r_1 is redundant information and should be subsumed by r_2. What records dominate others is application dependent. We can assume that for a given application there is some partial order relation (i.e., a reflexive, transitive, and anti-symmetric binary relation) that tells us when domination exists. The domination of r_1 by r_2 is denoted as $r_1 \preceq r_2$. For example, in some application where merges simply collect all information in records, we may have $r_1 \preceq r_2$ whenever $r_2 = \langle r_1, r' \rangle$ (for some r'). We will use this domination in our examples unless stated otherwise. In Section 4, we present a canonical domination order that holds when some additional properties for M and μ are satisfied.

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 as $I_1 \preceq I_2$) if $\forall r_1 \in I_1, \exists r_2 \in I_2$ s.t. $r_1 \preceq r_2$.*

2.2 Negative Rules

A negative rule is a predicate that takes an arbitrary number of records and returns either `consistent` or `inconsistent`. Negative rules can be categorized according to their numbers of arguments. In our work, we consider unary and binary negative rules.

A unary negative rule N_1 checks if a record r is valid by itself. If r is internally inconsistent, $N_1(r)$ = `inconsistent` (denoted as $r \nleftrightarrow r$). Otherwise, $N_1(r)$ = `consistent` (denoted as $r \leftrightarrow r$). An internally inconsistent record should not exist in an ER solution.

A binary negative rule N_2 checks if two different records r_1 and r_2 can coexist. We require r_1 and r_2 to be different in order to make a clean distinction between unary and binary negative rules. If r_1 and r_2 are inconsistent, $N_2(r_1, r_2)$ = `inconsistent` (denoted as $r_1 \nleftrightarrow r_2$). Otherwise, $N_2(r_1, r_2)$ = `consistent` (denoted as $r_1 \leftrightarrow r_2$). Two inconsistent records cannot coexist in an ER solution.

Neither type of negative rules can be incorporated into the match and merge rules. As we illustrated in Section 1.1, a unary negative rule cannot be supported by simply disallowing two records to merge into an internally inconsistent record because inconsistencies could be fixed in the future. Binary negative rules also do not fit in the match and merge rules for the same reason. Moreover, a match rule only has a local view of two records and cannot tell whether the merged record will generate any new binary inconsistencies with other records "outside." Thus, negative rules cannot be enforced by modifying the match and merge rules.

We say that a set of records is inconsistent if there exists a single record violating a unary negative rule or a pair of records violating a binary negative rule.

We assume the basic commutativity property for negative rules. That is, if r_1 is inconsistent with r_2, then r_2 is also inconsistent with r_1.

- *Commutativity (Negative Rule):*$\forall r_1, r_2$ s.t. $r_1 \nleftrightarrow r_2$, then $r_2 \nleftrightarrow r_1$.

Finally, the negative rules are black-box functions that can be implemented in any way as long as they satisfy commutativity.

2.3 ER-N

We now formally define entity resolution with negative rules.

Definition 2.3 *Given an instance I and the merge closure, $\bar{I}$, an ER-N of I is a consistent set of records J that satisfies the following conditions:*

1. $J \subseteq \bar{I}$,
2. $\forall r \in \bar{I} - J$, *either*
   - $\exists r' \in J$ *s.t.* $r \preceq r'$ *or*
   - $J \cup \{r\}$ *is inconsistent,*
3. *No strict subset of* J *satisfies conditions 1 and 2.*
4. *No other instances satisfying conditions 1,2, and 3 dominate* J.

Intuitively, J is a maximal consistent subset of $\bar{I}$ (The first three conditions of Definition 2.3 imply that J is consistent; the proof can be done by contradiction). The second condition ensures the maximality by saying that any record from $\bar{I}$ that is not in J is either dominated by a record in J or introduces an inconsistency to J. The third condition ensures that J is consistent and has no dominated records. Lastly, the fourth condition filters out "undesirable" solutions that are dominated by other solutions. Returning to our example in Figure 1, suppose that every pair of records match and that the merge closure $\bar{I}$ is $\{r_1, r_2, r_3, r_{12}, r_{13}, r_{23}, r_{123}\}$. The instance $\{r_{13}, r_2\}$ is a valid ER-N solution because 1) $\{r_{13}, r_2\}$ is a subset of $\bar{I}$; 2) any other record from $\bar{I}$ (i.e., r_1, r_3, r_{12}, r_{23}, r_{123}) is either dominated by a record in $\{r_{13}, r_2\}$ ($r_1 \preceq r_{13}$, $r_3 \preceq r_{13}$) or introduces an inconsistency (unary: $r_{23} \nleftrightarrow r_{23}$, $r_{123} \nleftrightarrow r_{123}$; binary: $r_{12} \nleftrightarrow r_{13}$); 3) $\{r_{13}, r_2\}$ is consistent, so no records can be dropped; and 4) $\{r_{13}, r_2\}$ is not dominated by the only other solution, $\{r_{12}\}$. The instance $\{r_{12}\}$ is also a valid solution for the same reasoning. To clarify the role of the fourth condition (i.e., the first three conditions do not imply the fourth condition), notice that the instances $\{r_1, r_2\}$ and $\{r_2, r_3\}$ satisfy the first three conditions, but are dominated by the solution $\{r_{13}, r_2\}$. Hence, $\{r_1, r_2\}$ and $\{r_2, r_3\}$ are not valid solutions.

2.4 Resolving Inconsistencies

There are two general approaches for resolving records in the presence of negative rules:

- *Late Approach.* The merge and match rules are used to generate a set $ER(I)$, which is after-the-fact checked for inconsistencies. As inconsistencies are discovered, appropriate "fixes" (see below) are taken, with the guidance of a domain expert. We call this domain expert the *solver*, to differentiate this person from that ones writing match, merge and negative rules.
- *Early Approach.* With the help of a solver, we start identifying records that we want to be in the final answer J. Even before the final answer is known, we start "fixing" problems between the selected records in J and other records not yet selected.

In this paper, we follow an early approach because the late approach involves backtracking (i.e., unmerging records), which can be very expensive. There are several ways inconsistencies can be "fixed" with the help of the solver:

- *Discard Data.* When an inconsistency is detected, the solver may decide to drop one of the records causing the problem. The dropped record will not be in the final answer.
- *Forced Merge.* The solver decides that two inconsistent records should have been merged and manually forces a merge. That is, it is deemed that the match rule made a mistake. For example, if two hotels Comfort Inn and Comfort Inn Milton are the same hotels but mistakenly not matched by the match rule, the negative rule could

flag an inconsistency (because the names are suspiciously similar), and the solver could merge them.

– *Override Negative Rule.* The solver decides that the flagged record(s) are consistent after all, i.e., the negative rule was incorrect in flagging an error. For example, Comfort Inn and Comfort Inn Milton, which were flagged by the negative rule to be suspiciously similar, might be different hotels after all. The records(s) are then allowed in the final answer.

When we present our algorithms (Sections 3 and 5), we will use a Discard technique. However, after each algorithm, we summarize the changes that are necessary to handle the other two approaches. In our experimental sections (Sections 6 and 7) we will address the accuracy and performance of the three approaches.

Note incidentally that with the Forced Merge and the Override NR approaches, we should also modify Definition 2.3 slightly, so that overridden negative rules do not count as inconsistencies, and so that forced merges are considered valid.

3 The GNR Algorithm

The GNR algorithm (General algorithm for Negative Rules) assumes the basic properties in Section 2 and that $\bar{I}$ is finite. We also assume that a solver makes decisions when there is a choice to be made. The solver looks at the records, and selects one that is "more desirable" to have in the final answer. If no solver is available, the algorithm could make the choice at random or based on heuristics (e.g., a record with more data fields is preferable to one with fewer). With human intervention, the algorithm will be guided to one of the possible solutions that is acceptable to the solver; without such guidance, the algorithm will still find a valid ER-N solution, but the solution may not be the "most desirable."

In our algorithm, the solver starts by choosing the non-dominated records from $\bar{I}$. The management of inconsistencies and domination are done by the algorithm. The algorithm is shown in Algorithm 2. The merge closure $\bar{I}$ is computed using Algorithm 1. Notice that we can automatically choose records that are non-dominated and consistent with every record in S because they will eventually be chosen by the solver.

To illustrate how the GNR algorithm works, we again refer to our motivating example in Figure 1. Again, assume that $\bar{I}$ (and thus S) is $\{r_1, r_2, r_3, r_{12}, r_{13}, r_{23}, r_{123}\}$. Since we assume that $r_i \preceq r_j$ whenever r_i was used to generate r_j, there is only one non-dominated record in $\bar{I}$, namely r_{123}. Thus, there is really no choice for the solver but to select r_{123} for the first iteration. However, r_{123} is internally inconsistent and is discarded (step 9). For the second iteration, the solver has a choice among $\{r_{12}, r_{13}, r_{23}\}$. Suppose the solver chooses r_{13}. At step 10, r_{13} is included in J. Then the records that are dominated by or inconsistent with r_{13} are removed from S, leaving S = $\{r_2, r_{23}\}$. Choosing r_2 and discarding r_{23} (since r_{23} is internally inconsistent) results in our final solution $\{r_{13}, r_2\}$. Notice that, if the solver had chosen r_{12} during the second iteration, the final solution would have been $\{r_{12}\}$.

```
 1: input: a set $I$ of records
 2: output: $J$ = ER-N($I$)
 3: $S \leftarrow \bar{I}$ /* Computed with Algorithm 1 */
 4: $J \leftarrow \emptyset$
 5: while $S \neq \emptyset$ do
 6:     $ndS \leftarrow$ the non-dominated records in $S$
 7:     $r \leftarrow$ a record from $ndS$ chosen by the solver
 8:     $S \leftarrow S \setminus \{r\}$
 9:     if $r \not\leftrightarrow r$ then continue (next iteration of loop)
10:     $J \leftarrow J \cup \{r\}$
11:     for all $r' \in S$ do
12:         if $r' \not\leftrightarrow r$ or $r' \preceq r$ then
13:             $S \leftarrow S \setminus \{r'\}$
14:         end if
15:     end for
16: end while
17: return $J$
```

Algorithm 2: The GNR Algorithm

Proposition 3.1 *The GNR algorithm returns a valid ER-N solution.*

Proof The solution J should satisfy the four conditions in Definition 2.3. First, J is a subset of $\bar{I}$ because we are not creating any new records. Second, each record r in $\bar{I}$ that is not in J was discarded (step 9) due to an internal inconsistency or deleted from S (step 13) because r was either inconsistent with or dominated by a record being inserted into J. Third, no stricter subset of J satisfies the second condition because any r removed from J is not dominated by a record in $J \setminus \{r\}$ and does not introduce an inconsistency to $J \setminus \{r\}$. Finally, no other solution dominates J: Suppose there exists such a solution J' (i.e., $J \preceq J'$). Let $[r_{s_1}, r_{s_2}, \ldots, r_{s_{|J|}}]$ be the records of J ordered by when they were added to J by Algorithm 2. Looking at r_{s_1}, we know that r_{s_1} must also exist in J' because r_{s_1} is a non-dominated record in $\bar{I}$ (ignoring internally inconsistent records), and there exists a record in J' that dominates r_{s_1} (i.e., the record that dominates r_{s_1} can only be r_{s_1}). Next, define S_1 as the records in $\bar{I}$ that are neither dominated by nor inconsistent with r_{s_1}. (Note that S_1 is what remains of the original S, after the first iteration of Algorithm 2.) According to the second condition of ER-N, no record outside S_1 can be in $J \setminus \{r_{s_1}\}$ or $J' \setminus \{r_{s_1}\}$. Now looking at r_{s_2}, we can see that r_{s_2} must also exist in J' because r_{s_2} is a non-dominated record in S_1 (ignoring internally inconsistent records) and there exists a record in $J' \setminus \{r_{s_1}\}$ that dominates r_{s_2}. After iterating through all the records of J in a similar fashion, we can see that J is a subset of J'. Moreover, J' cannot have more records than J according to the third ER-N condition. Thus, we conclude that J = J', which contradicts the assumption that the two instances are different. In conclusion, the GNR algorithm returns a valid ER-N solution.

While the GNR algorithm discards records to resolve inconsistencies (see Section 2.4), it can also use alternative strategies for resolving records. First, the algorithm can be extended to support forced merges. Once the solver chooses a record (step 7), that record is compared with every record in the set S for new inconsistencies. The solver can then view all the inconsistent pairs detected in step 12 and manually merge the records that should have been merged. After the merges, we can re-run the merge closure to identify additional matches that occur. While this step guarantees accuracy, it can be very expensive. An alternative approach is to simply continue after the forced merge without re-running the merge closure.

Second, the GNR algorithm can also support overriding of negative rules using a similar process as for forced merges. Looking at the new inconsistencies in steps 9 and 12, the solver can manually override the inconsistencies that are considered incorrect. The decisions of the solver can be stored in a hash table along with the records involved. (Thus, two records are inconsistent only if the binary inconsistency rule says they are, and the pair of records is not in the override hash table.)

Finally, the solver can use a combination of all three strategies to resolve inconsistencies. For unary inconsistencies in step 9, the solver can either discard the record or override the negative rule. For binary inconsistencies in step 12, the solver can use one of the three strategies. If the binary rule is incorrect, the solver overrides the negative rule. If the binary rule is correct but merging the two records results in an inconsistent record, the solver discards a record. However, if the merging does not introduce an inconsistency then the solver uses the forced merge technique.

Human Effort An important metric for the GNR algorithm is the "human effort" made by the solver. Of course, human effort is very hard to model and is seldom quantified in our database community. Nevertheless, because the human solver plays a key role in entity resolution with negative rules, we feel it is important to analyze human effort, even if our metric is far from perfect.

There are three ways the solver can be involved in the algorithm. First, the solver must choose records from the set ndS (step 7). Second, the solver must check whether a record is internally inconsistent during step 9 (but only if the unary negative rule returns `inconsistent`). Third, the solver must check a pair of records for inconsistencies during step 12 (only if the binary negative rule returns the result `inconsistent`). The effort made for each type of effort will vary depending on the strategy used by the solver.

Since it is difficult to predict the behavior of the solver, we use the following simple model as a surrogate of the human effort. For checking unary rules, we simply count the number of checked records. For binary rules, we count the number of pairs checked. For choosing records, the cost of selecting one record from a set of records ndS (step 7 in Algorithm 2) is $|ndS|$. The total human cost for choosing records is then the sum of costs for all such selections. For example, given a set of ten records with no inconsistencies or domination relationships, the total human effort for choosing all the ten records is $10 + 9 + ... + 2 = 54$. Notice that we do not count the effort for choosing the last record.

We caution that the human effort values we present, by themselves, are not very useful. The actual human effort will vary depending on the strategies we use for resolving inconsistencies. For example, the Discard Data and Forced Merged strategies can be run automatically and save most of the human effort while using a combination of strategies might require a significant amount of human effort (see Section 6.3). However, we believe the human effort values can be helpful in comparisons. For instance, if in Scenario A the cost is 10 times that in Scenario B, then we can infer that the solver will be significantly more loaded in Scenario A.

4 Properties for the rules

Entity resolution is an inherently expensive operation (especially with negative rules) regardless of the solution used. In general, practitioners use two types of techniques to reduce the cost: semantic partitioning and exploiting properties.

With partitioning, the initial data set is divided into independent blocks using semantic knowledge. For example, product records can be partitioned using a "category" field (book, CD, camera, ...). The assumption is that records in different blocks do not match, so an expensive algorithm like our GNR algorithm only needs to be run within each much smaller block. Of course, if some records do match across partitions, this approach will miss those matches. Similarly, inter-block inconsistencies will be missed. This partitioning technique is commonly known as "blocking" [20,21]. If the resulting blocks are relatively small, the GNR algorithm will be feasible. Also, note that the GNR algorithm becomes more attractive in scenarios where there are relatively few matches. (The more matches, the larger $\bar{I}$ becomes.)

A second general approach to reducing cost is to exploit properties of the match and merge rules to make it possible to find the correct solution with less effort. In this section we present such desirable properties: two for match and merge rules, and one for negative rules. In Section 5 we then use these properties to make the ER process significantly more efficient.

Of course, note that the properties we propose will not hold in all applications. If the properties do hold, then one will be able to achieve the improved performance. If the properties do not naturally hold, the solver may want to modify the rule so that the properties hold (e.g., by keeping more information in a merged record, one may be able to achieve the representativity property defined below). Finally, if the properties below definitely do not hold in a given application, the solver may nevertheless still want to use the efficient algorithm of Section 5, in order to get an answer in a reasonable time. The answer will *not* be correct because of the "wrong" algorithm we used for this case, but the answer may be "relatively close" to the correct answer.

The bound of incorrectness depends on the portion of "problematic" records that do not make the rules satisfy the properties. For example, the initial set of records I could conceptually be divided into two sets $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$ where the properties are satisfied when resolving the records in X while not necessarily so when resolving the records in Y. That is, Y includes all the records that could possibly generate inconsistencies. We suspect that the "incorrectness" of the ER solution would then be bounded by the fraction $\frac{Y}{X+Y}$ of the total records in the ER solution. In practice, the problematic record set Y is only a small fraction of the entire set of records (see Section 7.3). Further research is required to refine this intuitive "incorrectness" bound.

4.1 Match and Merge Rules

Two desirable properties for M and μ are associativity and representativity. Associativity says that the merge order is irrelevant. Representativity says that a merged record represents its base records and matches with all records that match with the base records.

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

Associativity and representativity together are somewhat strict, but powerful properties. Combined with the two basic properties, idempotence and commutativity, they are called the ICAR properties. It is shown in [2] that, given the ICAR properties, the merge closure of I is always finite.

Union Class of Match and Merge Rules There is a broad class of match and merge rules 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)

Beyond the Union Class, there are other rules 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.

Merge Domination If the ICAR properties are satisfied, we can use a natural domination order called merge domination.

Definition 4.1 r_1 is merge dominated by r_2 *(denoted* $r_1 \leqslant r_2$*), if* $r_1 \approx r_2$ *and* $\langle r_1, r_2\rangle = r_2$.

Reference [2] shows that merge domination is a partial order on records given the ICAR properties. Merge domination is a natural way of ordering records and will be our default domination order when the ICAR properties hold.

4.2 Negative Rules

One desirable property for negative rules is called persistence. In many applications, inconsistencies tend to hold regardless of future merges. Persistence is defined for both unary and binary negative rules.

Unary persistence is defined on unary negative rules. The property states that an internally inconsistent record r stays inconsistent regardless of its merging with other records.

Binary persistence is defined on binary negative rules. This time, two inconsistent records r_1 and r_2 stay inconsistent regardless of their merging with other records. The only exception is when r_1 and r_2 merge together, either directly or indirectly. In that case, the binary inconsistency is resolved because the two records no longer coexist ($\langle r_1, r_2\rangle$ could be internally inconsistent).

- *Unary Persistence:* If $r_1 \nleftrightarrow r_1$ and $r_3 = \langle r_1, r_2\rangle$, then $r_3 \nleftrightarrow r_3$.
- *Binary Persistence:* If $r_1 \nleftrightarrow r_2$ and $\langle r_1, r_3\rangle \neq r_2$, then $\langle r_1, r_3\rangle \nleftrightarrow r_2$.

We believe persistence holds in many applications. Unary persistence mostly holds if the merge rule is in the Union Class. For example, a hotel having two addresses will still have at least two addresses after merging with other records. Binary persistence is also reasonable – two hotels having the same address will still have the same address regardless of their merging with other hotels.

5 The ENR Algorithm

The ENR algorithm (Enhanced algorithm for Negative Rules; shown in Algorithm 3) exploits the properties in Section 4

```
input: a set $I$ of records
output: $J$ = ER-N($I$)
$P \leftarrow$ ER($I$)  /* e.g., using R-Swoosh */
$C \leftarrow$ the set of connected components of inconsistent packages in $P$
for all $c_i \in C$ do
    $J_i \leftarrow$ GNR($\bigcup_{p \in c_i} b(p)$)
end for
$J = J_1 \cup ... \cup J_{|C|}$
return $J$
```

Algorithm 3: The ENR Algorithm

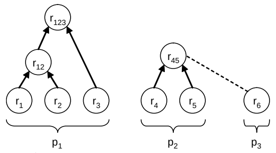

Fig. 2 Package formation

(i.e., the ICAR and Persistence properties) to make the GNR algorithm efficient. Rather than looking at the entire merge closure of I, we would like to partition I and look at the merge closure of each partition. Note that the partitions here are different from the components produced by blocking techniques (see Section 4). Specifically, we do not assume any semantic knowledge, as exploited by blocking techniques. The partitioning can be done in two steps. First, we partition I into "packages" (introduced in [22] in another context) where two records generated from different packages do not match. Next, we deal with inconsistencies by connecting "inconsistent packages" into connected components [26] so that two records generated from different components are always consistent with each other.

Packages partition I such that no two records generated from different packages match. (The two generated records may be inconsistent.) The (base) records of package p are denoted as $b(p)$, and the entire set of generated records (i.e., the merge closure) of p is denoted as $c(p)$. All the records in p can merge into a single representing record, which we denote as $r(p)$.

Packages are generated by running Algorithm 1, except that when we merge two records r and r' (step 9), we remove r and r' from further consideration. (Because of the ICAR properties any future record that would have matched r or r' will now match the merged record.) Furthermore, we do not explicitly remove dominated records at the end; the above optimization takes care of that. These two optimizations (plus a few other improvements) yield what is called the R-Swoosh Algorithm, which is studied in detail in [2]. From our point of view, the important point is that packages can be computed efficiently, given the ICAR properties and an algorithm like R-Swoosh.

Figure 2 illustrates the package formation step (ignore the dotted line for now). The bottom records are the input records, and the arrows show the merges that occur. In this example, three packages result. For instance, the leftmost package has record r_{123} as representative.

We next connect inconsistent packages together, forming connected components of inconsistent packages. We say two packages p and p' are inconsistent if their representing records, $r(p)$ and $r(p')$, are inconsistent. In our example in Figure 2, packages p_2 and p_3 are inconsistent because r_{45} and r_6 are inconsistent (dotted line). As a result, package p_1 forms one component by itself while packages p_2 and p_3 together form another component. To give an illustration why p_2 and p_3 should be connected although r_{45} and r_6 do not match, it could be the case that the name of the same hotel was written in different languages for r_{45} and r_6. While the match rule might have considered the two records different because of the different names, the negative rule could help fix that error by connecting p_2 and p_3. Proposition 5.1 shows that no two records generated from two consistent packages are inconsistent. Thus, there are no inconsistencies between records generated from different components.

Proposition 5.1 *Consider two consistent packages p, p', i.e., $r(p) \leftrightarrow r(p')$. Then $\forall r_1 \in c(p)$, $r_2 \in c(p')$, $r_1 \leftrightarrow r_2$.*

Proof Suppose that $r_1 \not\leftrightarrow r_2$. By the definition of packages, r_1 and r_2 can each merge with other records into $r(p)$ and $r(p')$, respectively. Then according to binary persistence, $r(p) \not\leftrightarrow r(p')$, which is a contradiction.

Finally, we run the GNR algorithm on the records of each connected component of packages. Returning to our example in Figure 2, the first component contains the package p_1. Thus, we run the GNR algorithm on $b(p_1) = \{r_1, r_2, r_3\}$. Notice that the solver only has to look at the merge closure of three records instead of the original six. Next, we run the GNR algorithm on the records of the second component containing package p_2 and p_3. In this case, we start with the set $b(p_2) \cup b(p_3) = \{r_4, r_5, r_6\}$. Combining the results of running the GNR algorithm on the two components gives us the final ER-N solution.

Proposition 5.2 *The ENR algorithm returns a valid ER-N solution.*

Proof It is sufficient to prove that running the ENR algorithm on I is equivalent to running the GNR algorithm on I. Adding all the merge closures of the partitions of I produced by the ENR algorithm results in $\bar{I}$ because records generated from different components are independent, i.e., they are consistent with each other and never match. Thus, the solver is looking at the same $\bar{I}$ for both algorithms. In the ENR algorithm, however, the solver is handling one subset of $\bar{I}$ at a time.

While the ENR algorithm assumes the Discard approach, it can also support alternative strategies for resolving records. Since the ENR algorithm only plays a role in isolating inconsistencies, the actual algorithmic changes are all done on the GNR algorithm. Hence, the ENR algorithm does not change regardless of the strategy used.

6 Precision and Recall

To evaluate our GNR and ENR algorithms, there are two sets of issues to consider: accuracy and performance. In this section we consider accuracy, i.e., how and by how much can precision and recall of a solution be improved by using negative rules and our algorithms. In the following section we address the performance, i.e., the human effort and system runtime needed for resolving records with negative rules.

6.1 Experimental Setting

We ran our experiments on a hotel dataset provided by Yahoo! Travel. In this application, hundreds of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. Because of the volume of data, we used blocking techniques (see Section 4) to partition the data into independent blocks and then applied our algorithms on each block. In our experiments, we used a partition containing hotels in the United States; we will call these U.S. hotels from now on.

To evaluate accuracy, we used a "Gold Standard" G also provided by Yahoo. Gold standard G is a set of record pairs. If a pair (A, B) is in G, then input records A and B are considered by a domain expert to be the same hotel. If a pair A, B is not in G, then A and B represent different hotels. Set G turns out to be transitive, i.e., if (A, B) and (B, C) are in G, then (A, C) is also in G.

To evaluate an ER-N solution we proceed as follows. We consider all the input records that merged into an output record to be identical to each other. For instance, if hotels A and B merged into $\langle A, B\rangle$ and then merged with C, all three hotels are considered to be the same. Let S be the set of all pairs found to be equal. In our example, (A, B), (B, C) and (A, C) are all in S. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. In addition, we also used the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, as a single metric for precision and recall.

The GNR and ENR algorithms were implemented in Java, and our experiments were run on a 2.0GHz Intel Xeon processor with 6GB of memory. Though our server had multiple processors, we did not exploit parallelism.

6.2 Rules

Since we did not have access to the proprietary code in Yahoo's match and merge rules, we developed our own rules, based on our understanding of how hotel records are handled. Our rules are union rules, as described in Section 4.1. That is, our merge rule μ retains all the distinct values of the base records.

The match rule M compares two hotel records using eight attributes: name, street address, city, state, zip, country, latitude, and longitude. When comparing two records, we do pairwise comparisons between all the possible attribute values from each record and look for a match. The names and street addresses are first compared using the Jaro-Winkler similarity measure [1] [20], to which a threshold T_M from 0 to 1 is applied to get a yes/no answer. We use the same threshold T_M for comparing names and addresses because they have similar string lengths. If the names and street addresses match, M returns `true` if at least one of the following holds:

- the cities, states, and countries are exactly the same.
- the zip codes and countries are exactly the same.
- the latitude and longitude values do not differ more than 0.1 degree (which corresponds to approximately 11.1km).

It is easy to show that the union operation for merging and existential comparison for matching guarantee the ICAR properties.

For our experiments we used two types of negative rules. Here we describe the first type, and the second type is discussed later on in this section. Our initial negative rules are based on the phone number attribute. This attribute is not as robust as say hotel name or city and zip code for determining matches, but is useful for detecting anomalies that should be checked by the solver.

In particular, our initial unary negative rule N_1 flags a hotel with different phone numbers. In order to precisely compare phone numbers, we first remove non-numeric characters (e.g., '(','(', and '-'). We then compare each digit starting from the last position until we have compared all the digits of either one of the phone numbers. We compare from the last digit because some phone numbers include area codes while others do not. For example, we consider "(650)123-4567" and "1234567" to be equal by trimming the first phone number into "6501234567" and then comparing the last seven digits. This strategy works very well in our dataset.

Our initial binary negative rule N_2 checks if two hotels have the same phone number. That is, N_2 does a pairwise phone number comparison between all the possible phones of the two records, looking for existing matches. N_2 uses the same phone number comparison function as N_1.

6.3 Strategies

We first resolved the records without using the negative rules, using only M and μ (i.e., just step 3 of the ENR Algorithm). We used as input 5,000 U.S. hotel records, and we used various thresholds for T_M. The solid line in Figure 3 shows the precision and recall curve for each threshold we used (ignore the other data points for now). Among them, the threshold that produces the highest F_1-measure is 0.74, and the point using that threshold is marked as the "Best Point." To give an idea on how many records actually merged together in the Best Point result, we show in Figure 4 the distribution of

[1] 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.

base records per output record. While most input records did not merge with any other record, a significant portion of the output records were formed by a merge of two input records.

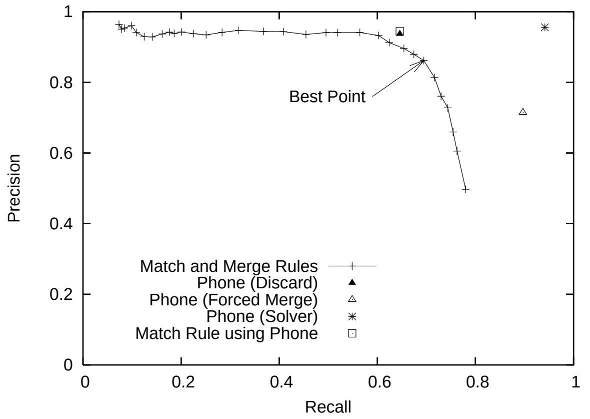

Fig. 3 Precision and recall for different strategies

Record size	Number of records
1	3477
2	725
3	21
4	0
5	2

Fig. 4 Distribution of base records per output record

Discard Strategy Next we ran the ENR Algorithm, using the negative rules and the threshold ($T_M = 0.74$) for the match rule that yielded the Best Point. Recall that with the Discard strategy, the solver only has to select records for the final result. Any negative rule violations are simply corrected by removing records. When choosing records during step 7 in Algorithm 2, we emulated the solver's decisions by always selecting the record containing the largest number of base records from the set ndS.

The resulting precision and recall of the Discard strategy is shown in Figure 3. (Note that the dark triangle for the Discard strategy overlaps with the square for a scheme that is described below. Both schemes have approximately the same performance.) Compared to the Best Point, the precision has increased while the recall has decreased. Intuitively, discarding records reduces incorrectly merged records (increasing the precision of merging), but may also mistakenly remove correct merges (decreasing the recall).

The advantage of the Discard strategy is that the human effort is relatively small compared to the Solver strategy (see below) because the human solver only needs to choose records and does not need to do any manual unary or binary checks. As a matter of fact, if records are selected based on size (as we did for our emulation), then the solver does no actual work.

Automatic Forced Merge Strategy An alternative to fixing inconsistencies by discarding records is to force the merge of records that violate the binary negative rule. When we run ENR in this fashion (everything else unchanged) we get the Forced Merge data point in Figure 3. We see that the Forced Merge strategy decreases the precision while increasing the recall of the Best Point. Forcing inconsistent records to merge may create internally inconsistent records (decreasing the precision) and also find correct matches (increasing the recall).

The Forced Merge strategy is effective when there are many record matches that were not identified by the match and merge rules. Since we are merging inconsistent records *automatically*, without solver intervention, the solver cost is the same as for the Discard strategy.

Solver Strategy Finally, we tested a strategy where all negative rule violations are examined by the human solver, and he decides in each case whether it is best to force a merge, ignore the negative rule firing, or to discard a record. To emulate what a solver would do, we rely on the Gold Standard G. When a unary inconsistency is detected in record r (step 9 of GNR), we check if any pair of base records for r is *not* in G. If all pairs are in G, then we ignore the negative rule. When a binary rule violation is detected (step 12), we check if the records can be safely merged. If the merged record would only contain base record pairs in G, then we go ahead and force a merge. Using G to drive the algorithm is fair since we expect the human solver to make decisions that are consistent with those made by the domain expert who created the gold standard.

The accuracy of the solver strategy is shown in Figure 3. We can see that the Solver strategy significantly outperforms any strategy both in precision and recall. However, note that the solution is still not 100% correct. The reason is that the solver can only fix problems flagged by the negative rules. If an incorrect merge or a missing merge is not detected by the negative rules, then the problem is not brought to the solver's attention. Of course, the Solver strategy is more expensive for the solver, as he has to manually examine and resolve all records flagged by the negative rules. Thus, it is important to design negative rules that do not generate too many unnecessary checks.

At this point the reader may wonder, if checking phone numbers is so effective in detecting problems, why were phone numbers not checked by the match rule? As we argued in the introduction, negative rules are integrity checks often developed after the match and merge rules are implemented. It is often safer not to embed integrity checks in the same code that is being checked. Furthermore, the match and merge rules may be legacy code that is hard to modify, developed by programmers that did not have perfect knowledge.

It is also important to notice that adding phone number checks to our original match function *does not* give the same results as the Solver Strategy. For example, we could modify our match rule so that hotels with different phone numbers do not match (effectively incorporating our unary negative rule into the match function). Figure 3 (point labeled "Match Rule using Phone") shows the result of using the new match rule with the Best Point threshold ($T_M = 0.74$). Compared to the Best Point, the precision increased to 0.944 while the recall dropped to 0.646. (Incidentally, the result is very similar to that of the Discard strategy.) Hence, accuracy is much better with the solver where hotels with questionable phone numbers are being examined by an expert, so opposed to simply not merged.

6.4 Other Negative Rules

To better understand how negative rules impact accuracy, we implemented a second type of rule. These rules, sometimes used in practice, flag "borderline cases" as suspicious so the solver checks them out. For instance, say two hotels r and s have very similar names and addresses, but not quite similar enough that the match function fires (or perhaps other attributes indicate a mismatch). Then we may want the solver to look at r and s to decide what to do.

In particular, our binary negative rule states two records are inconsistent if there exists a pair of names, one in each record, that have a string similarity over T_B, and a pair of street addresses also have a similarity over T_B. Our unary negative rule checks if the possible names and street addresses in a record are "too far apart" to be in the same record. Specifically, given a unary threshold T_U, a record is internally inconsistent if two possible names differ more than T_U and two possible street addresses differ more than T_U. We used the Jaro-Winkler similarity measure for all string comparisons. We call these rules the NameAddr negative rules.

Figure 5 shows the result of testing the NameAddr negative rules on several T_U and T_B thresholds using the Solver strategy. The nine points (black squares) are produced by assigning T_U the values 0.8, 0.9, and 0.99 while assigning T_B the values 0.75, 0.7, and 0.65. As T_U increases, precision increases, and as T_B decreases, recall increases. For example, the leftmost 3 points correspond to $T_B = 0.75$, while the bottom 3 points correspond to $T_U = 0.8$. The top, rightmost point is for T_U = 0.99 and T_B = 0.65. These trends are as one would expect: the unary rule detects more inconsistencies as T_U increases while the binary negative rule does so as T_B decreases. And the more inconsistencies that are flagged, the more opportunities the solver has to fix things (and of course, the more work for the solver).

Because our negative rules have parameters that let us vary how stringent they are, we can visualize the tradeoff between accuracy and solver cost. For example, Figure 6 shows how recall and cost relate. The horizontal axis is the recall achieved as we vary T_B (keeping T_U at its lowest value), and the vertical axis shows the solver cost. For example, the rightmost points are obtained with $T_B = 0.65$ and we get a recall of about 0.806. The top most curve shows our estimate for the selection cost; the middle curve shows the number of pairs of records manually checked by the solver for binary inconsistencies, and the bottom curve shows the number of records checked for unary inconsistencies. We can clearly see that achieving the higher recall comes at a price, as the solver needs to examine more records. Note that the record selection cost can be eliminated if we automate the record selection process, choosing the largest record as we did for our experiments here. The analogous precision-cost graph (not presented here) shows that, unlike recall, achieving a higher precision does not significantly increase the solver cost.

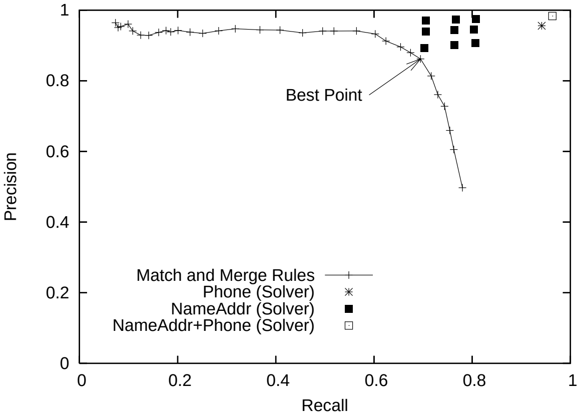

Fig. 5 Precision and recall for various negative rules

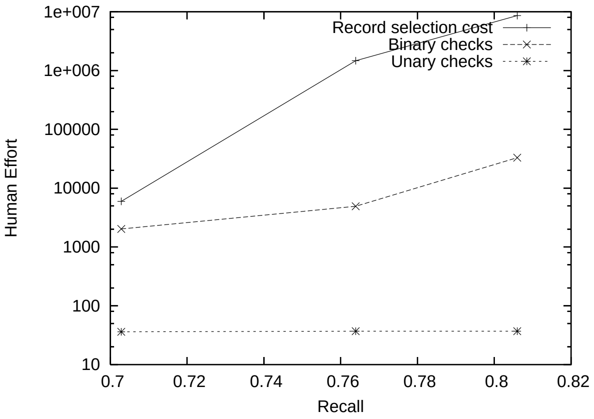

Fig. 6 Human effort versus recall

We also combined the NameAddr negative rules with the phone number negative rules presented earlier. The combined unary (binary) negative rule returns inconsistent when either one of the two unary (binary) negative rules returns inconsistent. (In this case we set T_U and T_B to 0.99 and 0.65, respectively.) Figure 5 shows that the combined method gives the highest precision and recall. Intuitively, the combined rules identify the largest number of inconsistencies for the solver to check. However, as a result, the solver does the most work.

In summary, the precision and recall of an ER-N solution depends on the strategy used for resolving records as well as the negative rules. However, in general:

- The Discard strategy increases precision, but decreases recall.
- The Forced Merge strategy increases recall, but decreases precision.
- The Solver strategy lets a human decide how to fix inconsistencies on a case by case basis. The accuracy improvement depends on how effectively the negative rules find actual inconsistencies.

6.5 Choosing Negative Rules

In general, choosing the right number of negative rules that maximize the precision and recall with a reasonable solver cost requires application knowledge about "common errors" of the match and merge rules. If the negative rules do not properly point out the errors, then the solver might end up checking unnecessary records without improving the precision or recall much.

Figure 7 shows several negative rules including the Phone and NameAddr negative rules defined in the previous sections. Figure 8 shows the precision, recall, and human effort results for various combinations of the negative rules. (We experimented on the 5,000 U.S. hotel records using the Solver strategy.) Adding the City negative rule to the NameAddress negative rules slightly increases the recall with a small additional human effort. The Latitude negative rule, on the other hand, significantly increases the precision, but also requires a much larger human effort because many different hotels can be within 1.1km (latitude) of each other. The Phone negative rule alone already gives a better precision and recall (while requiring a much smaller human effort) compared to previous combinations because it effectively pinpoints the errors of the match and merge rules.

Negative Rule(s)	Description
C	No hotel can have two different cities
L	No two hotels can have latitudes that differ less than 0.01 degree (i.e., 1.1km)
NA	The NameAddr negative rules defined in Section 6.4 where T_U=0.8 and T_B=0.75
P	The Phone negative rules defined in Section 6.2

Fig. 7 List of negative rules

Combination	Precision/Recall	Human Effort
NA	0.7/0.89	5968
NA + C	0.7/0.91	6017
NA + L	0.87/0.91	4692245
NA + C + L	0.87/0.93	4722335
P	0.94/0.95	934

Fig. 8 Results for various combinations of negative rules

The experiments show that finding the best negative rules requires a good understanding of the application and the match and merge rules. Carefully thought-out negative rules (like the Phone negative rules) will be able to find most of the real inconsistencies of the match and merge rules with little human effort. Other negative rules may either fail to find many inconsistencies or end up increasing the human effort too much.

7 Performance

In this section, we address the performances of the GNR and ENR algorithms. First, we compare the human efforts of the two algorithms and show that the ENR algorithm performs significantly better than the GNR algorithm except for cases where binary inconsistencies occur frequently. Next, we compare the system runtimes of the algorithms by analyzing the major runtime factors and conducting scalability tests. We also ran our experiments on a comparison shopping dataset provided by Yahoo! Shopping, and the results are analogous to those of the hotel dataset (see [30] for details).

7.1 Human Effort

Record Selection Cost and Rule Checks We measured the human efforts for the two algorithms on 1,000 to 5,000 U.S. hotel records using the phone number negative rules and the threshold $T_M = 0.74$. We used the Solver strategy from Section 6 for resolving records.

Figure 9 shows that the ENR algorithm requires much less solver effort than the GNR algorithm. The significantly larger selection cost for the GNR algorithm compared to the ENR algorithm is due to the highly redundant records views, which can be illustrated by the following example. Suppose that a set of initial records has a merge closure size of 100. Moreover, suppose that the initial records form ten connected components where each component has a merge closure size of 10. For simplicity, we ignore the inconsistency and domination relationships among records. For the GNR algorithm, the solver must view $\sum_{i=2}^{100} i = 5499$ records; for the ENR algorithm, the solver only needs to view $10 \times \sum_{i=2}^{10} i = 540$ records, which is about one-tenth the effort of the GNR algorithm effort. Although the merge closure is the same for both algorithms, the ENR algorithm saves a lot of redundant views because it partitions the merge closure into many smaller independent components.

Binary Density Impact However, the ENR algorithm does not always perform better than the GNR algorithm. In the case where many binary inconsistencies occur, the ENR algorithm loses the advantage of dividing the merge closure into many smaller components and handling one component at a time. To capture the degree of binary inconsistencies, we define the binary density measure as the ratio between

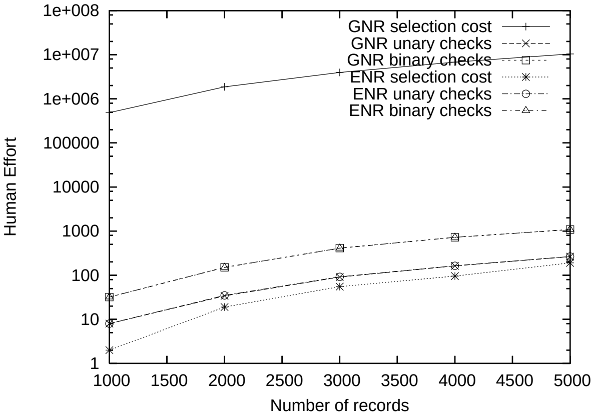

Fig. 9 Human effort

the number of inconsistent record pairs in $\bar{I}$ and the number of all the possible record pairs in $\bar{I}$. For example, if five records are inconsistent with each other among ten records of $\bar{I}$, the binary density is $\binom{5}{2}/\binom{10}{2}$ = 10/45 ≈ 0.202. For our experiments, we used the NameAddr negative rules on 5,000 U.S. hotel records and varied the binary density by changing the binary threshold T_B. A lower T_B results in a higher binary density because a pair of records is more likely to be inconsistent.

Figure 10 shows how human effort and binary density relate. For small binary densities, the ENR algorithm has a much lower record selection cost than GNR because the component sizes are small, minimizing the time for running the GNR algorithm on each component. As the binary density increases, however, the components get larger, and running the GNR algorithm on them takes longer. For high binary densities, the benefits of the ENR algorithm disappear because the merge closure is no longer partitioned into smaller components, and the selection cost of ENR becomes close to that of the GNR algorithm.

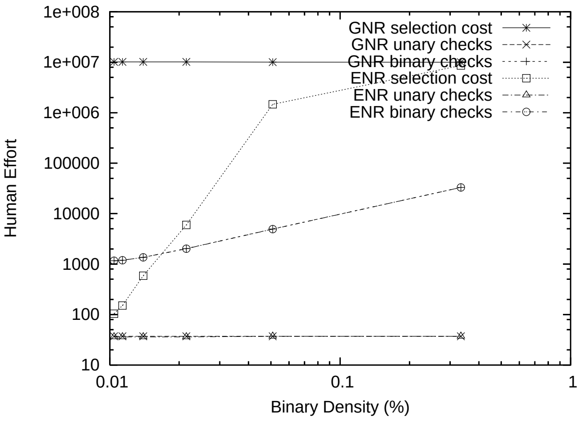

Fig. 10 Binary density impact on human effort

7.2 System Runtime

Runtime decomposition Figure 11 shows the runtime decomposition of the GNR and ENR algorithms using the phone number negative rules on 5,000 U.S. hotel records. We show the total runtimes and major runtime factors for each algorithm. The majority of the GNR runtime is used for managing the domination relationships between records in the merge closure in order to find the non-dominated records in S (step 6 of GNR). The next longest task for GNR is invoking the binary negative rule. The major runtime factors for the ENR algorithm are running the R-Swoosh algorithm and connecting the inconsistent components. Comparing the total runtimes, the ENR algorithm is 2.2 to 2.5 times faster than GNR.

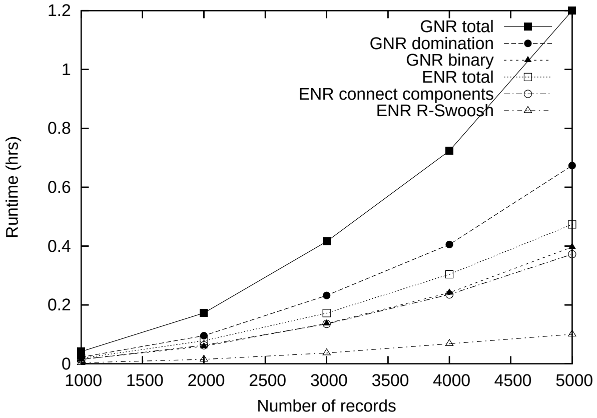

Fig. 11 Runtime decomposition

Scalability We conducted scalability tests for the GNR and ENR algorithms using the phone number negative rules on 1,000 to 27,000 records randomly selected (regardless of the country) from the entire hotel dataset provided by Yahoo!. (The entire dataset size was 27,049 records.) We used a slightly higher match threshold than usual ($T_M = 0.8$) in order to properly match non-U.S. hotels. For example, using the threshold $T_M = 0.74$ on the French hotels resulted in many different hotels incorrectly merging with each other. Figure 12 shows that the GNR algorithm cannot handle more than 17,000 records in a reasonable time while the ENR algorithm shows a quadratic growth in runtime. As the dataset gets larger, ENR outperforms GNR by up to 3.89 times.

In summary, although the ENR algorithm outperforms the GNR algorithm both in human effort and scalability, ER-N is an inherently expensive process and thus only relatively small sets of records can be handled. Thus, large data sets need to be partitioned into smaller sets (e.g., using blocking techniques) that can be resolved in detail. How large a data set can be exhaustively resolved depends on the application. For example, in our recent work on scaling ER on 2 million Yahoo! Shopping records [31], the average block size was 124 records while the maximum block size was 6082

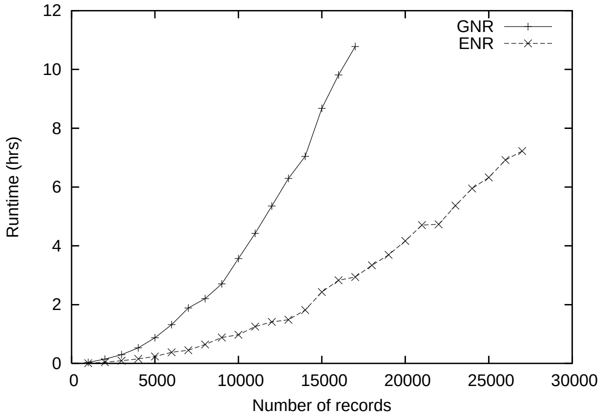

Fig. 12 Scalability

records. It is also possible to distribute the ER-N computations across multiple processors, in order to handle larger data sets. We can use techniques similar to the ones in [1] to distribute the data and computation among processors. There are also applications that do not require an exhaustive comparison on the entire data set. For example, a technique called Data Dipping compares a given record only with a small subset of candidate records that are likely to match the record.

7.3 Without the Properties

So far, we have only studied scenarios where the properties for the negative rules (Section 4.2) hold. We now consider a scenario where the properties do not hold. (We still assume the ICAR properties hold for the match and merge rules. See reference [2] for an extensive study on using the R-Swoosh algorithm without the ICAR properties.) In this case, we need to run the GNR algorithm for a correct ER-N answer. From our previous results, however, GNR can be very expensive in runtime. The alternatives are to either modify the negative rules to satisfy the properties or run ENR even though we might not get the correct ER-N answer. In this section, we consider the second alternative and investigate how similar the ENR result is to the GNR result.

We use a modified version of the NameAddress negative rules (defined in Section 6.4) where we only compare the longest strings. Specifically, the binary negative rule now compares the longest names and addresses of the two records while the unary negative rule compares the two longest names and two longest addresses of a single record. Although the Commutativity property is satisfied, the Unary and Binary Persistence properties are not guaranteed because the longest name and address of a record could change after a record merge, possibly making a previously inconsistent record consistent.

Figure 13 shows a comparison of the GNR and ENR results when we use the modified NameAddress negative rules and vary the T_U and T_B thresholds. We experimented on the 5,000 U.S. hotel records using the Solver strategy. For each possible threshold pair, we show the total number of records for each result (columns 2 and 3) and the Jaccard similarity between the two results (column 4). Given a GNR solution G and ENR solution E, the Jaccard similarity between the two results is defined as $\frac{|G \cap E|}{|G \cup E|}$. The average Jaccard similarity is 99.41%, making the ENR result almost identical to the GNR result.

T_B/T_U	**GNR**	**ENR**	**Jaccard Similarity**
0.65/0.8	4445	4445	100.0
0.65/0.9	4445	4445	100.0
0.65/0.99	4445	4445	100.0
0.70/0.8	4478	4454	99.38
0.70/0.9	4478	4454	99.38
0.70/0.99	4478	4454	99.38
0.75/0.8	4525	4475	98.85
0.75/0.9	4525	4475	98.85
0.75/0.99	4525	4475	98.85

Fig. 13 Result sizes and similarities

In summary, the ENR algorithm is a reasonable way to compute an ER-N result even when the properties for the negative rules do not hold. The experimental results show that the incorrectness of the ENR result is very small in practice.

8 Related Work

Entity Resolution has been studied under various names including record linkage [24], merge/purge [19], deduplication [27], reference reconciliation [10], object identification [29], and others (see [11,33,17] for recent surveys). Most of these works focus on exploiting positive rules to improve the accuracy of record matching. In contrast, our ER-N model provides a general framework for both positive and negative rules where the match, merge, and negative rules are black-box functions.

Several works [8,9,28,3,10] have addressed the use of negative rules. Doan et al. [8,9] introduced constraints to perform sanity checks for object matching. Dong et al. [10] used dependency graphs while Bhattacharya and Getoor [3] used negative relational evidence to improve the accuracy of the constraints. Shen et al. [28] provided a probabilistic interpretation of constraints and categorized them according to their semantics. However, the constraints used in the works above are local in a sense that they only prevent two records from incorrectly matching. To the best of our knowledge, our work is the first to introduce binary negative rules, which require a global view of records for detecting inconsistencies.

A recent work [6] uses aggregate constraints to improve the accuracy of record clustering. Their goal is to partition the initial set of records such that the number of constraint violations is minimized. The textual similarity between tuples is used to restrict the search space of partitions. Our work complements the above work in several ways. First, we guarantee a correct and maximal solution as opposed to using the constraints as a search heuristic. Second, we take a pair-wise approach, which is an alternative to their clustering approach. Finally, we consider record merges, which a clustering approach does not directly support, and do integrity checks on the merged records. Record merges are important for our pair-wise approach because they naturally provide the lineage for each record in the ER result, making it easy to view intermediate states of the ER process. Although negative rules could also be used in clustering approaches, we would need to define the notion of intermediate states for clusters.

A related topic to our work is maintaining integrity constraints in relational databases [18,12]. Active database systems [32] use triggers and rules to provide mechanisms for integrity constraints. More recently, a line of research [7,4] shows how to "repair" an inconsistent database into a consistent one while minimizing the difference. The possible repair actions are deleting, inserting, and modifying tuples. While the motivations of providing integrity constraints are similar to ours, the works above do not address the additional complexity of iteratively matching and merging records. Moreover, their focus is on specific constraints such as functional dependencies and inclusion dependencies.

Another related line of work is called statistical data editing [13,34,14] where missing or contradictory data is edited and imputed for intended analytic purposes. While statistical data editing is focused on fixing the data itself, our work complements this approach by improving the matching process of data (records) using negative rules.

An interesting analogy for negative rules can be found in a topic called non-monotonic reasoning [15,25,16]. Unlike conventional logic, a non-monotonic inference can later be retracted by contrary information. Although retracting inferences is similar to applying negative rules, the main focus of non-monotonic reasoning is to deduce whether a single statement is true or false. In contrast, we are trying to find all the records in the solution (in logic terminology, the theory) efficiently. Another difference is that, while non-monotonic reasoning does not alter its basic beliefs, an ER-N algorithm can discard base records that have incorrect data.

9 Conclusion

For the entity resolution process, unary and binary negative rules capture "sanity checks" typically written by domain specialists who are different from the ones writing the match and merge rules. As far as we know, our work is the first to formally define what *correct* and *maximal* entity resolution means in the context of negative rules (Section 2.3). Such a logical and formal foundation is critical for developing ER-N algorithms: it is easy to develop algorithms that apply rules in an ad-hoc fashion and give "some sort of answer." However, here we have presented two algorithms that are proven to give the correct answer. Another aspect that is often overlooked is that entity resolution often requires human guidance to handle unexpected situations and erroneous real-world data. Our algorithms demonstrate how a human "solver" can guide the resolution process. One of our algorithms (GNR) represents a generic way to solve ER-N while the the other (ENR) makes the GNR algorithm efficient by exploiting additional properties for the match, merge, and negative rules.

Neither of our solutions is perfect: the GNR algorithm can be expensive unless used for small data sets or when there are few matching records. The ENR algorithm is only guaranteed to return a correct solution if certain properties hold, and these properties may not hold in some applications. However, the algorithms can be used if one is willing to tolerate some loss in accuracy: As we discussed, one can partition the input data set and only run ER-N on a partition (bucket) at a time. Also, one can run the ENR algorithm when the properties do not hold, which introduces additional inaccuracies. Evaluating such inaccuracies is beyond the scope of this paper.

There may of course be applications where negative rules simply introduce too much cost, even when shortcuts are taken. Entity resolution with negative rules is inherently expensive, but we believe it is important to understand the options and their costs, so that application developers can make informed decisions.

References

1. 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)
2. Benjelloun, O., Garcia-Molina, H., Menestrina, D., Whang, S.E., Su, Q., Widom, J.: Swoosh: a generic approach to entity resolution. VLDB J. (2008). DOI http://dx.doi.org/10.1007/s00778-008-0098-x
3. Bhattacharya, I., Getoor, L.: Relational clustering for multi-type entity resolution. In: MRDM '05: Proceedings of the 4th international workshop on Multi-relational mining, pp. 3–12. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1090193.1090195
4. Bohannon, P., Flaster, M., Fan, W., Rastogi, R.: A cost-based model and effective heuristic for repairing constraints by value modification. In: SIGMOD Conference, pp. 143–154 (2005)
5. Chaudhuri, S., Ganti, V., Motwani, R.: Robust identification of fuzzy duplicates. In: Proc. of ICDE. Tokyo, Japan (2005)
6. Chaudhuri, S., Sarma, A.D., Ganti, V., Kaushik, R.: Leveraging aggregate constraints for deduplication. In: SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 437–448. ACM Press, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1247480.1247530
7. Chomicki, J., Marcinkowski, J.: On the computational complexity of minimal-change integrity maintenance in relational databases. In: Inconsistency Tolerance, pp. 119–150 (2005)
8. Doan, A., Lu, Y., Lee, Y., Han, J.: Object matching for information integration: A profiler-based approach. In: IIWeb, pp. 53–58 (2003)
9. Doan, A., Lu, Y., Lee, Y., Han, J.: Profile-based object matching for information integration. IEEE Intelligent Systems **18**(5), 54–59 (2003)
10. Dong, X., Halevy, A.Y., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference, pp. 85–96 (2005)
11. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007)

12. Eswaran, K.P., Chamberlin, D.D.: Functional specifications of subsystem for database integrity. In: VLDB, pp. 48–68 (1975)
13. Fellegi, I.P., Holt, D.: A systematic approach to automatic edit and imputation. Journal of the American Statistical Association **71**(353), 17–35 (1976). URL http://www.jstor.org/stable/2285726
14. Franconi, E., Palma, A.L., Leone, N., Perri, S., Scarcello, F.: Census data repair: a challenging application of disjunctive logic programming. In: LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming, pp. 561–578. Springer-Verlag, London, UK (2001)
15. Genesereth, M.R., Nilsson, N.J.: Logical Foundations of Artificial Intelligence. Morgan Kaufmann, Palo Alto, CA (1988)
16. Ginsberg, M.L.: Readings in Nonmonotonic Reasoning. Morgan Kaufmann, Los Altos, CA (1987)
17. 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)
18. Hammer, M., McLeod, D.: Semantic integrity in a relational data base system. In: VLDB, pp. 25–47 (1975)
19. Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: Proc. of ACM SIGMOD, pp. 127–138 (1995)
20. 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)
21. 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)
22. Menestrina, D., Benjelloun, O., Garcia-Molina, H.: Generic Entity Resolution with Data Confidences. In: First Int'l VLDB Workshop on Clean Databases. Seoul, Korea (2006)
23. 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)
24. Newcombe, H.B., Kennedy, J.M., Axford, S.J., James, A.P.: Automatic linkage of vital records. Science **130**(3381), 954–959 (1959)
25. Nilsson, N.J.: Artificial Intelligence: A New Synthesis. Morgan Kaufmann, San Francisco, CA (1998)
26. Rowland, T.: Connected component. Http://mathworld.wolfram.com/ConnectedComponent.html
27. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: Proc. of ACM SIGKDD. Edmonton, Alberta (2002)
28. Shen, W., Li, X., Doan, A.: Constraint-based entity matching. In: AAAI, pp. 862–867 (2005)
29. Tejada, S., Knoblock, C.A., Minton, S.: Learning object identification rules for information integration. Information Systems Journal **26**(8), 635–656 (2001)
30. Whang, S.E., Benjelloun, O., Garcia-Molina, H.: Additional experiments on negative rules. Tech. rep., Stanford University. URL http://dbpubs.stanford.edu/pub/2005-5
31. Whang, S.E., Menestrina, D., Koutrika, G., Theobald, M., Garcia-Molina, H.: Entity resolution with iterative blocking. Tech. rep., Stanford University (2008). URL http://dbpubs.stanford.edu/pub/2008-19
32. Widom, J., Ceri, S. (eds.): Active Database Systems: Triggers and Rules For Advanced Database Processing. Morgan Kaufmann (1996)
33. Winkler, W.: Overview of record linkage and current research directions. Tech. rep., Statistical Research Division, U.S. Bureau of the Census, Washington, DC (2006)
34. Winkler, W.E.: State of statistical data editing and current research problems. In: UN/ECE Work Session on Statistical Data Editing, Working Paper n.29, pp. 2–4 (1999)