Entity Resolution with Evolving Rules

Steven Euijong Whang and Hector Garcia-Molina
Computer Science Department
Stanford University
353 Serra Mall, Stanford, CA 94305, USA
{swhang,hector}@cs.stanford.edu

ABSTRACT

Entity resolution (ER) identifies database records that refer to the same real world entity. In practice, ER is not a one-time process, but is constantly improved as the data, schema and application are better understood. We address the problem of keeping the ER result up-to-date when the ER logic "evolves" frequently. A naïve approach that re-runs ER from scratch may not be tolerable for resolving large datasets. This paper investigates when and how we can instead exploit previous "materialized" ER results to save redundant work with evolved logic. We introduce algorithm properties that facilitate evolution, and we propose efficient rule evolution techniques for two clustering ER models: match-based clustering and distance-based clustering. Using real data sets, we illustrate the cost of materializations and the potential gains over the naïve approach.

1. INTRODUCTION

Entity resolution [7, 21, 13] (also known as record linkage or deduplication) is the process of identifying records that represent the same real-world entity. For example, two companies that merge may want to combine their customer records. In such a case, the same customer may be represented by multiple records, so these matching records must be identified and combined (into what we will call a cluster). This ER process is often extremely expensive due to very large data sets and complex logic that decides when records represent the same entity.

In practice, an entity resolution (ER) result is not produced once, but is constantly improved based on better understandings of the data, the schema, and the logic that examines and compares records. In particular, here we focus on changes to the logic that compares two records. We call this logic the *rule*, and it can be a Boolean function that determines if two records represent the same entity, or a distance function that quantifies how different (or similar) the records are. Initially we start with a set of records S, then produce a first ER result E_1 based on S and a rule B_1. Some time later rule B_1 is improved yielding rule B_2, so we need to compute a new ER result E_2 based on S and B_2. The process continues with new rules B_3, B_4 and so on.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

Record	Name	Zip	Phone
r_1	$John$	54321	123-4567
r_2	$John$	54321	987-6543
r_3	$John$	11111	987-6543
r_4	Bob	$null$	121-1212

Figure 1: Records to resolve

Comparison Rule	Definition
B_1	p_{name}
B_2	$p_{name} \wedge p_{zip}$
B_3	$p_{name} \wedge p_{phone}$

Figure 2: Evolving from rule B_1 to rule B_2

A naïve approach would compute each new ER result from scratch, starting from S, a potentially very expensive proposition. Instead, in this paper we explore an incremental approach, where for example we compute E_2 based on E_1. Of course for this approach to work, we need to understand how the new rule B_2 relates to the old one B_1, so we can understand what changes incrementally in E_1 to obtain E_2. As we will see, our incremental approach may yield large savings over the naïve approach, but not in all cases.

To motivate and explain our approach, consider the following example. Our initial set of people records S is shown in Figure 1. The first rule B_1 (see Figure 2) says that two records match (represent the same real world entity) if predicate p_{name} evaluates to true. Predicates can in general be quite complex, but for this example assume that predicates simply perform an equality check. The ER algorithm calls on B_1 to compare records and groups together records with name "John", producing the result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$. (As we will see, there are different types of ER algorithms, but in this simple case most would return this same result.)

Next, say users are not satisfied with this result, so a data administrator decides to refine B_1 by adding a predicate that checks zip codes. Thus, the new rule is B_2 shown in Figure 2. The naïve option is to run the same ER algorithm with rule B_2 on set S to obtain the partition $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. (Only records r_1 and r_2 have the same name and same zip code.) This process repeats much unnecessary work: For instance, we would need to compare r_1 with r_4 to see if they match on name and zip code, but we already know from the first run that they do not match on name (B_1), so they cannot match under B_2.

Because the new rule B_2 is stricter than B_1 (we define this term precisely later on), we can actually start the second ER from the first result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$. That is, we only need to check each cluster separately and see if it needs to split. In our example, we find that r_3 does not match the other records in its cluster, so we arrive at $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. This approach only works if the ER algorithm satisfies certain properties and B_2 is stricter than B_1. If B_2 is not stricter and the ER algorithm satisfies different properties, there are other incremental techniques we can apply. Our long-term goal in this paper is to explore these options: Under what conditions and for what ER algorithms are incremental approaches *feasible*? And in what scenarios are the savings over the naïve approach significant?

In addition, we study a complementary technique: *materialize* auxiliary results during one ER run, in order to improve the performance of future ER runs. To illustrate, say that when we process $B_2 = p_{name} \wedge p_{zip}$, we concurrently produce the results for each predicate individually. That is, we compute three separate partitions, one for the full B_2, one for rule p_{name} and one for rule p_{zip}. The result for p_{name} is the same $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ seen earlier. For p_{zip} it is $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. As we will see later, the cost of computing the two extra materializations can be significantly lower than running the ER algorithm three times, as a lot of the work can be shared among the runs.

The materializations pay off when rule B_2 evolves into a related rule that is not quite stricter. For example, say that B_2 evolves into $B_3 = p_{name} \wedge p_{phone}$, where p_{phone} checks for matching phone numbers. In this case, B_3 is not stricter than B_2 so we cannot start from the B_2 result. However, we can start from the p_{name} result, since B_3 is stricter than p_{name}. Thus, we independently examine each cluster in $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, splitting the first cluster because r_2 has a different phone number. The final result is $\{\{r_1, r_3\}, \{r_2\}, \{r_4\}\}$. Clearly, materialization of partial results may or may not pay off, just like materialized views and indexes may or may not help. Our long-term objective here is, again, to study when is materialization *feasible* and to show scenarios where it can pay off.

In summary, our contributions in this paper are as follows:

- We formalize rule evolution for two general types of record comparison rules: Boolean match functions and distance-based functions. We identify two desirable properties of ER algorithms (rule monotonic and context free) that enable efficient rule evolution. We also contrast these properties to two properties mentioned in the literature (order independent and incremental). We categorize a number of existing ER algorithms based on the properties they satisfy. (Existing ER algorithms are reviewed in Appendixes A.1 and B.1, while other related work is in Appendix E.) We then propose efficient rule evolution techniques that use one or more of the four properties (Sections 2 and 3). We believe that our results can be a useful guide for ER algorithm designers: if they need to handle evolving rules efficiently, they may want to build algorithms that have at least some of the properties we present.
- We experimentally evaluate (Section 4) the rule evolution algorithms for various ER algorithms using actual comparison shopping data from Yahoo! Shopping and hotel information from Yahoo! Travel. Our results show scenarios where rule evolution can be faster than the naïve approach by up to several orders of magnitude. We also illustrate the time and space cost of materializing partial results, and argue that these costs can be amortized with a small number of future evolutions. Finally, we also experiment with ER algorithms that do not satisfy our properties, and show that if one is willing to sacrifice accuracy, one can still use our rule evolution techniques.

2. MATCH-BASED EVOLUTION

We consider rule evolution for ER algorithms that cluster records based on Boolean comparison rules. (We consider ER algorithms based on distance functions in Section 3.) We first formalize an ER model that is based on clustering. We then discuss two important properties for ER algorithms that can significantly enhance the runtime of rule evolution. We also compare the two properties with existing properties for ER algorithms in the literature. Finally, we present efficient rule evolution algorithms that use one or more of the four properties.

2.1 Match-based Clustering Model

We define a Boolean comparison rule B as a function that takes two records and returns `true` or `false`. We assume that B is commutative, i.e., $\forall r_i, r_j, B(r_i, r_j) = B(r_j, r_i)$.

Suppose we are given a set of records $S = \{r_1, \ldots, r_n\}$. An ER algorithm receives as inputs a partition P_i of S and a Boolean comparison rule B, and returns another partition P_o of S. A partition of S is defined as a set of clusters $P = \{c_1, \ldots, c_m\}$ such that $c_1 \cup \ldots \cup c_m = S$ and $\forall c_i, c_j \in P$ where $i \neq j$, $c_i \cap c_j = \emptyset$.

We require the input to be a partition of S so that we may also run ER on the output of a previous ER result. In our motivating example in Section 1, the input was a set of records $S = \{r_1, r_2, r_3, r_4\}$, which can be viewed as a partition of singletons $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$, and the output using the comparison rule $B_2 = p_{name} \wedge p_{zip}$ was the partition $P_o = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. If we run ER a second time on the ER output $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, we may obtain the new output partition $P_o = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ where the cluster $\{r_1, r_2\}$ accumulated enough information to match with the cluster $\{r_3\}$.

How exactly the ER algorithm uses B to derive the output partition P_o depends on the specific ER algorithm. The records are clustered based on the results of B when comparing records. In our motivating example (Section 1), all pairs of records that matched according to $B_2 = p_{name} \wedge p_{zip}$ were clustered together. Note that, in general, an ER algorithm may not cluster two records simply because they match according to B. For example, two records r and s may be in the same cluster $c \in P_o$ even if $B(r, s) = $ `false`. Or the two records could also be in two different clusters $c_i, c_j \in P_o$ $(i \neq j)$ even if $B(r, s) = $ `true`.

We also allow input clusters to be un-merged as long as the final ER result is still a partition of the records in S. For example, given an input partition $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, an output of an ER algorithm could be $\{\{r_1, r_2\}, \{r_3, r_4\}\}$ and not necessarily $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ or $\{\{r_1, r_2, r_3, r_4\}\}$. Un-merging could occur when an ER algorithm decides that some records were incorrectly clustered [18].

Finally, we assume the ER algorithm to be *non-deterministic* in a sense that different partitions of S may be produced depending on the order of records processed or by some random factor (e.g., the ER algorithm could be a randomized algorithm). For example, a hierarchical clustering algorithm based on Boolean rules (see Appendix A.1) may produce different partitions depending on which records are compared first. While the ER algorithm is non-deterministic, we assume the comparison rule itself to be deterministic, i.e., it always returns the same matching result for a given pair of records.

We now formally define a valid ER algorithm.

DEFINITION 2.1. *Given any input partition P_i of a set of records S and any Boolean comparison rule B, a* valid *ER algorithm E non-deterministically returns an ER result $E(P_i, B)$ that is also a partition P_o of S.*

We denote all the possible partitions that can be produced by the ER algorithm E as $\bar{E}(P_i, B)$, which is a set of partitions of S. Hence, $E(P_i, B)$ is always one of the partitions in $\bar{E}(P_i, B)$. For example, given $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, $\bar{E}(P_i, B)$ could be

$\{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$ while $E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}$.

A rule evolution occurs when a Boolean comparison rule B_1 is replaced by a new Boolean comparison rule B_2. An important concept used throughout the paper is the relative strictness between comparison rules:

DEFINITION 2.2. *A Boolean comparison rule B_1 is* stricter than *another rule B_2 (denoted as $B_1 \leq B_2$) if $\forall r_i, r_j$, $B_1(r_i, r_j)$ =* `true` *implies $B_2(r_i, r_j)$ =* `true`.

For example, a comparison rule B_1 that compares the string distance of two names and returns `true` when the distance is lower than 5 is stricter than a comparison rule B_2 that uses a higher threshold of, say, 10. As another example, a comparison rule B_1 that checks whether the names and addresses are same is stricter than another rule B_2 that only checks whether the names are same.

2.2 Properties

We introduce two important properties for ER algorithms – *rule monotonicity* and *context free* – that enable efficient rule evolution for match-based clustering.

2.2.1 Rule Monotonicity

Before defining the rule monotonicity property, we first define the notion of refinement between partitions.

DEFINITION 2.3. *A partition P_1 of a set S* refines *another partition P_2 of S (denoted as $P_1 \leq P_2$) if $\forall c_1 \in P_1, \exists c_2 \in P_2$ s.t. $c_1 \subseteq c_2$.*

For example, given the partitions $P_1 = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}, \{r_4\}\}$, $P_1 \leq P_2$ because $\{r_1, r_2\}$ and $\{r_3\}$ are subsets of $\{r_1, r_2, r_3\}$ while $\{r_4\}$ is a subset of $\{r_4\}$.

We now define the rule monotonicity property, which guarantees that the stricter the comparison rule, the more refined the ER result.

DEFINITION 2.4. *An ER algorithm is* rule monotonic *($\mathcal{RM}$) if, for any three partitions P, P_o^1, P_o^2 and two comparison rules B_1 and B_2 such that*

- *$B_1 \leq B_2$ and*
- *$P_o^1 \in \bar{E}(P, B_1)$ and*
- *$P_o^2 \in \bar{E}(P, B_2)$*

then $P_o^1 \leq P_o^2$.

An ER algorithm satisfying $\mathcal{RM}$ guarantees that, if the comparison rule B_1 is stricter than B_2, the ER result produced with B_1 refines the ER result produced with B_2. For example, suppose that $P = \{\{r_1\},\{r_2\},\{r_3\},\{r_4\}\}$, $B_1 \leq B_2$, and $E(P_i, B_1) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$. If the ER algorithm is $\mathcal{RM}$, $E(P_i, B_2)$ can only return $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ or $\{\{r_1, r_2, r_3, r_4\}\}$.

2.2.2 Context Free

The second property, context free, tells us when a subset of P_i can be processed "in isolation" from the rest of the clusters. (For clarification, the second conditions says that none of the records in P can match with any of the records in $P_i - P$.)

DEFINITION 2.5. *An ER algorithm is* context free *($\mathcal{CF}$) if for any four partitions P, P_i, P_o^1, P_o^2 and a comparison rule B such that*

- *$P \subseteq P_i$ and*
- *$\forall P_o \in \bar{E}(P_i, B)$, $P_o \leq \{\bigcup_{c \in P} c, \bigcup_{c \in P_i - P} c\}$ and*
- *$P_o^1 \in \bar{E}(P, B)$ and*
- *$P_o^2 \in \bar{E}(P_i - P, B)$*

then $P_o^1 \cup P_o^2 \in \bar{E}(P_i, B)$.

Suppose that we are resolving $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$ with the knowledge that no clusters in $P = \{\{r_1\}, \{r_2\}\}$ will merge with any of the clusters in $P_i - P = \{\{r_3\}, \{r_4\}\}$. Then for any $P_o \in \bar{E}(P_i, B)$, $P_o \leq \{\{r_1, r_2\}, \{r_3, r_4\}\}$. In this case, an ER algorithm that is $\mathcal{CF}$ can resolve $\{\{r_1\}, \{r_2\}\}$ independently from $\{\{r_3\}, \{r_4\}\}$, and there exists an ER result of P_i that is the same as the union of the ER results of $\{\{r_1\}, \{r_2\}\}$ and $\{\{r_3\}, \{r_4\}\}$.

2.2.3 Existing ER Properties

To get a better understanding of $\mathcal{RM}$ and $\mathcal{CF}$, we compare them to two existing properties in the literature: incremental and order independence.

An ER algorithm is incremental [13] if it can resolve one record at a time. We define a more generalized version of the incremental property for our ER model where any subsets of clusters in P_i can be resolved at a time.

DEFINITION 2.6. *An ER algorithm is* general incremental *($\mathcal{GI}$) if for any four partitions P, P_i, P_o^1, P_o^2, and a comparison rule B such that*

- *$P \subseteq P_i$ and*
- *$P_o^1 \in \bar{E}(P, B)$ and*
- *$P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), B)$*

then $P_o^2 \in \bar{E}(P_i, B)$.

For example, suppose we have $P = \{\{r_1\}, \{r_2\}\}$, $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and $P_o^1 = \{\{r_1, r_2\}\}$. That is, we have already resolved P into the result P_o^1. We can then add to P_o^1 the remaining cluster $\{r_3\}$, and resolve all the clusters together. The result is as if we had resolved everything from scratch (i.e., from P_i). Presumably, the former way (incremental) will be more efficient than the latter.

The $\mathcal{GI}$ property is similar to the $\mathcal{CF}$ property, but also different in a number of ways. First $\mathcal{GI}$ and $\mathcal{CF}$ are similar in a sense that they use two subsets of P_i: P and $P_i - P$. However, under $\mathcal{GI}$, $P_i - P$ is not resolved until P has been resolved. Also, $\mathcal{GI}$ does not assume P and $P_i - P$ to be independent (i.e., a cluster in P may merge with a cluster in $P_i - P$).

We now explore the second property in the literature. An ER algorithm is order independent ($\mathcal{OI}$) [13] if the ER result is same regardless of the order of the records processed. That is, for any input partition P_i and comparison rule B, $\bar{E}(P_i, B)$ is a singleton (i.e., $\bar{E}(P_i, B)$ contains exactly one partition of S).

2.2.4 ER Algorithm Categorization

To see how the four properties $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$ hold in practice, we consider several ER algorithms in the literature: SN, HC_B, HC_{BR}, and ME (for their definitions, see Appendix A.1). The venn diagram in Figure 3 shows which ER algorithms satisfy which of the four properties. The SN^2 and HC_B^2 algorithms are variants of the SN and HC_B algorithms, respectively, and are discussed in our technical report [19]. For now, ignore the HC_{DS} and HC_{DC} algorithms, which are distance-based clustering algorithms covered in Section 3.2. All the proofs for verifying Figure 3 can be found in our technical report [19].

2.3 Materialization

To improve our chances that we can efficiently compute a new ER result with rule B_2, when we compute earlier results we can materialize results that involve predicates likely to be in B_2. In particular, let us assume that rules are Boolean expressions of smaller binary predicates. For example, a rule that compares the names and

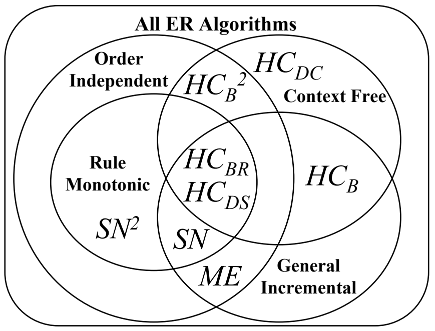

Figure 3: ER Algorithms satisfying properties

addresses of two people can be defined as $p_{name} \wedge p_{address}$ where p_{name} could be a function that compares the names of two people while the predicate $p_{address}$ could compare the street addresses and apartment numbers of two people. In general, a predicate can be any function that compares an arbitrary number of attributes. We assume that all predicates are commutative and (without loss of generality) all rules are in conjunctive normal form (CNF). For example, the rule $B = p_1 \wedge p_2 \wedge (p_3 \vee p_4)$ is in CNF and has three conjuncts p_1, p_2, and $p_3 \vee p_4$.

When we compute an earlier result $E(P_i, B_1)$ where say $B_1 = p_1 \wedge p_2 \wedge p_3$, we can also materialize results such as $E(P_i, p_1)$, $E(P_i, p_2)$, $E(P_i, p_1 \wedge p_2)$, and so on. The most useful materializations will be those that can help us later with $E(P_i, B_2)$. (See Appendix C.) For concreteness, here we will assume that we materialize *all* conjuncts of B_1 (in our example, $E(P_i, p_1)$, $E(P_i, p_2)$, and $E(P_i, p_3)$).

Instead of serially materializing each conjunct, however, we can amortize the common costs by materializing different conjuncts in a concurrent fashion. For example, parsing and initializing the records can be done once during the entire materialization. More operations can be amortized depending on the given ER algorithm. For example, when materializing conjuncts using an ER algorithm that always sorts its records before resolving them, the records only need to be sorted once for all materializations. In Section 4.4, we show that amortizing common operations can significantly reduce the time overhead of materializing conjuncts. A partition of the records in S can be stored compactly in various ways. One approach is to store sets of records IDs in a set where each inner set represents a cluster of records. A possibly more space-efficient technique is to maintain an array A of records (where the ID is used as the index) where each cell contains the cluster ID. For example, if r_5 is in the second cluster, then $A[5] = 2$. If there are only a few clusters, we only need a small number of bits for saving each cluster ID. For example, if there are only 8 clusters, then each entry in A only takes 3 bits of space.

2.4 Rule Evolution

We provide efficient rule evolution techniques for ER algorithms using the properties. Our first algorithm supports ER algorithms that are $\mathcal{RM}$ and $\mathcal{CF}$. As we will see, rule evolution can still be efficient for ER algorithms that are only $\mathcal{RM}$. Our second algorithm supports ER algorithms that are $\mathcal{GI}$. Before running the rule evolution algorithms, we materialize ER results for conjuncts of the old comparison rule B_1 by storing a partition of the input records S (i.e., the ER result) for each conjunct in B_1 (see Appendix C for possible optimizations). In general, we suspect that (although we will not explicitly show) the number of properties satisfied by the ER algorithm is correlated with better runtime performance.

To explain our rule evolution algorithms, we review a basic operation on partitions. The *meet* of two partitions P_1 and P_2 (denoted as $P_1 \wedge P_2$) returns a new partition of S whose members are the non-empty intersections of the clusters of P_1 with those of P_2. For example, given the partitions $P_1 = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and $P_2 = \{\{r_1\}, \{r_2, r_3, r_4\}\}$, the meet of P_1 and P_2 becomes $\{\{r_1\}, \{r_2, r_3\}, \{r_4\}\}$ since r_2 and r_3 are clustered in both partitions.

Algorithm 1 performs rule evolution for ER algorithms that are both $\mathcal{RM}$ and $\mathcal{CF}$. The input requires the input partition P_i, the old and new comparison rules (B_1 and B_2, respectively), and a hash table H that contains the materialized ER results for the conjuncts of B_1. The conjuncts of a comparison rule B is denoted as $Conj(B)$. For simplicity, we assume that B_1 and B_2 share at least one conjunct. Step 3 exploits the $\mathcal{RM}$ property and meets the partitions of the common conjuncts between B_1 and B_2. For example, suppose that we have $B_1 = p_1 \wedge p_2 \wedge p_3$ and $B_2 = p_1 \wedge p_2 \wedge p_4$. Given $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$, say we also have the materialized ER results $E(P_i, p_1) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and $E(P_i, p_2) = E(P_i, p_3) = \{\{r_1\}, \{r_2, r_3, r_4\}\}$. Since the common conjuncts of B_1 and B_2 are p_1 and p_2, we generate the meet of $E(P_i, p_1)$ and $E(P_i, p_2)$ as $M = \{\{r_1\}, \{r_2, r_3\}, \{r_4\}\}$. By $\mathcal{RM}$, we know that $E(P_i, B_2)$ refines M because B_2 is stricter than both p_1 and p_2. That is, each cluster in the new ER result is contained in exactly one cluster in the meet M. Step 4 then exploits the $\mathcal{CF}$ property to resolve for each cluster c of M, the clusters in P_i that are subsets of c (i.e., $\{c' \in P_i | c' \subseteq c\}$). Since the clusters in different $\{c' \in P_i | c' \subseteq c\}$'s do not merge with each other, each $\{c' \in P_i | c' \subseteq c\}$ can be resolved independently. As a result, we can return $\{\{r_1\}\} \cup E(\{\{r_2\}, \{r_3\}\}, B_2) \cup \{\{r_4\}\}$ as the new ER result of B_2.

```
1: input: The input partition $P_i$, the comparison rules $B_1$, $B_2$, the
   ER result for each conjunct of $B_1$, the hash table $H$ containing
   materializations of conjuncts in $B_1$
2: output: The output partition $P_o \in \bar{E}(P_i, B_2)$
3: Partition $M \leftarrow \bigwedge_{conj \in Conj(B_1) \cap Conj(B_2)} H(conj)$
4: return $\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2)$
```

Algorithm 1: Rule evolution given $\mathcal{RM}$ and $\mathcal{CF}$

The proofs for the correctness and complexity of Algorithm 1 can be found in our technical report [19].

PROPOSITION 2.7. *Algorithm 1 correctly returns a partition $P_o \in \bar{E}(P_i, B_2)$.*

PROPOSITION 2.8. *The complexity of Algorithm 1 is $O(c \times |S| + \frac{|S|^c}{z^c} \times g(\frac{|P_i| \times z^c}{|S|^c}, \frac{|S|}{|P_i|}))$ where S is the set of records in the input partition of records P_i, c is the number of common conjuncts between B_1 and B_2, z is the average cluster size for any partition produced by a conjunct, and $g(N, A)$ is the complexity of the ER algorithm E for an input partition containing N clusters with an average size of A records.*

While Algorithm 1 does not improve the complexity of the given ER algorithm E running without rule evolution, its runtime can be much faster in practice because the overhead for meeting partitions is not high (Step 3), and there can be large savings by running ER on small subsets of P_i (i.e., the $\{c' \in P_i | c' \subseteq c\}$'s) (Step 4) rather than on the entire partition P_i.

The rule evolution algorithm for ER algorithms that are only $\mathcal{RM}$ is identical to Algorithm 1 except for Step 4, where we can no longer process subsets of P_i independently. However, we can still run Step 4 efficiently using global information. We provide rule evolution techniques for the SN algorithm (which is $\mathcal{RM}$, but not $\mathcal{CF}$) in Appendix A.2.

Algorithm 2 performs rule evolution for ER algorithms that only satisfy the $\mathcal{GI}$ property. Algorithm 2 is identical to Algorithm 1 except that Step 4 is replaced with the code "**return** $E(\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2), B_2)$". Since the $\mathcal{RM}$ property is not satisfied anymore, we can no longer assume that the meet M is refined by the ER result of B_2. Hence, after each $\{c' \in P_i | c' \subseteq c\}$ is resolved, we need to run ER on the union of the results (i.e., the outermost ER operation in Step 4) to make sure we found all the matching records. The $\mathcal{GI}$ property guarantees that the output P_o is equivalent to a result in $\bar{E}(P_i, B_2)$. Using the same example for Algorithm 1, we now return $E(\{\{r_1\}\} \cup E(\{r_2, r_3\}, B_2) \cup \{\{r_4\}\}, B_2)$.

There are two factors that make Algorithm 2 efficient for certain ER algorithms. First, each cluster in M is common to several ER results and thus contains records that are likely to be clustered. An ER algorithm may run faster by resolving clusters that are likely to match first. Second, there are fewer clusters for the outer E operation to resolve compared to when E runs on the initial partition P_i. An ER algorithm may run faster when resolving fewer (but larger) clusters. While not all ER algorithms that are $\mathcal{GI}$ will speed up from these two factors, we will see in Section 4 that the HC_B algorithm indeed benefits from Algorithm 2.

The complexity of Algorithm 2 can be computed by adding the cost for meeting partitions and the cost for running ER on clusters. In comparison to Algorithm 1, the additional cost is the outermost ER operation in Step 4. In practice, Algorithm 2 is slower than Algorithm 1, but can still be faster than running the ER algorithm E without rule evolution.

The proof for the correctness of Algorithm 2 can be found in our technical report [19].

PROPOSITION 2.9. *Algorithm 2 correctly returns an ER result $P_o \in \bar{E}(P_i, B_2)$.*

3. DISTANCE-BASED EVOLUTION

We now consider rule evolution on distance-based clustering where records are clustered based on their relative distances instead of the Boolean match results used in the match-based clustering model. We first define our comparison rule as a distance function. We then define the notion of strictness between distance comparison rules and define properties analogous to those in Section 2.2. Finally, we provide a model on how the distance comparison rule can evolve and present our rule evolution techniques.

3.1 Distance-based Clustering Model

In the distance-based clustering model, records are clustered based on their relative distances with each other. The comparison rule is now defined as a commutative distance function D that returns a non-negative distance between two records instead of a Boolean function as in Section 2. For example, the distance between two person records may be the sum of the distances between their names, addresses, and phone numbers. The details on how exactly D is used for the clustering differs for each ER algorithm. In hierarchical clustering using distances [14], the closest pairs of records are merged first until a certain criterion is met. A more sophisticated approach [5] may cluster a set of records that are closer to each other compared to records outside, regardless of the absolute distance values. Other than using a distance comparison rule instead of a Boolean comparison rule, the definition of a valid ER algorithm remains the same as Definition 2.1.

In order to support rule evolution, we model D to return a *range* of possible non-negative distances instead of a single non-negative distance. For example, the distance $D(r_1, r_2)$ can be all possible distances within the range [13, 15]. We denote the minimum possible value of $D(r_1, r_2)$ as $D(r_1, r_2).min$ (in our example, 13) and the maximum value as $D(r_1, r_2).max$ (in our example, 15). As a result, an ER algorithm that only supports single-value distances must be extended to support ranges of values. The extension is specific to the given ER algorithm. However, in the case where the distance comparison rule only returns single value ranges, the extended algorithm must be identical to the original ER algorithm. Thus, the extension for general distances is only needed for rule evolution and does not change the behavior of the original ER algorithm.

A rule evolution occurs when a distance comparison rule D_1 is replaced by a new distance comparison rule D_2. We define the notion of relative strictness between distance comparison rules analogous to Definition 2.2.

DEFINITION 3.1. *A distance comparison rule D_1 is* stricter than *another rule D_2 (denoted as $D_1 \leq D_2$) if $\forall r, s$, $D_1(r, s).min \geq D_2(r, s).min$ and $D_1(r, s).max \leq D_2(r, s).max$.*

That is, D_1 is stricter than D_2 if its distance range is always within that of D_2 for any record pair. For example, if $D_2(r, s)$ is defined as all the possible distance values within $[D_1(r, s).min-1, D_1(r, s).max+1]$, then $D_1 \leq D_2$ (assuming $D_1(r, s).min \geq 1$).

3.2 Properties

We use properties analogous to $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$ from Section 2.2 for the distance-based clustering model. The only differences are that we now use distance comparison rules instead of Boolean comparison rules (hence we must replace all B's with D's) and Definition 3.1 instead of Definition 2.2 for comparing the strictness between distance comparison rules. To show how the properties hold in practice, we consider two distance-based clustering algorithms: HC_{DS} and HC_{DC} (see their definitions in Appendix B.1). Figure 3 shows that the HC_{DS} algorithm is $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$. As a result, the HC_{DS} algorithm can use Algorithm 1 (with minor changes; see Section 3.3) for rule evolution. On the other hand, the HC_{DC} algorithm is $\mathcal{CF}$ and $\mathcal{OI}$, but not $\mathcal{RM}$ or $\mathcal{GI}$. As a result, the extended HC_{DC} algorithm cannot use Algorithms 1 or 2 for rule evolution.

3.3 Rule Evolution

While we used the CNF structures of comparison rules to perform rule evolution in Section 2.4, the distance comparison rules are not Boolean expressions. Instead, we define a model on how the distance comparison rule can evolve. We assume that each distance $D_1(r, s)$ changes by at most $f(D_1(r, s))$ where f is a positive function that can be provided by a domain expert who knows how much D_1 can change. Examples of f include a constant value (i.e., each distance can change by at most some constant c) or a certain ratio of the original distance (i.e., each distance can change by at most X percent). As a result, $D_1(r, s).max + f(D_1(r, s)) \geq D_2(r, s).max$ and $D_1(r, s).min - f(D_1(r, s)) \leq D_2(r, s).min$. As a practical example, suppose that D_1 returns the sum of the distances for the names, addresses, and zip codes, and D_2 returns the sum of the distances for the names, addresses, and phone numbers. If we restrict the zip code and phone number distances to be at most 10, then when D_1 evolves to D_2, we can set f = 10. Or if the zip code and phone number distances are always within 20% of the D_1 distance, then $f = 0.2 \times D_1$.

Given D_1 and D_2, we can now define a third distance comparison rule $D_3(r, s) = [\max\{D_1(r, s).min - f(D_1(r, s)), 0\}, D_1(r, s).max + f(D_1(r, s))]$, which satisfies $D_3 \geq D_1$ and $D_3 \geq D_2$. (Notice that our definition ensures all the possible distances of D_3 to be non-negative.) Compared to the Boolean clustering model, rule D_3 acts as the "common conjuncts" between D_1 and D_2. As a result, we now materialize the ER result of D_3, $E(P_i, D_3)$, instead of the ER results for all the conjuncts in the first comparison rule. We also update Algorithm 1 in Section 2.4 by replacing Step 3 with "Partition $M \leftarrow H(D_3)$" where H is a hash table that only contains the result $E(P_i, D_3)$ for the comparison rule D_3. We illustrate rule evolution for the HC_{DS} algorithm using the updated Algorithm 1 in Appendix B.2.

4. EXPERIMENTAL EVALUATION

Evaluating rule evolution is challenging since the results depend on many factors including the ER algorithm, the comparison rules, and the materialization strategy. Obviously there are many cases where evolution and/or materialization are not effective, so our goal in this section is to show there are realistic cases where they can pay off, and that in some cases the savings over a naïve approach can be significant. (Of course, as the saying goes, "your mileage may vary"!) The savings can be very important in scenarios where data sets are large and where it is important to obtain a new ER result as quickly as possible (think of national security applications where it is critical to respond to new threats as quickly as possible).

For our evaluation, we assume that *blocking* [17] is used, as it is in most ER applications with massive data. With blocking, the input records are divided into separate blocks using one or more key fields. For instance, if we are resolving products, we can partition them by category (books, movies, electronics, etc). Then the records within one block are resolved independently from the other blocks. This approach lowers accuracy because records in separate blocks are not compared, but makes resolution feasible. (See [15, 20] for more sophisticated approaches). From our point of view, the use of blocking means that we can read a full block (which can still span many disk blocks) into memory, perform resolution (naïve or evolutionary), and then move on to the next block. In our experiments we thus evaluate the cost of resolving a single block. Keep in mind that these costs should be multiplied by the number of blocks.

There are three metrics that we use to compare ER strategies: CPU, IO and storage costs. (Except for Section 4.6, we do not consider accuracy since our evolution techniques do not change the ER result, only the cost of obtaining it.) We discuss CPU and storage costs in the rest of this section, leaving a discussion of IO costs to Appendix D.2. In general, CPU costs tend to be the most critical due to the quadratic nature of the ER problem, and because matching/distance rules tend to be expensive. In Appendix D.2 we argue that IO costs do not vary significantly with or without evolution and/or materialization, further justifying our focus here on CPU costs.

We start by describing our experimental setting in Section 4.1. Then in Sections 4.2 and 4.3, we discuss the CPU costs of ER evolution compared to a naïve approach (ignoring materialization costs, if any). In Section 4.4 we consider the CPU and space overhead of materializing partitions. Note that we do not discuss the orthogonal problem of *when* to materialize (a problem analogous to selecting what views to materialize). In Section 4.5 we briefly discuss total costs, including materialization and evolution.

4.1 Experimental Setting

We experiment on a comparison shopping dataset provided by Yahoo! Shopping and a hotel dataset provided by Yahoo! Travel. Table 1 summarizes the comparison rules used in our experiments. We evaluated the following ER algorithms: SN, HC_B, HC_{BR}, ME, HC_{DS}, and HC_{DC}. Details on the datasets, comparison rules, and which rule evolution algorithm was used for which ER algorithm can be found in Appendix D.1. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4GB of RAM.

Table 1: Comparison Rules

Type	Data	Comparison rules
Boolean	Shopping	$B_1^S : p_{ti} \wedge p_{ca}$ $B_2^S : p_{ti} \wedge p_{pr}$
Boolean	Hotel	$B_1^H : p_{st} \wedge p_{ci} \wedge p_{zi} \wedge p_{na}$ $B_2^H : p_{st} \wedge p_{ci} \wedge p_{zi} \wedge p_{sa}$
Distance	Shopping	$D_1^S : Jaro_{ti}$ $D_2^S : Jaro_{ti}$ changes randomly within 5%
Distance	Hotel	$D_1^H : Jaro_{na} + 0.05 \times Equals_{ci}$ $D_2^H : Jaro_{na} + 0.05 \times Equals_{zi}$

4.2 Rule Evolution Efficiency

We first focus on the CPU time cost of rule evolution (exclusive of materialization costs, if any) using blocks of data that fit in memory. For each ER algorithm, we use the best evaluation scheme (see Appendix D.1) given the properties of the ER algorithm. Table 2 shows the results. We run the ER algorithms SN, HC_B, and HC_{BR} using the Boolean comparison rules in Table 1 on the shopping and hotel datasets. When evaluating each comparison rule, the conjuncts involving string comparisons (i.e., p_{ti}, p_{na}, and p_{sa}) are evaluated last because they are more expensive than the rest of the conjuncts. We also run the HC_{DS} algorithm using the distance comparison rules in Table 1 on the two datasets. Each column head in Table 2 encodes the dataset used and the number of records resolved in the block. For example, Sh1K means 1,000 shopping records while Ho3K means 3,000 hotel records. The top five rows of data show the runtime results of the naïve approach while the bottom five rows show the runtime improvements of rule evolution compared to the naïve approach. Each runtime improvement is computed by dividing the naïve approach runtime by the rule evolution runtime. For example, the HC_{BR} algorithm takes 3.56 seconds to run on 1K shopping records and rule evolution is 162 times faster (i.e., having a runtime of $\frac{3.56}{162}$ = 0.022 seconds).

Table 2: ER algorithm and rule evolution runtimes

ER algorithm	Sh1K	Sh2K	Sh3K	Ho1K	Ho2K	Ho3K
ER algorithm runtime (seconds)						
SN	0.094	0.152	0.249	0.012	0.027	0.042
HC_B	1.85	7.59	17.43	0.386	2.317	5.933
HC_{BR}	3.56	19.37	48.72	0.322	1.632	4.264
HC_{DS}	8.33	40.38	111	5.482	27.96	73.59
Ratio of ER algorithm runtime to rule evolution runtime						
SN	4.09	4.22	4.45	1.2	1.93	2
HC_B	1.5	1.84	2.07	1.27	1.3	1.27
HC_{BR}	162	807	1218	36	136	237
HC_{DS}	298	708	918	322	499	545

As one can see in Table 2, the improvements vary widely but in many cases can be very significant. For the shopping dataset, the HC_{BR}, and HC_{DS} algorithms show up to orders of magnitude of runtime improvements. The SN algorithm has a smaller speedup because SN itself runs efficiently. The HC_B algorithm has the least speedup (although still a speedup). While the rule evolution algorithms for SN, HC_{BR}, and HC_{DS} only need to resolve few clusters at a time (i.e., each $\{c' \in P_i | c' \subseteq c\}$ in Algorithm 1), Algorithm 2 for the HC_B algorithm also needs to run an outermost ER operation (Step 4) to resolve the clusters produced by the inner ER operations. The hotel data results show worse runtime improvements overall because the ER algorithms without rule evolution ran efficiently.

4.3 Common Rule Strictness

The key factor of the runtime savings in Section 4.2 is the strictness of the "common comparison rule" between the old and new comparison rules. For match-based clustering, the common comparison rule between B_1 and B_2 comprises the common conjuncts $Conj(B_1) \cap Conj(B_2)$. For distance-based clustering, the common comparison rule between D_1 and D_2 is D_3, as defined in Section 3.3. A stricter rule is more selective (fewer records match or fewer records are within the threshold), and leads to smaller clusters in a resolved result. If the common comparison rule yields smaller clusters, then in many cases the resolution that starts from there will have less work to do.

By changing the thresholds used by the various predicates, we can experiment with different common rule strictness, and Figure 4 summarizes some of our findings. The horizontal axis shows the strictness of the common rule: it gives the ratio of record pairs placed by the common rule within in a cluster to the total number of record pairs. For example, if an ER algorithm uses p_{ti} to produce 10 clusters of size 10, then the strictness is $\frac{10\times\binom{10}{2}}{\binom{100}{2}}$ = 0.09. The lower the ratio is, the stricter the common rule, and presumably, fewer records need to be resolved using the new comparison rule.

The vertical axis in Figure 4 shows the runtime improvement (vs. naïve), for four algorithms using our shopping data comparison rules in Table 1. The runtime improvement is computed as the runtime of the naïve approach computing the new ER result divided by the runtime of rule evolution. As expected, Algorithms SN, HC_{BR}, and HC_{DS} achieve significantly higher runtime improvements as the common comparison rule becomes stricter. However, the HC_B algorithm shows a counterintuitive trend (performance decreases as strictness increases). In this case there are two competing factors. On one hand, having a stricter common comparison rule improves runtime for rule evolution because the computation of each $E(\{c' \in P_i | c' \subseteq c\}, B_2)$ in Step 4 becomes more efficient. On the other hand, a common comparison rule that is too strict produces many clusters to resolve for the outermost ER operation in Step 4, increasing the overall runtime. Hence, although not shown in the plot, the increasing line will eventually start decreasing as strictness decreases.

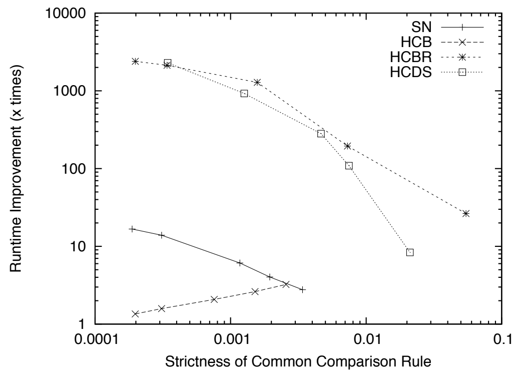

Figure 4: Degree of change impact on runtime, 3K shopping records

4.4 Materialization Overhead

In this section we examine the CPU and space overhead of materializations, independent of the question of what conjuncts should be materialized. Recall that materializations are done as we perform the initial resolution on records S. Thus the materialization can piggyback on the ER work that needs to be done anyway. For example, the parsing and initialization of records can be done once for the entire process of creating all materializations and running ER for the old comparison rule. In addition, there are other ways to amortize work, as the resolution is concurrently done for the old rule and the conjuncts we want to materialize (more details can be found in our technical report [19]). We can also compress the storage space needed by materializations by storing partitions of record IDs.

Table 3: Time overhead (ratio to old ER algorithm runtime) and space overhead (ratio to old ER result) of rule materialization, 3K records

ER algorithm	Sho3K		Ho3K	
	Time O/H	Space O/H	Time O/H	Space O/H
SN	0.52 (0.02)	0.28	1.14 (0.27)	0.14
HC_B	0.87 (0.04)	0.14	3.18 (0.71)	0.1
HC_{BR}	11 (3E-6)	0.14	13.28 (1.06)	0.1
HC_{DS}	0.44	0.07	0.61	0.02

Table 3 shows the time and space overhead of materialization in several representative scenarios. In particular, we use Algorithms SN, HC_B, HC_{BR}, and HC_{DS} on 3K shopping and hotel records, and assume *all* conjuncts in the old rule are materialized.

The *Time O/H* columns show the time overhead where each number is produced by dividing the materialization CPU time by the CPU runtime for producing the old ER result. For example, materialization time for the SN algorithm on 3K shopping records is 0.52x the time for running $E(P_i,B_1^S)$ using SN. Hence, the total time to compute $E(P_i,\ B_1^S)$ and materialize all the conjuncts of B_1^S is 1+0.52 = 1.52 times the runtime for $E(P_i,\ B_1^S)$ only. The numbers in parentheses show the time overhead when we do *not* materialize the most expensive conjunct. That is, for SN, HC_B, and HC_{BR} in the shopping column we only materialize p_{ca}; in the hotel column, we only materialize p_{st}, p_{ci}, and p_{zi} (without p_{na}).

For the shopping dataset, the SN and HC_B algorithms have time overheads less than 2 (i.e., the number of conjuncts in B_1^S) due to amortization. For the same reason, HC_{DS} has a time overhead below 1. The HC_{BR} algorithm has a large overhead of 11x because each common conjunct tends to produce larger clusters compared to $E(P_i,\ B_1^H)$, and HC_{BR} ran slowly when larger clusters were compared using the expensive p_{ti} conjunct.

The hotel dataset shows similar time overhead results, except that the time overheads usually do not exceed 4 (i.e., the number of conjuncts in B_1^H) for the match-based clustering algorithms.

The *Space O/H* columns show the space overhead of materialization where each number was produced by dividing the memory space needed for storing the materialization by the memory space needed for storing the old ER result. For example, the materialization space for the SN algorithm on 3K shopping records is 0.28x the memory space taken by $E(P_i,\ B_1^S)$ using SN. The total required space is thus 1+0.28 = 1.28 times the memory space needed for $E(P_i,\ B_1^S)$. The space overhead of materialization is small in general because we only store records by their IDs.

4.5 Total Runtime

The speedups achievable at evolution time must be balanced against the cost of materializations during earlier resolutions. The materialization cost of course depends on what is materialized: If we do not materialize any conjuncts, as in our initial example in Section 1, then clearly there is no overhead. At the other extreme, if the initial rule B_1 has many conjuncts and we materialize all of them, the materialization cost will be higher. If we have application knowledge and know what conjuncts are "stable" and likely to be used in future rules, then we can only materialize those. Then there is also the amortization factor: if a materialization can be used many times (e.g., if we want to explore many new rules that share the materialized conjunct), then the materialization cost, even if high, can be amortized over all the future resolutions.

In Appendix D.3 we study the total run time (CPU and IO time for original resolution plus materializations plus evolution) for several scenarios. We experiment on 0.25 to 1 million shopping records (multiple blocks are processed). Our results illustrate scenarios where materialization does pay off. That is, materialization and evolution lowers the total time, as compared to the naïve approach that runs ER from scratch each time. Of course, one can also construct scenarios where materialization does not pay off.

4.6 Without the Properties

In Appendix D.4 we consider scenarios where the necessary properties for rule evolution do not hold. In such a case, we need to use the naïve approach to get a correct answer. From our previous results, however, we know that the naïve approach can be very expensive compared to rule evolution. The alternatives are to fix the ER algorithm to satisfy one of the properties or to run one of our rule evolution algorithms even though we will not get correct answers. We have investigated the latter case for the ME and HC_{DS} algorithms and observe that we can still return ER results with small losses in accuracy, but with large benefits in rule evolution runtime.

5. CONCLUSION

In most ER scenarios, the logic for resolving records evolves over time, as the application itself evolves and as the expertise for comparing records improves. In this paper we have explored a fundamental question: when and how can we base a resolution on a previous result as opposed to starting from scratch? We have answered this question in two commonly-used contexts, record comparisons based on Boolean predicates and record comparisons based on distance (or similarity) functions. We identified two properties of ER algorithms, rule monotonic and context free (in addition to order independence and general incremental), that can significantly reduce runtime at evolution time. We also categorized several popular ER algorithms according to the four properties.

In some cases, computing an ER result with a new rule can be much faster if certain partial results are materialized when the original ER result (with the old rule) is computed. We studied how to take advantage of such materializations, and how they could be computed efficiently by piggybacking the work on the original ER computation.

Our experimental results evaluated the cost of both materializations and the evolution itself (computing the new ER result), as compared to a naïve approach that computed the new result from scratch. We considered a variety of popular ER algorithms (each having different properties), two data sets, and different predicate strictness. The results illustrate realistic cases where materialization costs are relatively low, and evolution can be done extremely quickly.

Overall, we believe our analysis and experiments provides guidance for the ER algorithm designer. The experimental results show the potential gains, and if these gains are attractive in an application scenario, our properties help us design algorithms that can achieve such gains.

6. ACKNOWLEDGEMENTS

We acknowledge Jeff Ullman for his helpful comments and David Marmaros for his help in implementation.

7. REFERENCES

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[2] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.

[3] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *DMKD*, 2004.

[4] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *STOC*, pages 626–635, 1997.

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

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[7] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[8] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Applied Statistics*, 18(1):54–64, 1969.

[9] L. M. Haas, M. Hentschel, D. Kossmann, and R. J. Miller. Schema and data: A holistic approach to mapping, resolution and fusion in information integration. In *ER*, pages 27–40, 2009.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

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

[12] P. Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[13] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[14] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[15] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of KDD*, pages 169–178, Boston, MA, 2000.

[16] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DMKD*, pages 23–29, 1997.

[17] H. B. Newcombe and J. M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962.

[18] S. E. Whang, O. Benjelloun, and H. Garcia-Molina. Generic entity resolution with negative rules. *VLDB J.*, 18(6):1261–1277, 2009.

[19] S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. Technical report, Stanford University, available at http://ilpubs.stanford.edu:8090/961/.

[20] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, pages 219–232, 2009.

[21] W. Winkler. Overview of record linkage and current research directions. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 2006.

APPENDIX

A. MATCH-BASED EVOLUTION

A.1 ER Algorithms

In the main body of the paper, we refer to a variety of well known match-based clustering ER algorithms. In this section we briefly describe the most important ones for our paper: SN, HC_B, HC_{BR}, and ME. While the original definitions of all four ER algorithms assume a set of records S as an input, we provide simple extensions for the algorithms to accept a set of clusters P_i as in Definition 2.1.

SN. The sorted neighborhood (SN) algorithm [11] first sorts the records in P_i (i.e., we extract all the records from the clusters in P_i) using a certain key assuming that closer records in the sorted list are more likely to match. For example, suppose that we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and sort the clusters by their names (which are not visible in this example) in alphabetical order to obtain the list $[r_1, r_2, r_3]$. The SN algorithm then slides a fixed-sized window on the sorted list of records and compares all the pairs of clusters that are inside the same window at any point. If the window size is 2 in our example, then we compare r_1 with r_2 and then r_2 with r_3, but not r_1 with r_3 because they are never in the same window. We thus produce pairs of records that match with each other. We can repeat this process using different keys (e.g., we could also sort the person records by their address values). After collecting all the pairs of records that match, we perform a transitive closure on all the matching pairs of records to produce a partition P_o of records. For example, if r_1 matches with r_2 and r_2 matches with r_3, then we merge r_1, r_2, r_3 together into the output $P_o = \{\{r_1, r_2, r_3\}\}$.

HC_B. Hierarchical clustering based on a Boolean comparison rule [2] (which we call HC_B) combines matching pairs of clusters in any order until no clusters match with each other. The comparison of two clusters can be done using an arbitrary function that receives two clusters and returns `true` or `false`, using the boolean comparison rule B to compare pairs of records. For example, suppose we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the comparison rule B where $B(r_1, r_2)$ = `true`, $B(r_2, r_3)$ = `true`, but $B(r_1, r_3)$ = `false`. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs (e.g., a record r_2 has an ID of 2) from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Depending on the order of clusters compared, the HC_B algorithm can merge $\{r_1\}$ and $\{r_2\}$ first, or $\{r_2\}$ and $\{r_3\}$ first. In the first case, the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$ (because the clusters $\{r_1, r_2\}$ and $\{r_3\}$ do not match) while in the second case, the ER result is $\{\{r_1\}, \{r_2, r_3\}\}$ (the clusters $\{r_1\}$ and $\{r_2, r_3\}$ do not match). Hence, $\bar{E}(P_i, B) = \{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$.

HC_{BR}. The HC_B algorithm both $\mathcal{RM}$ and $\mathcal{CF}$ if two clusters are guaranteed to match whenever at least one of their records match according to B. (This property is equivalent to the representativity property in reference [2].) For example, a cluster comparison function that compares all the records between two clusters using B for an existential match is representative. That is, given two clusters $\{r_1, r_2\}$ and $\{r_3, r_4\}$, the cluster comparison function returns `true` if at least one of $B(r_1, r_3)$, $B(r_1, r_4)$, $B(r_2, r_3)$, or $B(r_2, r_4)$ returns `true`. We denote the HC_B algorithm where the cluster comparison function satisfies representativity as HC_{BR}. We can prove that the HC_{BR} algorithm always returns a unique solution for any input P_i [2].

ME. The Monge Elkan (ME) clustering algorithm (we define a variant of the algorithm in [16] for simplicity) first sorts the records in P_i (i.e., we extract all the records from the clusters in P_i) by some key and then starts to scan each record. For example, suppose that we are given the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and we sort the records in P_i by their names (which are not visible in this example) in alphabetical order into the sorted list of records $[r_1, r_2, r_3]$. Suppose we are also given the Boolean comparison rule B where $B(r_1, r_2)$ = `true`, but $B(r_1, r_3)$ = `false` and $B(r_2, r_3)$ = `false`. Each scanned record is then compared with clusters in a fixed-length queue. A record r matches with a cluster c if $B(r, s)$ = `true` for any $s \in c$. If the new record matches one of the clusters, the record and cluster merge, and the new cluster is promoted to the head of the queue. Otherwise, the new record forms a new singleton cluster and is pushed into the head of the queue. If the queue is full, the last cluster in the queue is dropped. In our example, if the queue size is 1, then we first add r_1 into the head of the queue, and then compare r_2 with $\{r_1\}$. Since r_2 matches with $\{r_1\}$, we merge r_2 into $\{r_1\}$. We now compare r_3 with the cluster $\{r_1, r_2\}$ in the queue. Since r_3 does not match with $\{r_1, r_2\}$, then we insert $\{r_3\}$ into the head of the queue and thus remove $\{r_1, r_2\}$. Hence, the only possible ER result is $\{\{r_1, r_2\}, \{r_3\}\}$ and thus $\bar{E}(P_i, B) = \{\{\{r_1, r_2\}, \{r_3\}\}\}$. In general, ME always returns a unique partition.

A.2 Rule Evolution for ER algorithms that are $\mathcal{RM}$ only

In the main body of the paper, we discuss rule evolution algorithms for ER algorithms that satisfy certain properties. In this section, we cover rule evolution for ER algorithms that are only $\mathcal{RM}$. The rule evolution algorithm is identical to Algorithm 1 except for Step 4, where we can no longer process subsets of P_i independently. However, we can still run Step 4 efficiently using global information. We revisit the sorted neighborhood ER algorithm (SN) in Section 2.2.4. Recall that the first step of SN is to move a sliding window on a sorted list of records, comparing records pairwise only within the same window of size W. (The second step is a transitive closure of all matching pairs.) In Step 4, we are able to resolve each $\{c' \in P_i | c' \subseteq c\}$ ($c \in M$) using the same window size W as long as we also use the global sort information of the records to make sure only the records that would have been in the same window during the original run of SN should be compared with each other. Suppose that we have $B_1 = p_{name} \wedge p_{zip}$, $B_2 = p_{name} \wedge p_{phone}$, and the initial set $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_5\}\}$. We set the sort key to be the record ID (e.g., r_4 has the ID 4). As a result, the records are sorted into the list $[r_1, r_2, r_3, r_4, r_5]$. Using a window size of W= 3, suppose we materialize $E(P_i, p_{name}) = \{\{r_1, r_3, r_5\}, \{r_2\}, \{r_4\}\}$ because r_1 and r_3 matched when the window covered $[r_1, r_2, r_3]$ and r_3 and r_5 matched when the window covered $[r_3, r_4, r_5]$. The records r_1 and r_5 only match during the transitive closure in the second step of SN. The meet M in Algorithm 1 is also $\{\{r_1, r_3, r_5\}, \{r_2\}, \{r_4\}\}$ because there is only one common conjunct p_{name} between B_1 and B_2. Thus, we only need to resolve the set $\{r_1, r_3, r_5\}$ using B_2. However, we must be careful and should not simply run $E(\{r_1, r_3, r_5\}, B_2)$ using a sliding window of size 3. Instead, we must take into account the global ordering information and never compare r_1 and r_5, which were never in the same window. Thus, if $B_2(r_1, r_3)$ = `false`, $B_2(r_3, r_5)$ = `false`, and $B_2(r_1, r_5)$ = `true`, the correct ER result is that none of r_1, r_3, r_5 are clustered. While we need to use the global sort information of records, our rule evolution is still more efficient than re-running SN on the entire input P_i (see Section 4).

B. DISTANCE-BASED EVOLUTION

B.1 ER Algorithms

In the main body of the paper, we refer to two well known distance-based clustering ER algorithms. In this section, we briefly describe the two algorithms: HC_{DS} and HC_{DC}.

HC_{DS}. The Single-link Hierarchical Clustering algorithm [8, 14] (HC_{DS}) merges the closest pair of clusters (i.e., the two clusters that have the smallest distance) into a single cluster until the smallest distance among all pairs of clusters exceeds a certain threshold T. When measuring the distance between two clusters, the algorithm takes the smallest possible distance between records within the two clusters. Suppose we have the input partition P_i = $\{\{r_1\}, \{r_2\}, \{r_3\}\}$ where $D(r_1, r_2)$ = 2, $D(r_2, r_3)$ = 4, and $D(r_1, r_3)$ = 5 (we later extend HC_{DS} to support ranges of distances) with T = 2. The HC_{DS} algorithm first merges r_1 and r_2, which are the closest records and have a distance smaller or equal to T, into $\{r_1, r_2\}$. The cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the minimum of $D(r_1, r_3)$ and $D(r_2, r_3)$, which is 4. Since the distance exceeds T, $\{r_1, r_2\}$ and $\{r_3\}$ do not merge, and the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$.

We extend the HC_{DS} algorithm by allowing ranges of distances to be returned by a distance comparison rule, but only comparing the minimum value of a range with either another range or the threshold T. That is, $D(r, s)$ is considered a smaller distance than $D(u, v)$ if $D(r, s).min \leq D(u, v).min$. Also, $D(r, s)$ is considered smaller than T if $D(r, s).min \leq T$. For example, [3, 5] $<$ [4, 4] because 3 is smaller than 4, and [3, 5] $> T$ = 2 because 3 is larger than 2. The extended HC_{DS} algorithm is trivially identical to the original HC_{DS} algorithm when D only returns a single value.

HC_{DC}. The Complete-link Hierarchical Clustering (HC_{DC}) algorithm [14] is identical to the HC_{DS} algorithm except in how it measures the distance between two clusters. While the HC_{DS} algorithm chooses the smallest possible distance between records within the two clusters, the HC_{DC} algorithm takes the largest possible distance instead. For example, the cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the maximum of $D(r_1, r_3)$ and $D(r_2, r_3)$. We use the same extension used in HC_{DS} to support ranges of values for distances where only the minimum values of each range are compared to other ranges or thresholds.

B.2 Rule Evolution for the HC_{DS} Algorithm

In the main body of the paper, we propose rule evolution techniques for distance-based clustering ER algorithms. In this section, we illustrate rule evolution for the HC_{DS} algorithm using the updated Algorithm 1. Suppose we are given the input partition P_i = $\{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the distance comparison rule D_1 where $D_1(r_1, r_2)$ = [2], $D_1(r_2, r_3)$ = [4], and $D_1(r_1, r_3)$ = [5]. We use the threshold T = 2 for termination. If we are given $f(d)$ = 0.1 $\times$ d, D_3 is defined as $D_3(r_1, r_2)$ = [1.8, 2.2], $D_3(r_2, r_3)$ = [3.6, 4.4], and $D_3(r_1, r_3)$ = [4.5, 5.5]. We then materialize the ER result M = $E(P_i, D_3)$. Among the records, only r_1 and r_2 match having $D_3(r_1, r_2).min$ = 1.8 $\leq T$ = 2. Once the clusters $\{r_1\}$ and $\{r_2\}$ merge, $\{r_1, r_2\}$ and $\{r_3\}$ do not match because $D_3(r_1, r_3).min$ = 4.5 and $D_3(r_2, r_3).min$ = 3.6, both exceeding T. Hence M = $\{\{r_1, r_2\}, \{r_3\}\}$. Suppose we are then given D_2 such that $D_2(r_1, r_2)$ = [2.2], $D_2(r_2, r_3)$ = [3.9], and $D_2(r_1, r_3)$ = [4.9] (notice that indeed $D_2 \leq D_3$). We then return $\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, D_2)$ using the same threshold T = 2. For the first cluster in M, we run $E(\{\{r_1\}, \{r_2\}\}, D_2)$. Since $D_2(r_1, r_2).min$ = 2.2 $> T$, $\{r_1\}$ and $\{r_2\}$ do not merge. The next partition $\{\{r_3\}\}$ is a singleton, so our new ER result is $\{\{r_1\}, \{r_2\}, \{r_3\}\}$, which is identical to $E(P_i, D_2)$.

C. MATERIALIZATION STRATEGIES

In the main body of the paper, we described one materialization strategy where the ER result of each conjunct of a Boolean comparison rule is computed and stored. In this section, we list possible optimizations for materializations given more application-specific knowledge. Our list is by no means exhaustive, and the possible optimizations will depend on the ER algorithm and comparison rules.

A group of conjuncts is "stable" if they appear together in most comparison rules. As a result, the group can be materialized instead of all individual conjuncts. For example, if the conjuncts p_1, p_2, and p_3 are always compared as a conjunction in a person records comparison rule, then we can materialize on $p_1 \wedge p_2 \wedge p_3$ together rather than on the three conjuncts separately. Hence, the time and space overhead of materialization can be saved.

If we know the pattern of how the comparison rule will evolve, we can also avoid materializing on all conjuncts. In the ideal case where we know that the comparison rule can only get stricter, we do not have to save any additional materializations other than the ER result of the old comparison rule. Another scenario is when we are only changing the postfix of the old comparison rule, so we only need to materialize on all the prefixes of the old comparison rule. For example, if we have the comparison rule $p_1 \wedge p_2 \wedge p_3$, then we can materialize on p_1, $p_1 \wedge p_2$, and $p_1 \wedge p_2 \wedge p_3$. If the ER algorithm is both $\mathcal{RM}$ and $\mathcal{CF}$, then the ER result of $p_1 \wedge p_2$ can be computed efficiently from the ER result of p_1, and the ER result of $p_1 \wedge p_2 \wedge p_3$ from that of $p_1 \wedge p_2$.

D. EXPERIMENTAL EVALUATION

D.1 Experimental Setting

In the main body of the paper, we show experimental results for rule evolution. In this section, we describe the experimental settings used for our experiments.

Real Data. The comparison shopping dataset we use was provided by Yahoo! Shopping and contains millions of records that arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains various attributes including the title, price, and category of an item. We experimented on a random subset of 3,000 shopping records that had the string "iPod" in their titles and a random subset of 1 million shopping records. We also experimented on a hotel dataset provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. We experimented on a random subset of 3,000 hotel records located in the United States. While the 3K shopping and hotel datasets fit in memory, the 1 million shopping dataset did not fit in memory and had to be stored on disk.

Comparison Rules. Table 1 summarizes the comparison rules used in our experiments. The *Type* column indicates whether the comparison rules are Boolean comparison rules or distance comparison rules. The *Data* column indicates the data source: shopping or hotel data. The *Comparison rules* column indicates the comparison rules used. The first two rows define the Boolean comparison rules used on the shopping and hotel datasets. For the shopping datasets, B_1^S compares the titles and categories of two shopping records while B_2^S compares the titles and prices of shopping records. For the hotel data, B_1^H compares the states, cities, zip codes, and names of two hotel records. The B_2^H rule compares the states, cities, zip codes, and street addresses of two hotel records. The last two rows define the distance comparison rules for the two datasets. For the shopping data, D_1^S measures the Jaro distance [21] between the titles of two shopping records while D_2^S randomly alters the distance of D_1^S by a maximum ratio of 5%. The Jaro distance returns a value within the range [0, 1], and gives higher values for closer records. For the hotel data, D_1^H sums the Jaro distance between the names of two records and the Equality distance between the cities of two records weighted by 0.05. We define the Equality distance to return 1 if two values are exactly the same and 0 if they are not the same. The D_2^H rule sums the Jaro distance between names with the Equality distance between the zip codes of two records weighted by 0.05. As a result, the D_1^H distance can alter by at most the constant 0.05.

ER and Rule Evolution Algorithms. We experiment rule evolution on the following ER algorithms: SN, HC_B, HC_{BR}, ME, HC_{DS}, and HC_{DC}. Table 4 summarizes for each ER algorithm which section it was defined in and which rule evolution algorithm is used. The HC_{DS} and HC_{DC} distanced-based clustering algorithms terminate when the minimum distance between clusters is smaller than the threshold 0.95 (recall that *closer* records have *higher* Jaro+Equality distances). Although the ME and HC_{DC} algorithms do not satisfy the $\mathcal{RM}$ property, we can still use Algorithm 1 to efficiently produce new ER results with small loss in accuracy. Notice that, although ME is $\mathcal{GI}$, Algorithm 2 is not efficient because of the way ME extracts all records from the input partition P_i (without exploiting any of the clusters in P_i) and sorts them again. Both the HC_{DS} and HC_{DC} algorithms use Algorithm 1 adjusted for the distance-based clustering model (see Section 3.3).

Table 4: ER and rule evolution algorithms tested

ER algorithm	Section	Rule evolution algorithm used
SN	2.4	Algorithm for SN in Appendix A.2
HC_B	2.4	Algorithm 2
HC_{BR}	2.4	Algorithm 1
ME	2.4	Algorithm 1
HC_{DS}	3.2	Algorithm 1 (for distance-based clustering)
HC_{DC}	3.2	Algorithm 1 (for distance-based clustering)

D.2 Evaluating IO costs

In the main body of the paper, we focused on the CPU costs for materializing ER results, and for evolving a prior result under new logic. In this section, we discuss the corresponding IO costs and argue that the materialization IO costs are less significant than the CPU costs. Using our blocking framework, we can analyze the overall runtime of an ER process. The basic operations of an ER process are described in Table 5. The operations are categorized depending on whether they are disk IO consuming operations or CPU time consuming operations.

To compare the overall performance of an ER process using rule evolution and a naïve ER process without rule evolution, we consider the scenario where we run ER once using an old comparison rule and then perform one rule evolution using a new comparison rule. A naïve ER process without rule evolution would roughly require initializing the records, creating the blocks, and reading and resolving the blocks twice. An ER process using rule evolution on the other hand would require the same process above plus the additional work of creating and using rule materializations minus running ER on all blocks during the rule evolution. The decompositions of the two approaches for our one rule evolution scenario are shown in Table 6. Notice that the listed operations are not necessarily run sequentially. For example, for the naïve approach, the R_B and E operations are actually interleaved because each block is read and then resolved before the next block is read.

Table 5: Basic operations in blocking ER framework

Operation	Description
IO time consuming operations	
R_F	Read records from input file
R_B	Read all blocks to memory
W_B	Write out all blocks to disk
R_M	Read all materializations to memory
W_M	Write all materializations to disk
O	Write the output ER result to disk
CPU time consuming operations	
I	Initialize records (trim attributes not used in rules)
E	Run ER on all blocks (one block at a time)
M	Create materializations for all blocks (one at a time)
V	Run rule evolution (using materializations) on all blocks (one at a time)

Table 6: Decomposition of ER processes for one rule evolution

ER process	Decomposition
Naïve	R_F,I,W_B,R_B,E,O,R_B,E,O
Using rule evolution	R_F,I,W_B,R_B,E,O,M,W_M,R_B,R_M,V,O

The IO overhead of using rule evolution compared to the IO cost of the naïve approach can thus be written as $\frac{R_M+W_M}{R_F+W_B+2\times R_B+2\times O}$. Since the size of the materializations is usually much smaller than the size of the entire set of records (see Section 4.4), the additional IOs for rule evolution is also smaller than the IOs for reading and writing the blocks. Thus, the IO costs do not vary significantly with or without evolution and/or materialization.

D.3 Total Runtime

In the main body of the paper, we claim that the runtime benefits of rule evolution can exceed the time overhead that is payed. In this section, we measure the total runtimes of ER processes as defined in Appendix D.2 where we run ER once using an old comparison rule and then perform one rule evolution using a new comparison rule. We experimented on 0.25 to 1 million random shopping records and used the following Boolean comparison rules for the SN, HC_B, and HC_{BR} algorithms: $B_1 = p_{ca} \wedge p_{ti}$ (same as B_1^S in Table 1) and $B_2 = p_{ca} \wedge p_{pr}$. In addition, we only materialized on the conjunct p_{ca} instead of on both conjuncts in B_1. The time overheads for materializing p_{ca} were shown in parentheses in Figure 3. For the HC_{DS} algorithm, we used D_1^S and D_2^S in Table 1. We used minhash signatures [12] for distributing the records into blocks. For the shopping dataset, we extracted 3-grams from the titles of records. We then generated a minhash signature for each records, which is an array of integers where each integer is generated by applying a random hash function to the 3-gram set of the record.

Figure 5 shows our total time results where we measured the total runtimes of running ER on B_1 and then evolving once to B_2. Each rule evolution technique and its corresponding naïve approach use the same shape for points in their plots. For example, the rule evolution runtime plot for the SN algorithm uses *white square* points while the naïve SN approach uses *black square* points. In addition, all the naïve approach plots use white shapes while the rule evolution plots use black shapes. Our results show that the total runtimes for the SN and HC_B algorithms do not change much because the runtime benefits of using rule evolution more or less cancels out the runtime overheads of using rule evolution. For the HC_{BR} and HC_{DS} algorithms, however, the runtime benefits of rule evolution clearly exceed the overheads. While we have shown the worst case scenario results where only one evolution occurs, the improvements will most likely increase for multiple rule evolutions using the same materializations.

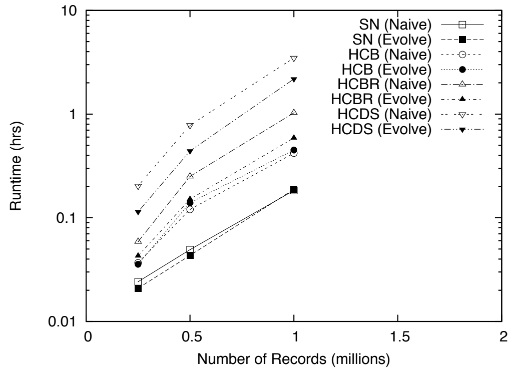

Figure 5: Scalability, 1M shopping records

D.4 Without the Properties

In the main body of the paper, we claim that rule evolution can also benefit ER algorithms that do not satisfy the necessary properties. In this section, we experiment on two ER algorithms that do not satisfy the $\mathcal{RM}$ property and thus cannot use Algorithm 1: the ME and HC_{DC} algorithms. While the ME algorithm is still $\mathcal{GI}$ and can thus use Algorithm 2, there is no runtime benefit because in ME all the records in P_i are extracted and sorted again regardless of the clusters in P_i (see Appendix A.1).

To measure accuracy, we compare a rule evolution algorithm result with the corresponding result of the naïve approach. We consider all the records that merged into an output cluster to be identical to each other. For instance, if the clusters $\{r\}$ and $\{s\}$ merged into $\{r, s\}$ and then merged with $\{t\}$ into $\{r, s, t\}$, all three records r, s, t are considered to be the same. Suppose that the correct answer A contains the set of record pairs that match for the naïve solution while set B contains the matching pairs for the rule evolution algorithm. Then the precision Pr is $\frac{|A \cap B|}{|B|}$ while the recall Re is $\frac{|A \cap B|}{|A|}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy metric.

Table 7 shows the runtime and accuracy results of running Algorithm 1 as the rule evolution algorithm on datasets that fit in memory. The columns show the dataset used and the number of records resolved. The top two rows of data show the runtimes for the naïve approach. The middle two rows of data show the runtime improvements of rule evolution compared to the naïve approaches. Each runtime improvement is computed by dividing the naïve approach runtime by the rule evolution runtime (not including the materialization costs). Overall, the runtime of ME improves by 1.67x to 5.53x while the runtime of HC_{DC} improves by 501x to 2386x. The bottom two rows of data show the accuracy values of each ER result compared to the correct result produced by the naïve aproach. The accuracy results are near-perfect for the ME algorithm while being at least 0.85 for HC_{DC}. The experiments show that rule evolution may produce highly-accurate ER results even if the ER algorithms do not satisfy any property while still significantly enhancing the runtime performance of rule evolution.

E. RELATED WORK

Entity resolution has been studied under various names including record linkage, merge/purge, deduplication, reference reconciliation, object identification, and others (see [7] for a recent survey). Entity resolution involves comparing records and determining if they refer to the same entity or not. Most of the works fall into one of the ER models we consider: match-based clustering [11, 2] and distance-based clustering [3, 14]. (In our technical report [19], we also consider a more primitive model where we return the matching pairs of records instead of a clustering of records.) While the ER literature focuses on improving the accuracy or runtime performance of ER, they usually assume a fixed logic for resolving records. To the best of our knowledge, our work is the first to consider the ER result update problem when the logic for resolution itself changes.

Table 7: Runtime and accuracy results for ER algorithms without the properties

ER algorithm	Sh1K	Sh2K	Sh3K	Ho1K	Ho2K	Ho3K
ER algorithm runtime (seconds)						
ME	0.094	0.162	0.25	0.015	0.033	0.051
HC_{DC}	8.08	39.2	105	5.51	28.1	73.57
Ratio of ER algorithm runtime to rule evolution time						
ME	5.53	5.23	5.43	1.67	2.06	2.04
HC_{DC}	674	1509	2386	501	879	1115
F_1 accuracy of rule evolution						
ME	0.94	0.95	0.97	1.0	1.0	0.997
HC_{DC}	0.93	0.86	0.85	1.0	0.999	0.999

One of the recent challenges in information integration research is called Holistic Information Integration [9] where both schema and data issues are addressed within a single integration framework. For example, schema mapping can help with understanding the data and thus with ER while ER could also provide valuable information for schema mapping. Hence, schema mapping and ER can mutually benefit each other in an iterative fashion. While our work does not address the schema mapping problem, we provide a framework for iteratively updating ER results when the comparison logic (related to the schema) changes.

Another related problem is updating clustering results when the records (data) change (also known as incremental clustering). A fundamental difference between incremental clustering and evolving rules is that the former updates clusters when the data changes while the latter updates clusters when the rules change. A number of works explore the problem of clustering data streams. Charikar et al. [4] propose incremental clustering algorithms that minimize the maximum cluster diameter given a stream of records. Aggarwal et al. [1] propose the CluStream algorithm, which views a stream as a changing process over time and provides clustering over different time horizons in an evolving environment. An interesting avenue of further research is to combine clustering techniques for both evolving data and rules. Since our rule evolution techniques are based on materializing ER results, we suspect that the same techniques for evolving data can be applied on the materialized ER results.

Materializing ER results is related to the topics of query optimization using materialized views [6] and incremental view maintenance, which have been studied extensively in the database literature. The focus of the two related works, however, is on optimizing the execution of SQL queries. In comparison, our work solves a similar problem for comparison rules that are Boolean or distance functions. Another fundamental difference between incremental view maintenance and evolving rules is that the former covers the problem of keeping query results up-to-date on data changes while the latter focuses on updating clustering results based on rule changes. Our work is also related to constructing data cubes [10] in data warehouses where each cell of a data cube is a view consisting of an aggregation (e.g., sum, average, count) of interests like total sales. In comparison, rule evolution stores the ER results of comparison rules. Nonetheless, we believe our rule evolution techniques can improve by using techniques from the literature above. For example, deciding which combinations of conjuncts to materialize is related to the problem of deciding which views to materialize.