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 naive 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 three ER models: join, match-based clustering, and distance-based clustering. Using real data sets, we illustrate the cost of materializations and the potential gains over the naive approach.

1 Introduction

Entity resolution [10, 29, 19] (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.

A naive 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 naive 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 naive 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 $B_2$. 

Entity Resolution with Evolving Rules

Steven Euijong Whang and Hector Garcia-Molina
Stanford University, CA
{swhang, hector}@cs.stanford.edu
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 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 $B_2 = p_{name} \land p_{zip}$, 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 $B_3 = p_{name} \land p_{phone}$, 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 objective here is, again, to study when is materialization feasible and to illustrate 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. We then propose efficient rule evolution techniques that use one or more of the four properties (Sections 3 and 5). 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 6) 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 Join Evolution

We consider rule evolution for the problem of identifying the matching pairs of records according to a Boolean
comparison rule. We formalize our join model and provide rule evolution algorithms for different types of
changes in the comparison rule. In general, our rule evolution techniques can be used for any application
that needs to update a join result given a new join condition.

2.1 Join Model

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

A join operation takes as input a set of records \( S \) along with a Boolean comparison rule \( B \) and returns
the set of pairs \( (r_i, r_j) \in S \times S \) such that \( i < j \) and \( B(r_i, r_j) = \text{true} \) by performing pairwise comparisons.
A number of works [2, 3] have proposed efficient join techniques for comparing special data (e.g., sets or
vectors). In the general case where specialized join techniques cannot be used, a common approach to avoiding
exhaustive pairwise comparisons is to use various blocking techniques [24] where only a small subset of the
entire data is resolved at a time.

2.2 Rule Evolution

A rule evolution occurs when a Boolean comparison rule \( B_1 \) is replaced by a new Boolean comparison rule
\( B_2 \). We discuss two basic and one general type of rule evolution and provide algorithms to efficiently produce
the new join result of \( B_2 \) using join results based on \( B_1 \). We first define the notion of relative strictness
between comparison rules.

Definition 2.1 A Boolean comparison rule \( B_1 \) is stricter than another rule \( B_2 \) (denoted as \( B_1 \leq B_2 \)) if
\( \forall r_i, r_j, B_1(r_i, r_j) = \text{true} \) implies \( B_2(r_i, r_j) = \text{true} \).

For example, a comparison rule \( B_1 \) that compares the string distance of two names and returns \( \text{true} \)
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.

The order of strictness among comparison rules is a partial order among comparison rules where the
following properties hold:

- Reflexivity: \( B \leq B \)
- Antisymmetry: if \( B_1 \leq B_2 \) and \( B_2 \leq B_1 \) then \( B_1 = B_2 \)
- Transitivity: if \( B_1 \leq B_2 \) and \( B_2 \leq B_3 \) then \( B_1 \leq B_3 \)

Basic Rule Evolution We start with the two basic types of rule evolution where one rule is stricter than
the other. The first case is where \( B_1 \) is stricter than \( B_2 \). Figure 3 illustrates rule evolution where each of
the three boxes represents all the pairs of records in \( S \). The record pairs that match according to \( B_1 \) (i.e.,
the original ER result) are depicted as the shaded area in the center box while the record pairs that match
according to \( B_2 \) (i.e., the desired new ER result) are depicted as the shaded area in the left box. When
deriving the new join result according to \( B_2 \), we know that all the record pairs that match according to \( B_1 \)
also match according to \( B_2 \). Hence, we only need to compare the record pairs that did not match according
to $B_1$ (i.e., the non-shaded area of the center box) and add into the previous join result the newly matching pairs according to $B_2$.

On the other hand, suppose that the new comparison rule is now $B_3$, and $B_3$ is stricter than the original comparison rule $B_1$. The shaded area of the right box of Figure 3 depicts the matching record pairs according to $B_3$ (i.e., the desired new ER result). In this case, we are guaranteed that all the record pairs that do not match according to $B_1$ will not match according to $B_3$ either. Hence, we only need to compare the record pairs that match according to $B_1$ (i.e., the shaded area of the center box) and only return the pairs that also match according to $B_3$.

![Fig. 3. Basic Rule Evolution for Join](image)

**General Rule Evolution** We now consider the general case of rule evolution where neither the original comparison rule $B_1$ nor the new comparison rule $B_2$ is stricter than the other. This time, we cannot assume that a matching pair of records according to $B_1$ will still match according to $B_2$ or that a non-matching pair according to $B_1$ will still not match according to $B_2$. The naïve approach for rule evolution then would be to run the join using $B_2$ from scratch, i.e., to compare all pairs using $B_2$.

However, we can improve the naïve approach by exploiting the Boolean expression structure of the comparison rules. In practice, $B$ is usually a Boolean expression of smaller binary predicates. For example, a comparison rule that compares the names and addresses of two people can be defined as $p_{\text{name}} \land p_{\text{address}}$ where $p_{\text{name}}$ could be a function that compares the names of two people while the predicate $p_{\text{address}}$ 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 comparison rules are in conjunctive normal form (CNF). For example, the comparison rule $B = p_{\text{name}} \land p_{\text{address}} \land (p_{\text{zip}} \lor p_{\text{phone}})$ is in CNF and has three conjuncts $p_{\text{name}}$, $p_{\text{address}}$, and $p_{\text{zip}} \lor p_{\text{phone}}$.

Figure 4 illustrates general rule evolution by showing rule evolution from $B_1 = p_1 \land p_2$ to $B_2 = p_1 \land p_3$ using the three predicates $p_1$, $p_2$, and $p_3$. The shaded area of the left box represents the matching record pairs according to $B_1$ while the two shaded areas of the right box represent the matching record pairs according to $B_2$. (There are two shaded areas in the right box because the top one represents the record pairs that match according to both $B_2$ and $p_2$ while the bottom one represents the record pairs that still match with $B_2$, but not with $p_2$.)

When computing the join result using $B_1 = p_1 \land p_2$, our general evolution process also computes the join results of $p_1$ and $p_2$, and stores both results. In our example, we only show the join result of $p_1$ as the shaded area of the center box. Once we have the new comparison rule $B_2 = p_1 \land p_3$, general evolution uses the join result of $p_1$ to improve the join performance of $B_2$. Specifically, since we know that $B_2$ is stricter
than $p_1$, general evolution only needs to compare the record pairs that match according to $p_1$ using $p_3$ to derive all the record pairs that match according to $B_2$. (In Section 3.3, we consider the overhead of building the auxiliary structures like the $p_1$ and $p_2$ join results.)

Based on our illustration above, we propose an enhanced technique for general rule evolution that uses join results for common subexpressions of $B_1$ and $B_2$. We first store the join result for each conjunct of the comparison rule $B_1$ as a list of all matching record pairs according to the conjunct. We refer to storing a join result of a conjunct a materialization of that conjunct. Compression techniques can be used to store the lists more compactly (e.g., we can store just the IDs of the records and group the ID pairs by their smaller IDs). We can further materialize various combinations of conjuncts if such combinations are likely to occur in the new comparison rule. The materialization of all conjuncts of $B_1$ can either be done while running the ER algorithm using $B_1$ or separately from the ER algorithm (but before the rule evolution occurs).

Algorithm 1 shows how rule evolution occurs from $B_1$ to $B_2$. For simplicity, we assume that $B_1$ and $B_2$ share at least one conjunct. In Step 3, we first identify the common conjuncts between $B_1$ and $B_2$. (The set of conjuncts of a rule $B$ is denoted as $\text{Conj}(B)$.) We then intersect the record pair lists materialized for the common conjuncts (retrieved using the hash table $H$, which contains the join result for each conjunct in $B_1$) into the intersection $I$. For example, if the rule $B_1 = p_{\text{name}} \land p_{\text{address}} \land p_{\text{zip}}$ evolves to the new rule $B_2 = p_{\text{name}} \land p_{\text{address}} \land p_{\text{phone}}$, we intersect the materialized record pair lists of $p_{\text{name}}$ and $p_{\text{address}}$ (which are the common conjuncts between $B_1$ and $B_2$) to produce a superset of matching record pairs for $B_2$. In general, intersecting the matching record pairs for the common conjuncts is guaranteed to contain all the matching record pairs of $B_2$ because any record pair $(r, s)$ matching according to $B_2$ is also guaranteed to satisfy the common conjuncts between $B_1$ and $B_2$. We can further show that using a conjunct other than the common conjuncts for the intersection may fail to produce a superset of the matching record pairs produced by $B_2$. In Step 4, the MatchingPairs function then compares the record pairs in $I$ using only the conjuncts exclusively in $B_2$ to produce the correct join result of using $B_2$. In the case where $\text{Conj}(B_2) - \text{Conj}(B_1) = \emptyset$, MatchingPairs simply returns $I$.

```
1: input: The comparison rules $B_1$, $B_2$, the join result for each conjunct of $B_1$, a hash table $H$ containing the join results for each conjunct in $B_1$
2: output: Join result using $B_2$
3: Pairs $I \leftarrow \bigcap_{\text{conj} \in \text{Conj}(B_1) \cap \text{Conj}(B_2)} H(\text{conj})$
4: return MatchingPairs($I$, $\text{Conj}(B_2) - \text{Conj}(B_1)$)
```

**Algorithm 1:** General rule evolution algorithm for joins
After the rule evolution is finished, we can prepare for subsequent rule evolutions by materializing the conjuncts exclusively in $B_2$ and optionally removing the materialization of conjuncts exclusively in $B_1$.

**Proposition 2.2** Algorithm 1 correctly returns the matching pairs of different records in $S$ using $B_2$.

**Proof.** We first prove that there are no false positives where a pair of records $(r_i, r_j)$ that do not match according to $B_2$ (i.e., $B_2(r_i, r_j) = \text{false}$) appear in the join result of Algorithm 1. Since at least one of the conjuncts in $B_2$ return false for $(r_i, r_j)$, either $(r_i, r_j)$ is not contained in $I$ or is filtered out by MatchingPairs. As a result, $(r_i, r_j)$ does not appear in the join result. Next, we prove that there are no false negatives where a pair of records $(r_i, r_j)$ that match according to $B_2$ (i.e., $B_2(r_i, r_j) = \text{true}$) do not appear in the join result of Algorithm 1. The pair $(r_i, r_j)$ always appears in $I$ because the common conjuncts of $B_1$ and $B_2$ are satisfied (Step 3). The pair $(r_i, r_j)$ is always produced by MatchingPairs($I, \text{Conj}(B_2) - \text{Conj}(B_1)$) because all the conjuncts in $B_2$ return true for that pair. In the case where $\text{Conj}(B_2) - \text{Conj}(B_1) = \emptyset$, MatchingPairs returns $I$ by definition, and the pair is in $I$.

**Example** Revisiting our motivating example in Section 1, suppose that we are performing rule evolution for a join. We first materialize the two conjuncts of $B_1 = p_{\text{name}} \land p_{\text{zip}}$. The result of conjunct $p_{\text{name}}$ is materialized as the list $\{(r_1, r_2), (r_1, r_3), (r_2, r_3)\}$ since the three records $r_1$, $r_2$, and $r_3$ match with each other. (Throughout this paper, we use square brackets to denote lists and curly brackets to denote sets.) The conjunct $p_{\text{zip}}$ (which also has exactly one predicate) is materialized as $\{(r_1, r_2)\}$ because only $r_1$ and $r_2$ have the same zip code. During the rule evolution using $B_2 = p_{\text{name}} \land p_{\text{phone}}$, we start comparing the record pairs in the $p_{\text{name}}$ list (because $p_{\text{name}}$ is the only conjunct between $B_1$ and $B_2$) using the new $p_{\text{phone}}$ predicate. Since only the pair $(r_2, r_3)$ matches according to $p_{\text{phone}}$, our final result is $\{(r_2, r_3)\}$, which is the exact same solution as comparing all four records $r_1$, $r_2$, $r_3$, and $r_4$ using $B_2$. Finally, if we might have more rule evolutions in the future, we materialize the $p_{\text{phone}}$ conjunct into the list $\{(r_2, r_3)\}$ and optionally remove the materialization of $p_{\text{zip}}$.

3 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 5.) 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.

3.1 Match-based Clustering Model

Just like in Section 2.1, 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., $\forall r_1, r_j, B(r_1, r_j) = B(r_j, r_1)$.

Suppose we are given a set of records $S = \{r_1, \ldots, r_n\}$. An ER algorithm receives as inputs a partition $P_1$ of $S$ and a Boolean comparison rule $B$, and returns another partition $P_2$ 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 $P_1 = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$, and the output using the comparison rule $B_2 = p_{\text{name}} \land p_{\text{zip}}$ was the partition $P_2 = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. 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 $P_3 = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ 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 $P_2$ 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 $B_2 = p_{\text{name}} \land p_{\text{zip}}$ 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 $B(r, s) = \text{false}$. Or the two records could also be in two different clusters $c_i, c_j \in P_o (i \neq j)$ even if $B(r, s) = \text{true}$.

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 the partitions $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, 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 [27].

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 Section 3.2) 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 3.1** Given any input partition $P_1$ of a set of records $S$ and any Boolean comparison rule $B$, a valid ER algorithm $E$ non-deterministically returns an ER result $E(P_1, B)$ 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 $\bar{E}(P_1, B)$, which is a set of partitions of $S$. Hence, $E(P_1, B)$ is always one of the partitions in $\bar{E}(P_1, B)$. For example, given $P_1 = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, $\bar{E}(P_1, B)$ could be $\{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$ while $E(P_1, B) = \{\{r_1, r_2\}, \{r_3\}\}$.

We use Definition 2.1 for comparing the relative strictness between comparison rules.

### 3.2 Properties

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

**Rule Monotonic** Before defining the rule monotonic property, we first define the notion of refinement between partitions.

**Definition 3.2** A partition $P_1$ of a set $S$ refines another partition $P_2$ of $S$ (denoted as $P_1 \leq P_2$) if $\forall c_1 \in P_1, \exists c_2 \in P_2 \text{ s.t. } c_1 \subseteq c_2$.

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 monotonic property, which guarantees that the stricter the comparison rule, the more refined the ER result.

**Definition 3.3** 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 $E(P_1, B_1) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$. If the ER algorithm is $\mathcal{RM}$, $E(P_1, B_2)$ can only return $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ or $\{\{r_1, r_2, r_3, r_4\}\}$.
**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.

**Definition 3.4** 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
- $\forall P_o \in \bar{E}(P_i, B), P_o \leq \{ \bigcup_{c \in P} c \cup \bigcup_{c \in P_i - P} c \}$ 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 $\{ \{ r_3 \}, \{ r_4 \} \}$.

**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 independent.

An ER algorithm is incremental [19, 18] 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 3.5** 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 ($\mathcal{OI}$) [19] 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 $S$).

**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: $\mathcal{SN}, \mathcal{HC}_B, \mathcal{HC}_{BR}$, and $\mathcal{ME}$. 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 3.1.
The sorted neighborhood (SN) algorithm [16] first sorts the records in \( P_1 \) (i.e., we extract all the records from the clusters in \( P_1 \)) 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 \( P_1 = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \) 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 \( P_0 \) 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 \( P_0 = \{ \{ r_1, r_2, r_3 \} \} \).

Proposition 3.6 The SN algorithm is \( \mathcal{R.M} \), but not \( \mathcal{C.F} \).

Proof. We prove that the SN algorithm is \( \mathcal{R.M} \). Given any partition \( P \) and two comparison rules \( B_1 \) and \( B_2 \) such that \( B_1 \subseteq B_2 \), the set of pairs of matching records found by \( B_1 \) is clearly a subset of that found by \( B_2 \), during the first phase of the SN algorithm. As a result, the transitive closure of the matching pairs by \( B_1 \) refines the transitive closure result of \( B_2 \) (i.e., \( P^1_0 \leq P^2_0 \)).

We prove that the SN algorithm is not \( \mathcal{C.F} \) using a counter example. Suppose that the input partition is \( P_1 = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \), and we sort the records in \( P_1 \) by their record IDs (e.g., the record \( r_2 \) has the ID of 2) into the sorted list \([ r_1, r_2, r_3 ]\). Given the comparison rule \( B \), suppose that only the pair \( r_1 \) and \( r_3 \) match with each other. Using a window size of 2, we do not identify any matching pairs of records because \( r_1 \) and \( r_3 \) are never in the same window according to the sorted list. Hence, \( E(P_1, B) = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \). To apply the \( \mathcal{C.F} \) property, we set \( P = \{ \{ r_1 \}, \{ r_3 \} \} \) and \( P_i = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \). The first two conditions in Definition 3.4 are satisfied because \( P \subseteq P_i \) and \( \forall P_o \in E(P_i, B) = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \). \( P_o \leq \bigcup_{i \in I} \bigcup_{j \in I} \) \( \{ \{ r_1 \}, \{ r_3 \} \} \). \( P_o = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \). Also, \( E(P, B) \) is always \( \{ \{ r_1, r_3 \} \} \) (because \( r_1 \) and \( r_3 \) surely match being in the same window) and \( E(P_i - P, B) \) is always \( \{ \{ r_2 \} \} \). However, \( E(P, B) \cup E(P_i - P, B) = \{ \{ r_1, r_3 \}, \{ r_2 \} \} \not\subseteq E(P_i, B) = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \). Violating the \( \mathcal{C.F} \) property.

Hierarchical clustering based on a Boolean comparison rule [4] (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 \text{true} or \text{false}, using the boolean comparison rule \( B \) to compare pairs of records. For example, suppose we have the input partition \( P_1 = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \) and the comparison rule \( B \) where \( B(r_1, r_2) = \text{true} \), \( B(r_2, r_3) = \text{true} \), but \( B(r_1, r_3) = \text{false} \). 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, \( E(P, B) = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \} \} \) \( \{ \{ r_1 \}, \{ r_2, r_3 \} \} \).

Proposition 3.7 The \( HC_B \) algorithm is \( \mathcal{C.F} \), but not \( \mathcal{R.M} \).

Proof. We prove that the \( HC_B \) algorithm is \( \mathcal{C.F} \). Given the four partitions \( P, P_1, P^1_0, P^2_0 \) of Definition 3.4, suppose that \( P^1_0 \cup P^2_0 \not\in E(P_1, B) \). We then prove that the four conditions of Definition 3.4 cannot all be satisfied at the same time. We first assume that the first, third, and fourth conditions are satisfied. That is, \( P \subseteq P_1 \), \( P^1_0 \in E(P, B) \), and \( P^2_0 \in E(P_i - P, B) \). Now suppose when deriving \( E(P_i, B) \) that we “replay” all the merges among the clusters of \( P \) that were used to derive \( P^1_0 \) and then “replay” all the merges among the clusters of \( P_i - P \) that were used to derive \( P^2_0 \). We thus arrive at the state \( P^1_0 \cup P^2_0 \) \( E(P, B) \), and can further merge any matching clusters until no clusters match to produce a possible ER result in \( E(P_i, B) \). Since \( P^1_0 \cup P^2_0 \not\in E(P_1, B) \), there must have been new merges among clusters in \( P^1_0 \) and \( P^2_0 \). Since none of the
clusters within $P_1^0$ or clusters within $P_2^0$ match with each other, we know that there must have been at least one more merge between a cluster in $P_1^0$ and a cluster in $P_2^0$ when deriving $E(P_i, B)$. As a result, the second condition cannot hold because there exists an ER result $P_o \in E(P_i, B)$ such that $P_o \notin \bigcup_{e \in P} c. \bigcup_{e \in P_i - P} e$ because there exists a cluster in $P_o$ that contains records in $P$ as well as $P_i - P$. Hence we have proved that all four conditions can never be satisfied if $P_1^0 \cup P_2^0 \notin E(P_i, B)$.

We prove that the $HC_B$ algorithm is not $RM$ using a counter example. Consider the example we used for illustrating the $HC_B$ algorithm where $E(P_i, B) = \{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$. Now without having to define a new Boolean comparison rule, by setting $B_1 = B, B_2 = B, P_1^0 = \{\{r_1, r_2\}, \{r_3\}\}$, and $P_2^0 = \{\{r_1\}, \{r_2, r_3\}\}$, we see that $P_1^0 \not\subseteq P_2^0$ (although $B_1 \leq B_2$), contradicting the $RM$ property. This result suggests that any ER algorithm that can produce more than one possible partition given any $P_i$ and $B$ is not $RM$. We show in Proposition 3.12 the equivalent statement that any ER algorithm that is $RM$ always returns a unique solution.

**$HC_{BR}$** We define the $HC_{BR}$ algorithm as a hierarchical clustering algorithm based on a Boolean comparison rule (just like $HC_B$). In addition, we require the comparison rule to match two clusters whenever at least one of the records from the two clusters match according to $B$. (This property is equivalent to the representativity property in reference [4].) 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 can prove that the $HC_{BR}$ is both $RM$ and $CF$ (see Proposition 3.10). We first define the notion of connectedness. Two records $r$ and $s$ are connected under $B$ and $P_i$ if there exists a sequence of records $[r_1 (= r), \ldots, r_n (= s)]$ where for each pair $(r_i, r_{i+1})$ in the path, either $B(r_i, r_{i+1}) = \text{true}$ or $\exists c \in P_i$ s.t. $r_i \in c, r_{i+1} \in c$. Notice that connectedness is “transitive,” i.e., if $r$ and $s$ are connected and $s$ and $t$ are connected, then $r$ and $t$ are also connected.

**Lemma 3.8** Two records $r$ and $s$ are connected under $B$ and $P_i$ if and only if $r$ and $s$ are in the same cluster in $P_o \in E(P_i, B)$ using the $HC_{BR}$ algorithm.

**Proof.** Suppose that $r$ and $s$ are in the same cluster in $P_o$. If $r$ and $s$ are in the same cluster in $P_i$, then $r$ and $s$ are trivially connected under $B$ and $P_i$. Otherwise, there exists a sequence of merges of the clusters in $P_i$ that grouped $r$ and $s$ together. If two clusters $c_a$ and $c_b$ in $P_i$ merge where $r \in c_a$ and $s \in c_b$, then $r$ and $s$ are connected because there is at least one pair of records $r' \in c_a$ and $s' \in c_b$ such that $r'$ and $s'$ match (i.e., $B(r', s') = \text{true}$), and $r$ is connected to $r'$ while $s$ is connected to $s'$. Furthermore, we can prove that any record in $c_a \cup c_b$ is connected with any record in a cluster $c_c$ that merges with $c_a \cup c_b$ using a similar argument: we know there exists a pair of records $r' \in c_a \cup c_b$ and $s' \in c_c$ that match with each other, and $r$ is connected to $r'$ while $s$ is connected to $s'$, which implies that $r$ and $s$ are connected under $B$ and $P_i$. By repeatedly applying the same argument, we can prove that any $r$ and $s$ are connected if they end up in the same cluster in $P_o$.

Conversely, suppose that $r$ and $s$ are connected as the sequence $[r_1 (= r), \ldots, r_n (= s)]$ under $B$ and $P_i$. If $r$ and $s$ are in the same cluster in $P_i$, they are already clustered together in $P_o$. Otherwise, all the clusters that contain $r_1, \ldots, r_n$ eventually merge together according to the $HC_{BR}$ algorithm, clustering $r$ and $s$ together in $P_o$.

We next prove that $HC_{BR}$ returns a unique solution.

**Proposition 3.9** The $HC_{BR}$ algorithm always returns a unique solution.

**Proof.** Suppose that $HC_{BR}$ produces two different output partitions for a given partition $P_i$ and comparison rule $B$, i.e., $E(P_i, B) = \{P_1^0, P_2^0, \ldots\}$. Then there must exist two records $r$ and $s$ that have merged into the same cluster according to one ER result, but not in the same cluster for the other ER result. Suppose that $r$ and $s$ are in the same cluster in $P_1^0$, but in separate clusters in $P_2^0$. Since $r$ and $s$ are clustered together in $P_1^0$, they are connected by Lemma 3.8. Hence, $r$ and $s$ must also be clustered in $P_2^0$ again by Lemma 3.8,
The Proof. partition. contradicting our hypothesis that they are in different clusters in \( P^2_o \). Hence, \( HC_{BR} \) always returns a unique partition.

Finally, we prove that \( HC_{BR} \) is both \( RM \) and \( CF \).

**Proposition 3.10** The \( HC_{BR} \) algorithm is both \( RM \) and \( CF \).

**Proof.** The \( HC_{BR} \) algorithm is \( CF \) because the \( HC \) algorithm already is \( CF \). To show that the \( HC_{BR} \) algorithm also is \( RM \), suppose that \( B_1 \leq B_2 \). Then all the clusters that match according to \( B_1 \) also match according to \( B_2 \). Hence for any \( P^i_o \in \tilde{E}(P_i, B_1) \), we can always construct an ER result \( P^2_o \in \tilde{E}(P_i, B_2) \) (which is unique by Proposition 3.9) where \( P^1_o \leq P^2_o \) by performing the exact same merges done for \( P^1_o \) and then continuing to merge clusters that still match according to \( B_2 \) until no clusters match according to \( B_2 \).

**ME** The Monge Elkan (ME) clustering algorithm (we define a variant of the algorithm in [23] for simplicity) first sorts the records in \( P \) (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 \( B(r_1, r_2) = \text{true} \), but \( B(r_1, r_3) = \text{false} \) and \( B(r_2, r_3) = \text{false} \). Each scanned record is then compared with clusters in a fixed-length queue. A record \( r \) matches with a cluster \( c \) if \( B(r, c) = \text{true} \) 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 then remove \( \{r_1, r_2\} \). Hence, the only possible ER result is \( \{r_1, r_2, r_3\} \) and thus \( E(P, B) = \{\{r_1, r_2\}, \{r_3\}\} \). In general, ME always returns a unique partition.

**Proposition 3.11** The ME algorithm does not satisfy \( RM \) or \( CF \).

**Proof.** We prove that the ME algorithm is not \( RM \) using a counter example. Suppose that the input partition is \( P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\} \), and we sort the records by their record IDs (e.g., the record \( r_2 \) has the ID of 2) into the sorted list of records \( [r_1, r_2, r_3] \). Suppose that \( B_1(r_1, r_3) = \text{true} \), but \( B_1(r_1, r_2) = \text{false} \) and \( B_1(r_2, r_3) = \text{false} \). Compared to \( B_1 \), the only difference of \( B_2 \) is that \( B_2(r_2, r_3) = \text{true} \). Clearly, \( B_1 \leq B_2 \). Using a queue size of 2, \( E(P_i, B_1) \) returns \( \{r_1, r_3\}, \{r_2\}\) only because \( r_1 \) and \( r_2 \) are inserted into the queue separately, and \( r_3 \) then merges with \( \{r_1\} \). On the other hand, \( E(P_i, B_2) \) returns \( \{r_1\}, \{r_2, r_3\}\) because \( r_1 \) and \( r_2 \) are inserted into the queue separately, and \( r_3 \) matches with \( \{r_2\} \) first. Since \( E(P_i, B_1) = \{\{r_1, r_3\}, \{r_2\}\} \not\subseteq \{\{r_1\}, \{r_2, r_3\}\} = E(P, B_2) \), the \( RM \) property does not hold.

We prove that the ME algorithm is not \( CF \) using a counter example. Suppose that the input partition is \( P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\} \), and we sort the records by their IDs into the sorted list of records \( [r_1, r_2, r_3] \). Suppose that \( B(r_1, r_3) = \text{true} \), but \( B(r_1, r_2) = \text{false} \) and \( B(r_2, r_3) = \text{false} \). Using a queue size of 1, we do not identify any matching pairs because \( r_1 \) is never compared with \( r_3 \) because once \( r_2 \) enters the queue, \( \{r_1\} \) is pushed out of the queue. Hence, \( E(P_i, B) \) is always \( \{\{r_1\}, \{r_2\}, \{r_3\}\} \). Using Definition 3.5, suppose we set \( P = \{\{r_1\}, \{r_3\}\} \) and \( P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\} \). The first condition is satisfied because \( P \subseteq P_i \). The second condition is satisfied because \( \forall P_o \in \tilde{E}(P_i, B), P_o = \{\{r_1\}, \{r_2\}, \{r_3\}\} \leq \{\bigcup_{c \in P \setminus P_o} c, \bigcup_{c \in P \setminus P_o} c\} = \{\{r_1\}, \{r_2\}\} \}. Also, \( P^1_o \) is always \( \{\{r_1, r_3\}\} \) because \( r_3 \) matches with \( \{r_1\} \) while \( \{r_1\} \) is in the queue, and \( P^2_o \) is always \( \{\{r_2\}\} \). As a result, \( P^1_o \cup P^2_o = \{\{r_1, r_3\}, \{r_2\}\} \notin \tilde{E}(P_i, B) = \{\{r_1\}, \{r_2\}, \{r_3\}\}\), contradicting the \( CF \) property.

The venn diagram in Figure 5 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 used to
prove Propositions 3.16 and 3.17, respectively. For now, ignore the $HC_{DS}$ and $HC_{DC}$ algorithms, which are distance-based clustering algorithms covered in Section 5.2.

We first add the $OI$ property into the venn diagram in Figure 5. Proposition 3.12 shows that the $OI$ property includes the $RM$ property. Proposition 3.13 shows that the $ME$ algorithm is $OI$, but not $RM$. Also, $OI$ partially overlaps with $CF$, but does not contain it. According to Proposition 3.14, the $HC_{BR}$ algorithm is both $OI$ and $CF$. According to Proposition 3.15, the $HC_B$ algorithm is $CF$, but not $OI$. Finally, Proposition 3.13 shows that the $ME$ algorithm is $OI$, but not $CF$.

**Proposition 3.12** Any ER algorithm that is $RM$ is also $OI$.

**Proof.** Suppose that we are given an $RM$ ER algorithm $E$, and the $OI$ property does not hold. Then there exists a partition $P_i$ and comparison rule $B$ such that $E(P_i, B)$ contains at least two different partitions $P_x$ and $P_y$. Without loss of generality, we assume that $P_x \neq P_y$. However, we violate Definition 3.3 by setting $B_1 = B$, $B_2 = B$, $P_o^1 = P_x$, and $P_o^2 = P_y$ because $B_1 \leq B_2$, but $P_o^1 = P_x \neq P_y = P_o^2$. Hence, $E$ cannot satisfy the $RM$ property, a contradiction.

**Proposition 3.13** The $ME$ algorithm is $OI$ and $GI$, but not $RM$ or $CF$.

**Proof.** Proposition 3.11 shows that $ME$ does not satisfy $RM$ or $CF$. The $ME$ algorithm is $OI$ because it first sorts the records in $P_i$ before resolving them with a sliding window and thus produces a unique solution. The $ME$ algorithm is $GI$ because $E(P_o^1 \cup (P_i - P), B)$ always returns the same result as $E(P_i, B)$. That is, $ME$ extracts all the records from its input partition before sorting and resolving them, and $P_o^1 \cup (P_i - P)$ contains the exact same records as those in $P_i$ (i.e., $\bigcup_{c \in P_o^1 \cup (P_i - P)} c = \bigcup_{c \in P_i} c$).

**Proposition 3.14** The $HC_{BR}$ algorithm is $RM$, $CF$, $GI$, and $OI$.

**Proof.** Proposition 3.10 shows that $HC_{BR}$ is $RM$ and $CF$. The $HC_{BR}$ algorithm is also $OI$ by Proposition 3.9. To show that $HC_{BR}$ is $GI$, consider the four partitions $P, P_i, P_o^1, P_o^2$ of Definition 3.5, and suppose that the three conditions $P \subseteq P_i$, $P_o^1 \in E(P_i, B)$, and $P_o^2 \in E(P_o^1 \cup (P_i - P), B)$ hold. We show that $P_o^2 \in E(P_i, B)$. Starting from $P_i$, $P_o^2$ is the result of “replaying” the merges used to produce $P_o^1$, and then “replaying” the merges used to produce a possible result of $E(P_o^1 \cup (P_i - P), B)$. Since no clusters in $P_o^2$ match with each other, $P_o^2$ is also a possible result for $E(P_i, B)$. Hence, $P_o^2 \in E(P_i, B)$.

**Proposition 3.15** The $HC_B$ algorithm is $CF$ and $GI$, but not $OI$ (and consequently not $RM$).

**Proof.** Proposition 3.7 shows that $HC_B$ is $CF$. The proof that $HC_B$ is $GI$ is identical to the proof for $HC_{BR}$ and is omitted.
The $HC_B$ algorithm does not satisfy $OI$ because, depending on the order of records compared, there could be several possible ER results. For example, given the input partition $P_1 = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the comparison rule $B$, suppose that $B(r_1, r_2) = \text{true}$ and $B(r_2, r_3) = \text{true}$, but $B(r_1, r_3) = \text{false}$. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using $B$. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Then depending on the order of clusters in $P_1$ compared, the $HC_B$ algorithm either produces $\{\{r_1, r_2\}, \{r_3\}\}$ or $\{\{r_1, r_2, r_3\}\}$.

We now add the $GI$ property into the venn diagram in Figure 5. The $GI$ property partially intersects with the other three properties $RM$, $CF$, and $OI$, and does not include any of them. Proposition 3.14 shows that $HC_B$ is $RM$, $CF$, $GI$, and $OI$. Hence, $GI$ intersects with the other three properties. Proposition 3.15 shows that the $HC_B$ algorithm is $GI$, but not $OI$ (and consequently not $RM$). Proposition 3.13 shows that the $ME$ algorithm is $GI$, but not $CF$. Hence, none of the other three properties include $GI$. Proposition 3.16 shows the existence of an ER algorithm that is $RM$ (the same algorithm is also $OI$), but not $GI$. Proposition 3.17 shows the existence of an ER algorithm that is $CF$, but not $GI$. Hence, $GI$ does not include any of the other three properties.

**Proposition 3.16** There exists an ER algorithm that is $RM$ (and consequently $OI$ as well), but not $GI$ or $CF$.

**Proof.** We define a variant of the $SN$ algorithm called $SN^2$. Recall that the $SN$ algorithm involves identifying the matching pairs of records by first sorting the records from $P_1$ and then comparing records within the same sliding window. At the end, we produce a transitive closure of all the matching records. During the transitive closure, however, we define $SN^2$ to also consider all the records within the same cluster in the input partition $P_1$ as matching. For example, suppose we have $P_1 = \{\{r_1, r_2\}, \{r_3\}\}$ and a comparison rule $B$ such that $B(r_1, r_3) = \text{true}$, but $B(r_1, r_2) = \text{false}$ and $B(r_2, r_3) = \text{false}$. Given a window size of 2, suppose that we sort the records in $P_1$ by record ID into the list $[r_1, r_2, r_3]$. Then the original $SN$ algorithm sorts will return $\{\{r_1\}, \{r_2\}, \{r_3\}\}$ because $r_1$ and $r_2$ are never compared in the same window. However, the new $SN^2$ algorithm will consider $r_1$ and $r_2$ as matching because they were in the same cluster in the input partition $P_1$. Hence, the result of $SN^2$ is $\{\{r_1, r_2\}, \{r_3\}\}$.

We prove that the $SN^2$ algorithm is $RM$. Given any partition $P$ and two comparison rules $B_1$ and $B_2$ such that $B_1 \leq B_2$, the set of pairs of matching records found by $B_1$ is clearly a subset of that found by $B_2$, during the first phase of the $SN$ algorithm where we find all the matching pairs of records within the same sliding window at any point. In addition, the set of matching pairs of records identified from $P_i$ is the same for both $B_1$ and $B_2$. As a result, the transitive closure of the matching pairs by $B_1$ refines the transitive closure result of $B_2$ (i.e., for any $P_o^1 \in E(P_i, B_1)$ and $P_o^2 \in E(P_i, B_2)$, $P_o^1 \leq P_o^2$).

We prove that the $SN^2$ algorithm is not $GI$ using a counter example. Suppose that we have $P_1 = \{\{r_1, r_2\}, \{r_3\}\}$ and a comparison rule $B$ such that $B(r_1, r_3) = \text{true}$, but $B(r_1, r_2) = \text{false}$ and $B(r_2, r_3) = \text{false}$. Given a window size of 2, suppose that we sort the records in $P_1$ by record ID into the list $[r_1, r_2, r_3]$. As a result, $E(P_1, B) = \{\{r_1, r_2\}, \{r_3\}\}$ because $r_1$ and $r_3$ are never compared within the same window. Using Definition 3.5, suppose that we set $P = \{\{r_1\}, \{r_3\}\}$ and $P_1 = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. As a result, $P_o^1 = E(P, B) = \{\{r_1, r_3\}\}$ because $r_1$ and $r_3$ surely match being in the same window. Also, $P_o^2 = E(P_1, B) = \{\{r_1, r_2\}, \{r_3\}\}$ because $r_1$ and $r_3$ are in the same cluster of the input partition $P_1 \cup (P_1 - P)$. Since $E(P, B)$ is always $\{\{r_1\}, \{r_2\}, \{r_3\}\}$, $P_o^3 \notin E(P, B) = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. Hence, the $GI$ property is not satisfied. (On the other hand, the original $SN$ algorithm is $GI$; see Proposition 3.18.)

To prove that $SN^2$ is not $CF$ (in order to show that Figure 5 correctly categorizes $SN^2$), we can directly use the proof of Proposition 3.6 that was used to show $SN$ is not $CF$.

**Proposition 3.17** There exists an ER algorithm that is $CF$ and $OI$, but not $GI$ or $RM$.

**Proof.** We define a variant of the $HC_B$ algorithm (called $HC_B^2$) where all matching clusters are merged, but only in a certain order. For example, we can define a total ordering between pairs of clusters where the clusters with the ‘smallest record IDs’ are merged first. We first define the following notations: $MinID_1$
= \min\{\min_{r \in C_1} \text{ID}(r), \min_{r \in C_2} \text{ID}(r)\}, \quad \text{MaxID}_1 = \max\{\min_{r \in C_1} \text{ID}(r), \min_{r \in C_2} \text{ID}(r)\}, \quad \text{MinID}_2 = \min\{\min_{r \in C_3} \text{ID}(r), \min_{r \in C_4} \text{ID}(r)\}, \quad \text{MaxID}_2 = \max\{\min_{r \in C_3} \text{ID}(r), \min_{r \in C_4} \text{ID}(r)\} \) where the ID(r) function returns the ID of r. We merge the pair of clusters \((c_1, c_2)\) before the pair of matching clusters \((c_3, c_4)\) if either \(\text{MinID}_1 < \text{MinID}_2\) or both \(\text{MinID}_1 = \text{MinID}_2\) and \(\text{MaxID}_1 < \text{MaxID}_2\). For example, the matching clusters \(\{r_3, r_5\}\) and \(\{r_6, r_7\}\) merge before the matching clusters \(\{r_4, r_8\}\) and \(\{r_5, r_7\}\) because \(\text{MinID}_1 = 3, \text{MaxID}_1 = 6, \text{MinID}_2 = 4, \text{MaxID}_2 = 6\) and thus \(\text{MinID}_1 < \text{MinID}_2\). In general, we can use any total ordering of pairs of clusters. As a result of using the total ordering, the \(HC_B^2\) algorithm always produces a unique ER result. For example, suppose that we have \(P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}\) and the boolean comparison rule \(B\) where \(B(r_1, r_2) = \text{true}\) and \(B(r_2, r_3) = \text{true}\), but \(B(r_1, r_3) = \text{false}\). Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using \(B\). For instance, when comparing \(\{r_1, r_2\}\) with \(\{r_3\}\), we return the result of \(B(r_1, r_3)\). Then the ER result \(E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}\). The clusters \(\{r_1\}\) and \(\{r_2\}\) merge before \(\{r_2\}\) and \(\{r_3\}\) because \(\text{MinID}_1 = 1 < 2 = \text{MinID}_2\). Once \(\{r_1\}\) and \(\{r_2\}\) merge, \(\{r_1, r_2\}\) does not match with \(\{r_3\}\) because \(B(r_1, r_3) = \text{false}\).

We show that the \(HC_B^2\) algorithm is \(\mathcal{CF}\). Given the four partitions \(P, P_1, P_2^1, P_2^2\), suppose that the four conditions in Definition 3.4 are satisfied. That is, \(P \subseteq P_i\), \(\forall P_o \in E(P_i, B), P_o \subseteq \{\bigcup_{c \in P_i - c}, \bigcup_{c \in P_i - P_o}\}, P_1 \in E(P, B), \) and \(P_2^2 \in E(P - P, B)\). Since \(HC_B^2\) returns a unique solution, there is exactly one \(P_o \in E(P, B)\). Suppose that \(P_o\) was generated via a sequence of cluster merges \(M_1, M_2, \ldots\) where each \(M_i\) involves a merge of two clusters. Since \(P_o \subseteq \{\bigcup_{c \in P_i - c}, \bigcup_{c \in P_i - P_o}\}\), we can split the sequence into merges \(M_1^o, M_2^o, \ldots\) that only involve clusters in \(P\) and merges \(M_3^o, M_4^o, \ldots\) that only involve clusters in \(P_i - P\). We can run the first batch of merges \(M_1^o, M_2^o, \ldots\) to produce a possible result of \(E(P, B)\) and run the second batch of merges \(M_3^o, M_4^o, \ldots\) to produce a possible result of \(E(P_i - P, B)\). Since \(HC_B^2\) returns a unique solution, both ER results \(E(P, B)\) and \(E(P_i - P, B)\) are in fact unique and thus are equal to \(P_1^o\) and \(P_2^2\), respectively.

The union of \(P_1\) and \(P_2\) is equivalent to the result of running the merges \(M_1, M_2, \ldots, \) i.e., \(E(P_i, B)\). Hence, \(P_1^o \cup P_2^2 \in E(P_i, B)\).

The \(HC_B^2\) algorithm is \(\mathcal{OT}\) because the merges are done in a fixed order.

We show that the \(HC_B^2\) algorithm does not satisfy \(\mathcal{GT}\) using a counter example. Suppose that we have \(P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}\) and the Boolean comparison rule \(B\) where \(B(r_1, r_2) = \text{true}\) and \(B(r_2, r_3) = \text{true}\), but \(B(r_1, r_3) = \text{false}\). Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using \(B\). For instance, when comparing \(\{r_1, r_2\}\) with \(\{r_3\}\), we return the result of \(B(r_1, r_3)\). Then the ER result \(E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\} \) because \(\{r_1\}\) and \(\{r_2\}\) merge first (before \(\{r_2\}\) and \(\{r_3\}\)) and then \(\{r_1, r_2\}\) does not match with \(\{r_3\}\) because \(B(r_1, r_3) = \text{false}\). However, in Definition 3.5 suppose that we set \(P = \{\{r_2\}, \{r_3\}\}\) and \(P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}\). Then \(P_1^o = E(P, B) = \{\{r_2, r_3\}\} \) because \(\{r_2\}\) matches with \(\{r_3\}\). Also \(P_2^2 = E(P_1^o \cup (P_i - P), B) = E(\{\{r_2, r_3\}, \{r_1, r_2, r_3\}\}, B) = \{\{r_1, r_2, r_3\}\} \) because \(\{r_2, r_3\}\) and \(\{r_1, r_2, r_3\}\) match. Since \(E(P_i, B)\) is always \(\{\{r_1, r_2, r_3\}\}\), \(P_2^2 \notin E(P_i, B) = \{\{r_1, r_2, r_3\}\}\). Hence, the \(\mathcal{GT}\) property is not satisfied.

We show that the \(HC_B^2\) algorithm is not \(\mathcal{RM}\) using a counter example. We use the same example in the previous paragraph where \(E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}\). Now suppose that we are given a stricter comparison rule \(B_1\) where \(B_1(r_1, r_2) = \text{true}\), but \(B_1(r_1, r_2) = \text{false}\) and \(B_1(r_1, r_3) = \text{false}\). Then \(E(P_i, B_1) = \{\{r_1, r_2, r_3\}\}\) because only \(\{r_2\}\) and \(\{r_3\}\) match. Hence, although \(B_1 \leq E(P_i, B) = \{\{r_1, r_2, r_3\}\}\), \(E(P_i, B_1) \notin E(P_i, B)\), violating \(\mathcal{RM}\).

Finally, Proposition 3.18 shows that the \(SN\) algorithm is \(\mathcal{RM}, \mathcal{OT}\), and \(\mathcal{GT}\), but not \(\mathcal{CF}\).

**Proposition 3.18** The \(SN\) algorithm is \(\mathcal{RM}\) (and consequently \(\mathcal{OT}\) and \(\mathcal{GT}\)), but not \(\mathcal{CF}\).

**Proof.** Proposition 3.6 shows that \(SN\) is \(\mathcal{RM}\) (and consequently \(\mathcal{OT}\)), but not \(\mathcal{CF}\). We prove that the \(SN\) algorithm is \(\mathcal{GT}\). Recall that \(SN\) extracts all records in the input partition before sorting and resolving them. Hence, the ER result is the same for different input partitions as long as they contain the same records. For example, \(E(\{\{r_1, r_2\}, \{r_3\}\}, B) = E(\{\{r_1\}, \{r_2, r_3\}\}, B)\) because the two input partitions contain the same records \(r_1, r_2, r_3\) and \(r_3\). In Definition 3.5, we know that \(E(P_1^o \cup (P_i - P), B)\) always returns the same result as \(E(P_i, B)\) because \(P_1^o \cup (P_i - P)\) contains the same records as those in \(P_i\). Thus, for any \(P_2 \in \{\{r_1\}, \{r_2\}, \{r_3\}\}\), \(P_2 \in E(P_i, B)\).
3.3 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 addresses of two people can be defined as $p_{\text{name}} \land p_{\text{address}}$ where $p_{\text{name}}$ could be a function that compares the names of two people while the predicate $p_{\text{address}}$ 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 $B = p_1 \land p_2 \land (p_3 \lor p_4)$ is in CNF and has three conjuncts $p_1$, $p_2$, and $p_3 \lor p_4$.

When we compute an earlier result $E(P_i, B_1)$ where say $B_1 = p_1 \land p_2 \land p_3$, we can also materialize results such as $E(P_i, p_1)$, $E(P_i, p_2)$, $E(P_i, p_1 \land p_2)$, and so on. The most useful materializations will be those that can help us later with $E(P_i, B_2)$. (See Section 4.) 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 6.5, 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.

3.4 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 Section 4 for possible optimizations).

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 $P_1 \land P_2$) 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. We also show the following lemma regarding the meet operation.

Lemma 3.19 If $\forall i, P \leq P_i$ then $P \leq \land P_i$

Proof. We prove by induction on the number $k$ of partitions that are combined with the meet operation.

Base case: $k = 1$. Obviously, $P \leq P_1$.

Induction: Suppose that $P \leq \land_{i \leq 1, \ldots, k} P_i$. We then show that the same equation holds when $k$ increases by 1. Any pair of records $r, s$ that are clustered in $P$ are also clustered in $\land_{i \leq 1, \ldots, k} P_i$ and $P_{k+1}$ by the induction hypothesis. Since $r$ and $s$ are clustered in both $\land_{i \leq 1, \ldots, k} P_i$ and $P_{k+1}$, $r$ and $s$ are also clustered in the meet of the two partitions, i.e., $\land_{i \leq 1, \ldots, k, k+1} P_i$. Hence any pair of records that are clustered in $P$ are also clustered in $\land_{i \leq 1, \ldots, k, k+1} P_i$.  

15
Lemma 3.22 which also implies that $\mathcal{E} \cup \mathcal{Y}$ from that the input partition $\mathcal{M}$, the conjuncts of a comparison rule $B$ is denoted as $\text{Conj}(B)$. For simplicity, we assume that $B_1$ and $B_2$ share at least one conjunct. Step 3 exploits the $\mathcal{R}\mathcal{M}$ property and meets the partitions of the common conjuncts between $B_1$ and $B_2$. For example, suppose that we have $B_1 = p_1 \land p_2 \land p_3$ and $B_2 = p_1 \land p_2 \land p_4$. 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 $\mathcal{R}\mathcal{M}$, 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 $\mathcal{C}\mathcal{F}$ 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 $B_2$.

Algorithm 2: Rule evolution given $\mathcal{R}\mathcal{M}$ and $\mathcal{C}\mathcal{F}$

In order to prove that Algorithm 2 is correct, we first prove three lemmas below. For simplicity, we denote $\{c' \in P_i | c' \subseteq c\}$ as $\text{IN}(P_i, c)$. For a set of clusters $Q$, we define $\text{IN}(P_i, Q) = \bigcup_{c \in Q} \text{IN}(P_i, c)$.

Lemma 3.20 For an $\mathcal{R}\mathcal{M}$ algorithm, $\forall P_o \in \bar{E}(P_i, B_2), P_o \leq M$.

Proof. We first use the $\mathcal{R}\mathcal{M}$ property to prove that $\forall P_o \in \bar{E}(P_i, B_2), P_o \leq M$. For each $\text{conj} \in \text{Conj}(B_1) \cap \text{Conj}(B_2)$, $B_2 \leq \text{conj}$. Hence, by $\mathcal{R}\mathcal{M}$, $\forall P_o^1 \in \bar{E}(P_i, B_2)$ and $\forall P_o^2 \in \bar{E}(P_i, \text{conj})$, $P_o^1 \leq P_o^2$. Since $M = \bigwedge_{\text{conj} \in \text{Conj}(B_1) \cap \text{Conj}(B_2)} E(P_i, \text{conj})$, we conclude that $\forall P_o \in \bar{E}(P_i, B_2), P_o \leq M$ using Lemma 3.19. (Notice that $M$ is unique because each $E(P_i, \text{conj})$ is also unique by $\mathcal{R}\mathcal{M}$ and $\mathcal{C}\mathcal{F}$.)

Lemma 3.21 Say we have an algorithm that is $\mathcal{R}\mathcal{M}$ and $\mathcal{C}\mathcal{F}$, an initial partition $P_i$, and two rules $B_1$, $B_2$ with conjuncts $\text{Conj}(B_1)$, $\text{Conj}(B_2)$. Let $M = \bigwedge_{\text{conj} \in \text{Conj}(B_1) \cap \text{Conj}(B_2)} E(P_i, \text{conj})$. For any $W \subseteq M$, $E(\text{IN}(P_i, W), B_2) \leq W$.

Proof. We use $\mathcal{C}\mathcal{F}$ to prove that for any $W \subseteq M$, $E(\text{IN}(P_i, W), B_2) \leq W$. To avoid confusion with the initial set of clusters $P_i$, we use the symbol $P_i'$ instead of $P_i$ when using Definition 3.4. We satisfy the first two conditions in Definition 3.4 by setting $P = \text{IN}(P_i, W)$ and $P_i' = P_i$. The first condition, $P \subseteq P_i'$, is satisfied because $\text{IN}(P_i, W)$ is a subset of $P_i$ by definition. The second condition, $\forall P_o \in \bar{E}(P_i', B_2), P_o \leq \bigcup_{c \in P_i} \bigcup_{c' \in \text{IN}(P_i, W) \setminus \{c\}}$, is satisfied because we know by Lemma 3.20 that $\forall P_o \in \bar{E}(P_i, B_2), P_o \leq M$. Also, we know that $W \subseteq M$. Thus, for $P_o^1 = E(P_i, B_2)$ and $P_o^2 = E(P_i' - P, B_2)$, $P_o^1 \cup P_o^2 = E(P_i', B_2)$ (Notice that $E$ returns a unique solution being $\mathcal{R}\mathcal{M}$ and $\mathcal{C}\mathcal{F}$.) Since $E(P_i', B_2) \leq M$, $P_o^1 = E(\text{IN}(P_i, W), B_2) \leq W$, which also implies that $E(\text{IN}(P_i, W), B_2) \leq M$.

Lemma 3.22 Given the same setup as in Lemma 3.21, let $Y \subseteq M$ and $Z \subseteq M$ such that $Y \cap Z = \emptyset$ and $Y \cup Z = W$ (note: $W \subseteq M$). Let $Q = E(\text{IN}(P_i, W), B_2)$ (there is only one solution). Then $Q \leq \{\bigcup_{c \in Y} c, \bigcup_{c \in Z} c\}$.

Proof. Suppose that $Q \not\leq \{\bigcup_{c \in Y} c, \bigcup_{c \in Z} c\}$. Then a cluster in $Q$ must have one cluster from $Y$ and one from $Z$. Since $Q \leq M$ (by Lemma 3.21), however, we arrive at a contradiction.

16
Proposition 3.23  Algorithm 2 correctly returns a partition $P_o \in \bar{E}(P_1, B_2)$.

Proof. We use $\mathcal{CF}$ to prove that $P_o = \bigcup_{c \in M} E(\{c' \in P_t | c' \subseteq c\}, B_2) \in \bar{E}(P_1, B_2)$. Suppose that $M = \{c_1, \ldots, c_{|M|}\}$. We omit the $B_2$ from any expression $E(P, B_2)$ for brevity ($B_1$ is not used in this proof). To avoid confusion with the initial set of clusters $P_t$, we use the symbol $P_t'$ instead of $P_t$ when using Definition 3.4 later in this proof. We define the following notation: $\alpha(k) = \bigcup_{c \in \{c_1, \ldots, c_k\}} E(IN(P_t, c))$ and $\beta(k) = \bigcup_{c \in M - \{c_1, \ldots, c_k\}} E(IN(P_t, c))$. Our goal $\bigcup_{c \in M} E(\{c' \in P_t | c' \subseteq c\}, B_2) \in \bar{E}(P_1; B_2)$ can thus be written as $\alpha(|M|) \in \bar{E}(P_1)$. To prove that $\alpha(|M|) \in \bar{E}(P_1)$, we first prove a more general statement: any $\alpha(k) \cup E(\beta(k)) \in \bar{E}(P_1)$ for $k \in \{0, \ldots, |M|\}$. Clearly, if our general statement holds, we can show that $\alpha(|M|) \in \bar{E}(P_1)$. We use induction on the number of partitions $k$ processed in isolation.

Base case: $k = 0$. Then any $\alpha(0) \cup E(\beta(0)) \in \bar{E}(P_1)$.

Induction: Suppose the equation above holds for $k = n$ (i.e., any $\alpha(n) \cup E(\beta(n)) \in \bar{E}(P_1)$). We want to show that the equation also holds when $k = n + 1$ where $n + 1 \leq |M|$. We first show that $E(IN(P_t, c_{n+1})) \cup E(\beta(n+1)) = E(\beta(n))$ using $\mathcal{CF}$. We satisfy the first two conditions of Definition 3.4 by setting $P = IN(P_t, c_{n+1})$ and $P'_t = \beta(n)$. The first condition, $P \subseteq P'$, is satisfied because $IN(P_t, c_{n+1})$ is a subset of $\beta(n)$ by definition. The second condition, $\forall P_o \in \bar{E}(P'_t), P_o \subseteq \bigcup_{c \in P'_t - P} E(IN(P_t, c))$, is satisfied by Lemma 3.22 by setting $Y = \{c_{n+1}\}$ and $Z = \{c_{n+2}, \ldots, c_{|M|}\}$. Hence, for $P_1^o = E(P)$ and $P_2^o = E(P'_t - P)$, $P_1^o \cup P_2^o = E(IN(P_t, c_{n+1}) \cup E(\beta(n+1)) = E(\beta(n))$.

Now $\alpha(n + 1) \cup E(\beta(n+1)) = \alpha(n) \cup E(IN(P_t, c_{n+1})) \cup E(\beta(n+1))$, which is equal to some result $\alpha(n) \cup E(\beta(n))$. Using the induction hypothesis, we know that any $\alpha(n) \cup E(\beta(n)) \in \bar{E}(P_1)$, so $\alpha(n + 1) \cup E(\beta(n+1)) \in \bar{E}(P_1)$ as well, which proves our induction step.

Proposition 3.24 The complexity of Algorithm 2 is $O(c \times |S| + \frac{|S|c}{z_c} \times g(\frac{|P_t| \times z_c}{|S|c}, \frac{|S|}{|P_t|}))$ where $S$ is the set of records in the input partition of records $P_t$, $c$ is the number of common conjuncts between $B_1$ and $B_2$, $z_c$ 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.

Proof. The complexity of Algorithm 2 can be computed by adding the cost for meeting partitions of the common conjuncts (Step 3) and the cost for running ER on the clusters in $M$ (Step 4). In Step 3, we perform $c - 1$ meets, which takes about $O(c \times |S|)$ time where the meet operation can be run in $O(|S|)$ time [22]. Given a record $r$, the probability of some other record $s$ clustering with $r$ is $\frac{z_c^{-1}}{|S| - 1}$ because each cluster has an average size of $z_c$. The probability for $s$ to be in the same cluster with $r$ for all the $c$ meeting partitions is thus $(\frac{z_c^{-1}}{|S| - 1})^c$, assuming that all conjuncts cluster records independently. As a result, the expected number of records to be clustered with $r$ in $M$ is $(|S| - 1) \times (\frac{z_c^{-1}}{|S| - 1})^c$. Hence, the average cluster size of $M$ is $1 + (|S| - 1) \times (\frac{z_c^{-1}}{|S| - 1})^c \approx \frac{|S|}{z_c}$ records. The expected number of clusters in $M$ is thus approximately $\frac{|S|}{z_c}$.

Each $\{c' \in P_t | c' \subseteq c\}$ (where $c \in M$) has on average $\frac{|P_t|}{|S|c}$ clusters of $P_t$ where the average size of each cluster in $P_t$ is $\frac{|S|}{|P_t|}$. The complexity of Step 4 is thus $O\left(\frac{|S|}{z_c} \times g\left(\frac{|P_t| \times z_c}{|S|c}, \frac{|S|}{|P_t|}\right)\right)$. Hence, the total algorithm complexity is $O(c \times |S| + \frac{|S|c}{z_c} \times g\left(\frac{|P_t| \times z_c}{|S|c}, \frac{|S|}{|P_t|}\right))$.

While Algorithm 2 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 $P_t$ (i.e., $\{c' \in P_t | c' \subseteq c\}$)’s (Step 4) rather than on the entire partition $P_t$. The rule evolution algorithm for ER algorithms that are only $\mathcal{RM}$ is identical to Algorithm 2 except for Step 4, where we can no longer process subsets of $P_t$ independently. However, we can still run Step 4 efficiently using global information. We revisit the sorted neighborhood ER algorithm ($\mathcal{SN}$) in Section 3.2. Recall that the first step of $\mathcal{SN}$ 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 $\{c' \in P_t | c' \subseteq c\} (c \in M)$ using the same window size $W$ as long as we also use the global sort 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 \( B_1 = p_{name} \land p_{zip} \), \( B_2 = p_{name} \land p_{phone} \), and the initial set \( P_1 = \{ \{ r_1 \}, \{ r_2 \}, \{ r_3 \}, \{ r_4 \}, \{ r_5 \} \} \). 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 \( E(P_1, p_{name}) = \{ \{ r_1, r_3, r_5 \}, \{ r_2 \}, \{ r_4 \} \} \) because \( r_1 \) and \( r_5 \) matched when the window covered \([r_1, r_2, r_3] \) and \( r_2 \) and \( r_3 \) 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 2 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 \( B_2(r_1, r_3) = \text{false} \), \( B_2(r_3, r_5) = \text{false} \), and \( B_2(r_1, r_5) = \text{true} \), 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 \( P_1 \) (see Section 6).

Algorithm 3 performs rule evolution for ER algorithms that only satisfy the \( \mathcal{GI} \) property. Algorithm 3 is identical to Algorithm 2 except that Step 4 is replaced with the code \( \text{return } E(\bigcup_{c \in M} E(\{c' \in P_1 | c' \subseteq c \}, B_2), B_2) \). 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 \( \{c' \in P_1 | c' \subseteq c \} \) 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 \( E(P_1, B_2) \). Using the same example for Algorithm 2, 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 3 efficient for certain ER algorithms. First, each cluster in \( E \) 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 \( P \). 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 6 that the \( HC_R \) algorithm indeed benefits from Algorithm 3.

The complexity of Algorithm 3 can be computed by adding the cost for meeting partitions and the cost for running ER on clusters. In comparison to Algorithm 2, the additional cost is the outermost ER operation in Step 4. In practice, Algorithm 3 is slower than Algorithm 2, but can still be faster than running the ER algorithm \( E \) without rule evolution.

**Proposition 3.25** Algorithm 3 correctly returns an ER result \( P_o \in \tilde{E}(P_1, B_2) \).

**Proof.** Suppose \( M = \{c_1, c_2, \ldots, c_{|M|}\} \). For this proof, we denote \( \{c' \in P_1 | c' \subseteq c \} \) as \( \text{IN}(P_1, c) \). We also omit \( B_2 \) from each \( E(P, B_2) \) expression for brevity (\( B_2 \) is not used in this proof). We define the following notations: \( \alpha(k) = \bigcup_{c \in M \setminus \{c_1, \ldots, c_k\}} \text{IN}(P_1, c) \) and \( \beta(k) = \bigcup_{c \in \{c_1, \ldots, c_k\}} \text{IN}(P_1, c) \). To avoid confusion with the initial set of clusters \( P_1 \), we use \( P'_1 \) instead of \( P_1 \) when using Definition 3.5 later in this proof. To prove that \( P_o = E(\alpha(0)) \in \tilde{E}(P_1) = \tilde{E}(\beta(|M|)) \), we prove the more general statement that \( P_o \in \tilde{E}(\alpha(k) \cup \beta(k)) \) for \( k \in \{0, \ldots, |M|\} \). Clearly, if our general statement holds, we can show that \( P_o \in \tilde{E}(\beta(|M|)) = \tilde{E}(P_1) \) by setting \( k = |M| \).

**Base case:** We set \( k = 0 \). Then \( P_o = E(\alpha(0)) \in \tilde{E}(\alpha(0)) = \tilde{E}(\alpha(0) \cup \beta(0)) \).

**Induction:** Suppose that our statement holds for \( k = n \), i.e., \( P_o = E(\alpha(0)) \in \tilde{E}(\alpha(n) \cup \beta(n)) \). We want to show that the same expression holds for \( k = n + 1 \) where \( n + 1 \leq |M| \). We use the \( \mathcal{GI} \) property by setting \( P = \text{IN}(P_1, c_{n+1}) \) and \( P'_1 = \alpha(n + 1) \cup \beta(n) \). The first condition \( P \subseteq P'_1 \) is satisfied because \( \beta(n + 1) \) contains \( P \). We then set \( P^1_o = E(P) = E(\text{IN}(P_1, c_{n+1})) \) and \( P^2_o = E(P^1_o \cup (P'_1 - P)) = E(E(\text{IN}(P_1, c_{n+1})) \cup \alpha(n + 1) \cup \beta(n)) = E(\alpha(n) \cup \beta(n)) \). The \( \mathcal{GI} \) property tells us that \( P^2_o \in \tilde{E}(P'_1) = \tilde{E}(\alpha(n + 1) \cup \beta(n + 1)) \). Thus, any \( E(\alpha(n) \cup \beta(n)) \) is in \( \tilde{E}(\alpha(n + 1) \cup \beta(n + 1)) \). Using our induction hypothesis, we conclude that \( P'_0 \in \tilde{E}(\alpha(0)) \in \tilde{E}(\alpha(n) \cup \beta(n)) \subseteq \tilde{E}(\alpha(n + 1) \cup \beta(n + 1)) \).
4 Materialization Strategies

Until now, we have used a general strategy for rule materialization where we materialize on each conjunct. 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 \( p_1 \land p_2 \land p_3 \) 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 \( p_1 \land p_2 \land p_3 \), then we can materialize on \( p_1, p_1 \land p_2, \) and \( p_1 \land p_2 \land p_3 \). If the ER algorithm is both \( RM \) and \( CF \), then the ER result of \( p_1 \land p_2 \) can be computed efficiently from the ER result of \( p_1 \), and the ER result of \( p_1 \land p_2 \land p_3 \) from that of \( p_1 \land p_2 \).

5 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 3.2. Finally, we provide a model on how the distance comparison rule can evolve and present our rule evolution techniques.

5.1 Distance-based Clustering Model

In the distance-based clustering model, records are clustered based on their relative distances instead of 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 3. For example, the distance between two records differs for each ER algorithm. In hierarchical clustering that returns a non-negative distance 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 2.1.

**Definition 5.1** 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 that \( D_1(r,s).min \geq 1 \)).

5.2 Properties

We use properties analogous to \( RM, CF, GI \), and \( OT \) from Section 3.2 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 5.1 instead of Definition 2.1 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 \( HC_{DC} \).

\( HC_{DS} \) The Single-link Hierarchical Clustering algorithm [12, 20] \((HC_{DS})\) 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 \( T \), \( \{ r_1, r_2, r_3 \} \) 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.

Proposition 5.2 shows that the \( HC_{DS} \) algorithm is \( RM, CF, GI \), and \( OT \). As a result, the \( HC_{DS} \) algorithm can use Algorithm 2 (with minor changes; see Section 5.3) for rule evolution.

**Proposition 5.2** The extended \( HC_{DS} \) algorithm is \( RM, CF, GI \), and \( OT \).

**Proof.** We first define the notation of connectedness for \( HC_{DS} \). Two records \( r \) and \( s \) are connected under \( D, T \), and \( P_i \) if there exists a sequence of records \([ r_1 (= r), \ldots, r_n (= s) ] \) where for each pair \((r_i, r_{i+1})\) in the path, either \( D(r_i,r_{i+1}).min \leq T \) or \( \exists c \in P_i \) s.t. \( r_i \in c, r_{i+1} \in c \). Notice that connectedness is “transitive,” i.e., if \( r \) and \( s \) are connected and \( s \) and \( t \) are connected, then \( r \) and \( t \) are also connected.

We now prove the following Lemma.

**Lemma 5.3** Two records \( r \) and \( s \) are connected under \( D, T \), and \( P_i \) if and only if \( r \) and \( s \) are in the same cluster in \( E(P_i, D) \) using the \( HC_{DS} \) algorithm.

**Proof.** The proof is identical to the proof in Section 3.8, except that we use \( D \) and \( T \) instead of \( B \) for comparing records.

We prove that the extended \( HC_{DS} \) algorithm is \( RM \) (and thus \( OT \)). Given the input partition \( P_i \) and two distance comparison rules \( D_1 \) and \( D_2 \) where \( D_1 \leq D_2 \), we want to show that for \( P_i^1 \in E(P_i, D_1) \) and \( P_i^2 \in E(P_i, D_2) \), \( P_o^1 \leq P_o^2 \). By Lemma 5.3, any pair of records \( r \) and \( s \) that get clustered together in \( P_o^1 \) are connected under \( D_1 \), and \( P_i \). If \( D_1 \leq D_2 \), we know that \( D_2(r_i,r_{i+1}).min \leq D_1(r_i,r_{i+1}).min \leq T \). Hence, \( r \) and \( s \) are also connected under \( D_2 \), and \( P_i \). By Lemma 5.3, \( r \) and \( s \) are guaranteed to be clustered in \( P_o^2 \) as well. As a result, \( P_o^1 \leq P_o^2 \). Since \( HC_{DS} \) is \( RM \), it is also \( OT \).

We prove that the extended \( HC_{DS} \) algorithm is \( GI \). Using Definition 3.5, suppose that the three conditions hold, i.e., \( P \subseteq P_i \), \( P_i^1 \in E(P_i, D) \), and \( P_o^2 \in E(P_o^2 \cup (P_i - P), D) \). Since \( HC_{DS} \) is \( GI \) (proved in previous paragraph), the ER results \( E(P_i, D) \) and \( E(P_o^2 \cup (P_i - P), D) \) are both unique.
We prove that \( E(P_\ell, D) \) and \( E(P_\ell^1 \cup (P_i - P), D) \) are in fact the same partition. By Lemma 5.3, any pair \( r \) and \( s \) in \( E(P_\ell, D) \) are connected under \( D, T, \) and \( P_i \). As a result, there exists a sequence of records \( \{r_1 = r, \ldots, r_n = s\} \) where for each pair \((r_i, r_{i+1})\) in the path, either \( D(r_i, r_{i+1}).min \leq T \) or \( \exists \in P \) s.t. \( r_i \in c, r_{i+1} \in c \). The fact that \( D(r_i, r_{i+1}).min \leq T \) does not change when evaluating \( E(P_\ell^1 \cup (P_i - P), D) \). If \( \exists \in P \) s.t. \( r_i \in c, r_{i+1} \in c \), then \( \exists \in P_\ell^1 \cup (P_i - P) \) s.t. \( r_i \in c, r_{i+1} \in c \) because none of the clusters in \( P_i \) are un-merged in \( P_\ell^1 \cup (P_i - P) \). Hence, \( r_i \) and \( r_{i+1} \) are connected in \( E(P_\ell^1 \cup (P_i - P), D) \), which means that \( r \) and \( s \) are connected in \( E(P_\ell^1 \cup (P_i - P), D) \). Thus, \( r \) and \( s \) are clustered together by Lemma 5.3.

Conversely, suppose that \( r \) and \( s \) are clustered together in \( E(P_\ell^1 \cup (P_i - P), D) \). Then by Lemma 5.3, \( r \) and \( s \) are connected under \( D, T, \) and \( P_\ell^1 \). As a result, there exists a sequence of records \( \{r_1 = r, \ldots, r_n = s\} \) where for each pair \((r_i, r_{i+1})\) in the path, either \( D(r_i, r_{i+1}).min \leq T \) or \( \exists \in P^1 \) s.t. \( r_i \in c, r_{i+1} \in c \). The fact that \( D(r_i, r_{i+1}).min \leq T \) does not change when evaluating \( E(P_\ell, D) \). If \( \exists \in P^1 \cup (P_i - P) \) s.t. \( r_i \in c, r_{i+1} \in c \), then \( \exists \in P^1 = E(P, D) \) s.t. \( r_i \in c, r_{i+1} \in c \) or \( \exists \in P_i - P \) s.t. \( r_i \in c, r_{i+1} \in c \). In the latter case, we know that \( \exists \in P \) s.t. \( r_i \in c, r_{i+1} \in c \) is true. In the former case, we know by Lemma 5.3 that \( r_i \) and \( r_{i+1} \) are connected under \( D, T, \) and \( P_i \). As a result, \( r_i \) and \( r_{i+1} \) are also connected under \( D, T, \) and \( P_i \) because \( P \subseteq P_i \). Combining the former and latter cases, \( r_i \) and \( r_{i+1} \) are always connected under \( D, T, \) and \( P_i \). Hence, \( r \) and \( s \) are also connected under \( D, T, \) and \( P_i \). By Lemma 5.3, \( r \) and \( s \) must be clustered together in \( E(P_\ell, D) \). Hence, we have proved that \( E(P_\ell, D) \) and \( E(P_\ell^1 \cup (P_i - P), D) \) are the same.

Using the third condition \( P_\ell^2 \in \tilde{E}(P_\ell^1 \cup (P_i - P), D) \), we know that \( P_\ell^2 \in \tilde{E}(P_\ell, D) \), so the \( HC_{DS} \) algorithm is \( GT \).

Finally, we prove that the extended \( HC_{DS} \) algorithm is \( CF \). Given the four partitions \( P, P_1, P_1^1, P_\ell^2 \), suppose that the four conditions in Definition 3.4 are satisfied. That is, \( P \subseteq P_i, \forall \in \tilde{E}(P_\ell, D), P_\ell \leq \{ \cup_{c P \cup P_i \cup P - P} c \}, P_1^1 \in \tilde{E}(P, D), \) and \( P_\ell^2 \in \tilde{E}(P_i - P, D) \). Since \( HC_{DS} \) returns a unique solution, there is exactly one \( P \in \tilde{E}(P_i, D) \). Suppose that \( E(P, D) \) generates a sequence of cluster mergers \( M_1, M_2, \ldots \) where each \( M \) involves a merge of two clusters. Since \( P \leq \{ \cup_{c P \cup P_i \cup P - P} c \}, \) we can split the sequence into merges \( M_1^1, M_2^1, \ldots \) that only involve clusters in \( P \) and merges \( M_1^2, M_2^2, \ldots \) that only involve clusters in \( P_i - P \). We can run the first batch of merges \( M_1^1, M_2^1, \ldots \) to produce a possible result of \( E(P, D) \) and run the second batch of merges \( M_1^2, M_2^2, \ldots \) to produce a possible result of \( E(P_i - P, D) \). Since \( HC_{DS} \) returns a unique solution, both ER results \( E(P, D) \) and \( E(P_i - P, D) \) are in fact unique and thus are equal to \( P_\ell^1 \) and \( P_\ell^2 \), respectively. The union of \( P_\ell^1 \) and \( P_\ell^2 \) is equivalent to the result of running the merges \( M_1, M_2, \ldots \), i.e., \( E(P, D) \). Hence, \( P_\ell^1 \cup P_\ell^2 \in \tilde{E}(P_\ell, D) \).

**Proposition 5.4** The extended \( HC_{DC} \) algorithm is \( CF \), but not \( RM \) or \( OT \). As a result, the extended \( HC_{DC} \) algorithm cannot use Algorithms 2 or 3 for rule evolution.

Proof. We prove that the extended \( HC_{DC} \) algorithm is \( CF \). Given the four partitions \( P, P_1, P_1^1, P_\ell^2 \), suppose that the four conditions in Definition 3.4 are satisfied. That is, \( P \subseteq P_i, \forall \in \tilde{E}(P_\ell, D), P_\ell \leq \{ \cup_{c P \cup P_i \cup P - P} c \}, P_1^1 \in \tilde{E}(P, D), \) and \( P_\ell^2 \in \tilde{E}(P_i - P, D) \). We define a cluster merge \( M \) as two clusters merging. We also denote the distance of the two clusters merging during \( M \) as \( M.dist \). Suppose that the sequence of cluster mergers \( M_1, M_2, \ldots \) were used to generate \( P_\ell^1 \) while a sequence of cluster mergers \( M_1^2, M_2^2, \ldots \) were used to generate \( P_\ell^2 \). Notice that both sequences are sorted by their distances (i.e., by \( M.dist \)). We now construct the sequence of merges \( M_1, M_2, \ldots \) by merging the two sequences \( M_1, M_2, \ldots \) and \( M_1^2, M_2^2, \ldots \) sorted by distance. Running the sequence \( M_1, M_2, \ldots \) on \( P_i \) in fact generates a possible result of \( E(P_\ell, D) \). We sketch a proof by contradiction. Suppose that one of the merges \( M_i \) is incorrect, i.e., the two clusters being
merged in $M_i$ do not have the minimum distance among all pairwise distances between all clusters. First, we know that none of the cluster pairs that contain records in $P$ have distances less than $M_i.dist$ because all the merges $M_1^q, \ldots, M_a^q$ where $M_b^q.dist < M_i.dist$ and $M_b^q.i + 1.dist \geq M_i.dist$ have been performed. For similar reasons, none of the cluster pairs that contain records in $P_i \sim P$ have distances less than $M_i.dist$ either. Hence, the cluster pair with the minimum distance should be a pair of one cluster that consists of records in $P$ and another cluster that consists of records in $P_i \sim P$. However, by merging these two clusters, we are contradicting the second condition of Definition 3.4 where none of the records in $P$ are supposed to merge with any records in $P_i \sim P$. Hence, the merge sequence $M_1, M_2, \ldots$ indeed produces a possible result of $E(P, B)$. Since running the sequence $M_1, M_2, \ldots$ produces an equivalent result as running the sequence $M_1^q, M_2^q, \ldots$ and then running the sequence $P_1^q \cup P_2^q \in E(P, D)$.

We prove that the extended $HC_{DC}$ algorithm does not satisfy $OI$ and consequently not $RM$ using a counter example. Suppose that we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and a distance comparison rule where $D(r_1, r_2) = [2]$, $D(r_2, r_3) = [2]$, and $D(r_1, r_3) = [4]$. We are also given the threshold value $T = 3$. If $r_1$ and $r_2$ merge first, then the algorithm terminates without any more merges because the next minimum distance is 4 (i.e., the distance between $r_1, r_2$ and $r_3$), which exceeds $T$. Similarly, if $r_2$ and $r_3$ merge first, then the algorithm terminates without any more merges. Thus, $HC_{DC}$ returns two possible ER results $\{\{r_1, r_2\}, \{r_3\}\}$ or $\{\{r_1\}, \{r_2, r_3\}\}$, violating the $OI$ property and thus the $RM$ property.

Finally, we prove that the extended $HC_{DC}$ algorithm is not $GI$ using a counter example. Suppose that we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and a distance comparison rule where $D(r_1, r_2) = [2]$, $D(r_2, r_3) = [3]$, and $D(r_1, r_3) = [4]$. We are also given the threshold value $T = 3$. Then the ER result $E(P, D)$ is always $\{\{r_1, r_2\}, \{r_3\}\}$ because $r_1$ and $r_2$ merge first (having the shortest distance 2), and then no clusters match anymore. Using Definition 3.5, suppose we set $P = \{\{r_2\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. Then $P_1^o \in E(P, D)$ is always $\{\{r_2, r_3\}\}$ because $r_2$ and $r_3$ merge together. As a result, $P_2^o \in E(P_1^o \cup (P_i \sim P), D)$ is always $\{\{r_1\}, \{r_2, r_3\}\}$ because $r_1$ does not match with $\{r_2, r_3\}$ (having a distance of 4). Thus, $P_2^o \not\in E(P, D) = \{\{r_1, r_2\}, \{r_3\}\}$, violating the $GI$ property.

We have now completed the venn diagram in Figure 5, which shows the results of Propositions 5.2 and 5.4.

### 5.3 Rule Evolution

While we used the CNF structures of comparison rules to perform rule evolution in Section 3.4, 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 $D_i(r, s)$ changes by at most $f(D_i(r, s))$ where $f$ is a positive function that can be provided by a domain expert who knows how much $D_i$ 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, $D_i(r, s).max + f(D_i(r, s)) \geq D_2(r, s).max$ and $D_i(r, s).min - f(D_i(r, s)) \leq D_2(r, s).min$. As a practical example, suppose that $D_i$ 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 $D_i$ evolves to $D_2$, we can set $f = 10$. Or if the zip code and phone number distances are always within 20% of the $D_i$ distance, then $f = 0.2 \times D_i$.

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 $D_3$, $E(P, D_3)$, instead of the ER results for all the conjuncts in the first comparison rule. We also update Algorithm 2 in Section 3.4 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 $D_3$.

**Example** We illustrate rule evolution for the $HC_{DS}$ algorithm using the updated Algorithm 2. 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 \( f(d) = 0.1 \times d \), \( D_2 \) is defined as \( D_2(r_1, r_2) = \{1.8, 2.2\} \), \( D_2(r_2, r_3) = \{3.6, 4.4\} \), and \( D_3(r_1, r_3) = \{4.5, 5.5\} \). We then materialize the ER result \( M = E(P_1, D_3) \). 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 \( D_2(r_1, r_3).min = 4.5 \) 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 \( \bigcup_{c \in M} E(P_1, D_2) \) 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 \( E(P_1, D_2) \).

6 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 [24] 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 [21, ?] 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 6.7, 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 Section 6.2. 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 Section 6.2 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 6.1. Then in Sections 6.3 and 6.4, we discuss the CPU costs of ER evolution compared to a naïve approach (ignoring materialization costs, if any). In Section 6.5 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 6.6 we discuss total costs, including materialization and evolution.

6.1 Experimental Setting

We experiment on a comparison shopping dataset provided by Yahoo! Shopping and a hotel dataset provided by Yahoo! Travel. We evaluated the following ER algorithms: \( SN, HC_B, HC_{BR}, ME, HC_{DS}, \) and \( HC_{DC} \). Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4GB of RAM.

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 [29] 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Data</th>
<th>Comparison rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Shopping</td>
<td>$B_{1}^{S} : p_{ti} \land p_{ca}$</td>
</tr>
<tr>
<td>Boolean</td>
<td>Hotel</td>
<td>$B_{2}^{S} : p_{ti} \land p_{pr}$</td>
</tr>
<tr>
<td>Boolean</td>
<td>Shopping</td>
<td>$B_{1}^{H} : p_{st} \land p_{ci} \land p_{zi} \land p_{na}$</td>
</tr>
<tr>
<td>Boolean</td>
<td>Hotel</td>
<td>$B_{2}^{H} : p_{st} \land p_{ci} \land p_{zi} \land p_{sa}$</td>
</tr>
<tr>
<td>Distance</td>
<td>Shopping</td>
<td>$D_{1}^{S} : \text{Jaro}_{ti}$</td>
</tr>
<tr>
<td>Distance</td>
<td>Hotel</td>
<td>$D_{2}^{S} : \text{Jaro}_{ti}$ changes randomly within 5%</td>
</tr>
<tr>
<td>Distance</td>
<td>Shopping</td>
<td>$D_{1}^{H} : \text{Jaro}<em>{na} + 0.05 \times \text{Equals}</em>{ci}$</td>
</tr>
<tr>
<td>Distance</td>
<td>Hotel</td>
<td>$D_{2}^{H} : \text{Jaro}<em>{na} + 0.05 \times \text{Equals}</em>{zi}$</td>
</tr>
</tbody>
</table>

ER and Rule Evolution Algorithms We experiment rule evolution on the following ER algorithms: $SN$, $HC_{B}$, $HC_{BR}$, $ME$, $HC_{DS}$, and $HC_{DC}$. (We do not experiment rule evolution using the join ER model, but focus on the clustering ER models.) Table 2 summarizes for each ER algorithm which section it was defined in and which rule evolution algorithm is used. The $HC_{DS}$ and $HC_{DC}$ distance-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 $RM$ property, we can still use Algorithm 2 to efficiently produce new ER results with small loss in accuracy. Notice that, although $ME$ is $GL$, Algorithm 3 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 2 adjusted for the distance-based clustering model (see Section 5.3).
### Table 2. ER and rule evolution algorithms tested

<table>
<thead>
<tr>
<th>ER algorithm</th>
<th>Section</th>
<th>Rule evolution algorithm used</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN</td>
<td>3.4</td>
<td>Algorithm for SN in Section 3.4</td>
</tr>
<tr>
<td>HC_B</td>
<td>3.4</td>
<td>Algorithm 3</td>
</tr>
<tr>
<td>HC_BR</td>
<td>3.4</td>
<td>Algorithm 2</td>
</tr>
<tr>
<td>ME</td>
<td>3.4</td>
<td>Algorithm 2</td>
</tr>
<tr>
<td>HC_DS</td>
<td>5.2</td>
<td>Algorithm 2 (for distance-based clustering)</td>
</tr>
<tr>
<td>HC_DC</td>
<td>5.2</td>
<td>Algorithm 2 (for distance-based clustering)</td>
</tr>
</tbody>
</table>

### 6.2 Evaluating IO costs

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 3. The operations are categorized depending on whether they are disk IO consuming operations or CPU time consuming operations.

### Table 3. Basic operations in blocking ER framework

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

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 4. 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 4. Decomposition of ER processes for one rule evolution

<table>
<thead>
<tr>
<th>ER process</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>$R_F, I, W_B, R_B, E, O, R_B, E, O$</td>
</tr>
</tbody>
</table>
The IO overhead of using rule evolution compared to the IO cost of the naïve approach can thus be written as \( R_M + W_M + R_B + 2 \times O \). Since the size of the materializations is usually much smaller than the size of the entire set of records (see Section 3.3), 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.

6.3 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 Section 6.1) given the properties of the ER algorithm. Table 5 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 5 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 5. ER algorithm and rule evolution runtimes

<table>
<thead>
<tr>
<th>ER algorithm</th>
<th>Sh1K</th>
<th>Sh2K</th>
<th>Sh3K</th>
<th>Ho1K</th>
<th>Ho2K</th>
<th>Ho3K</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SN )</td>
<td>0.094</td>
<td>0.152</td>
<td>0.249</td>
<td>0.012</td>
<td>0.027</td>
<td>0.042</td>
</tr>
<tr>
<td>( HC_B )</td>
<td>1.85</td>
<td>7.59</td>
<td>17.43</td>
<td>0.386</td>
<td>2.317</td>
<td>5.933</td>
</tr>
<tr>
<td>( HC_{BR} )</td>
<td>3.56</td>
<td>19.37</td>
<td>48.72</td>
<td>0.322</td>
<td>1.632</td>
<td>4.264</td>
</tr>
<tr>
<td>( HC_{DS} )</td>
<td>8.33</td>
<td>40.38</td>
<td>111</td>
<td>5.482</td>
<td>27.96</td>
<td>73.59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ratio of ER algorithm runtime to rule evolution runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SN )</td>
</tr>
<tr>
<td>( HC_B )</td>
</tr>
<tr>
<td>( HC_{BR} )</td>
</tr>
<tr>
<td>( HC_{DS} )</td>
</tr>
</tbody>
</table>

As one can see in Table 5, 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 \( \{ c' \in P | c' \subseteq c \} \) in Algorithm 2), Algorithm 3 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.

6.4 Common Rule Strictness

The key factor of the runtime savings in Section 6.3 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 5.3. 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 6 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 $p_i$ to produce 10 clusters of size 10, then the strictness is $\frac{10 \times \binom{10}{2}}{100} = 0.09$. 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 6 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.

Fig. 6. Degree of change impact on runtime, 3K shopping records

### 6.5 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. For example, when materializing for the $SN$ and $ME$ algorithms, the sorting of records is only done once. For the $HC_B$ and $HC_{BR}$ algorithms, we cache the merge information of records. For the $HC_{DS}$ and $HC_{DC}$ algorithms, the pairwise distances between records are only computed once. We can also compress the storage space needed by materializations by storing partitions of record IDs.
Table 6. Time overhead (ratio to old ER algorithm runtime) and space overhead (ratio to old ER result) of rule materialization, 3K records

<table>
<thead>
<tr>
<th>ER algorithm</th>
<th>Sho3K</th>
<th>Ho3K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time O/H</td>
<td>Space O/H</td>
</tr>
<tr>
<td>SN</td>
<td>0.52 (0.02)</td>
<td>0.28</td>
</tr>
<tr>
<td>HC_B</td>
<td>0.87 (0.04)</td>
<td>0.14</td>
</tr>
<tr>
<td>HC_BR</td>
<td>11 (3E-6)</td>
<td>0.14</td>
</tr>
<tr>
<td>HC_DS</td>
<td>0.44</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 6 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 \( E(P_i, B^S_1) \) using \( SN \). Hence, the total time to compute \( E(P_i, B^S_1) \) and materialize all the conjuncts of \( B^S_1 \) is \( 1+0.52 = 1.52 \) times the runtime for \( E(P_i, B^S_1) \) 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 \( B^S_1 \)) 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 \( E(P_i, B^H_1) \), 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 \( B^H_1 \)) 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 \( E(P_i, B^S_1) \) using \( SN \). The total required space is thus \( 1+0.28 = 1.28 \) times the memory space needed for \( E(P_i, B^S_1) \). The space overhead of materialization is small in general because we only store records by their IDs.

### 6.6 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.

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.

We measure the total runtimes of ER processes as defined in Section 6.2 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, and HCBR algorithms: $B_1 = p_{ca} \land p_{ti}$ (same as $B^S_1$ in Table 1) and $B_2 = p_{ca} \land p_{pr}$. 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 6. For the HCDS algorithm, we used $D^S_1$ and $D^S_2$ in Table 1. We 
used minhash signatures [17] 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 record, 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 7 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 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 HCBR and HCDS 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.

![Fig. 7. Scalability, 1M shopping records](image)

### 6.7 Without the Properties

So far, we have only studied scenarios where one or more of the properties needed for our rule evolution 
techniques held. We now consider a scenario where the necessary properties do not hold. In this 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 investigate the latter case and see if we can still return ER results with 
minimum loss in accuracy.

We experiment on two ER algorithms that do not satisfy the RM property and thus cannot use Algorithm 2: the ME and HCDC algorithms. While the ME algorithm is still GT and can thus use Algorithm 3, 
there is no runtime benefit because in ME all the records in $P_1$ are extracted and sorted again regardless of 
the clusters in $P_i$.

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 2 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 approach. 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.

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

<table>
<thead>
<tr>
<th>ER algorithm</th>
<th>Sh1K</th>
<th>Sh2K</th>
<th>Sh3K</th>
<th>Ho1K</th>
<th>Ho2K</th>
<th>Ho3K</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME</td>
<td>0.094</td>
<td>0.162</td>
<td>0.25</td>
<td>0.015</td>
<td>0.033</td>
<td>0.051</td>
</tr>
<tr>
<td>HC_{DC}</td>
<td>8.08</td>
<td>39.2</td>
<td>105</td>
<td>5.51</td>
<td>28.1</td>
<td>73.57</td>
</tr>
</tbody>
</table>

| 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|

7 Related work

Entity Resolution has been studied under various names including record linkage [24], merge/purge [16], deduplication [25], reference reconciliation [9], object identification [26], and others (see [10, 29, 13] for recent surveys). 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 [16, 4] and distance-based clustering [5, 20]. 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.

One of the recent challenges in information integration research is called Holistic Information Integration [14] 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. A number of works explore the problem of clustering data streams. Charikar et al. [6] propose incremental clustering algorithms that minimize the maximum cluster diameter given a stream of records. Aggarwal et al. [1] propose the ChuStream 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 topic of query optimization using materialized views, which has been studied extensively in the database literature [8, 11]. The focus of materialized views, 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. Our work is also related to constructing data cubes [15] 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.

8 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 independent 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 naive 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.

9 Acknowledgements

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

References


