Towards Interactive Debugging of Rule-based Entity Matching

Fatemah Panahi Wentao Wu AnHai Doan Jeffrey F. Naughton
Department of Computer Sciences, University of Wisconsin-Madison
{fatemeh, wentaowu, anhai, naughton}@cs.wisc.edu

ABSTRACT

Entity Matching (EM) identifies pairs of records referring to the same real-world entity. In practice, this is often accomplished by employing analysts to iteratively design and maintain sets of matching rules. An important task for such analysts is a "debugging" cycle in which they make a modification to the matching rules, apply the modified rules to a labeled subset of the data, inspect the result, and then perhaps make another change. Our goal is to make this process interactive by minimizing the time required to apply the modified rules. We focus on a common setting in which the matching function is a set of rules where each rule is in conjunctive normal form (CNF). We propose the use of "early exit" and "dynamic memoing" to avoid unnecessary and redundant computations. These techniques create a new optimization problem, and accordingly we develop a cost model and study the optimal ordering of rules and predicates in this context. We also provide techniques to reuse previous results and limit the computation required to apply incremental changes. Through experiments on six real-world data sets we demonstrate that our approach can yield a significant reduction in matching time and provide interactive response times.

1. INTRODUCTION

Entity matching (EM) identifies pairs of records that refer to the same real-world entity. For example, the records (Matthew Richardson, 206-453-1978) and (Matt W. Richardson, 453 1978) may refer to the same person, and (Apple, Cupertino CA) and (Apple Corp, California) refer to the same company. Entity matching is crucial for data integration and data cleaning.

Rule-based entity matching is widely used in practice [2, 13]. This involves analysts designing and maintaining sets of rules. Analysts typically follow an iterative debugging process, as depicted in Figure 1. For example, imagine an e-commerce marketplace that sells products from different vendors. When a vendor submits products from a new category, the analyst writes a set of rules designed to match these products with existing products. He or she then applies these rules to a labeled subset of the data and waits for the results. If the analyst finds errors in the matching output, he or she will refine the rules and re-run them, repeating the above process until the result is of sufficiently high quality.

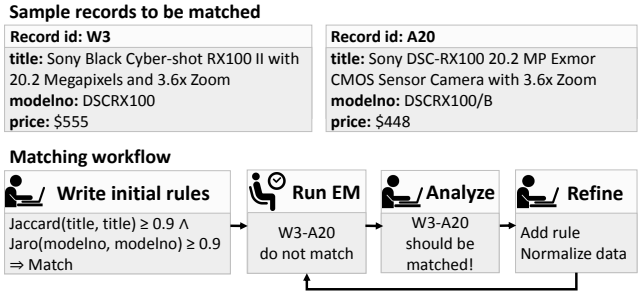

Figure 1: A typical matching workflow for analysts.

Our goal is to make this process interactive. This naturally introduces two challenges:

- *Efficiency*. The time that an analyst is idle in the "Run EM" step has to be short. Research shows that when interacting with software, if the response time is greater than one second, the analyst's flow of thought will be interrupted, and if it is greater than 10 seconds, the user will turn their attention to other tasks [12]. Therefore, it is imperative to reduce the idle "waiting" time as much as possible.
- *Maintainability*. The refinement made by the analyst is conceivably incremental. It is therefore desirable for an interactive rule-based entity matching system to maintain matching states between consecutive runs of "Run EM." Although a "stateless" system is easier to implement, e.g., it could just rerun the whole matching algorithm again from the scratch upon each refinement of the rules, it is clearly suboptimal in terms of both runtime efficiency and resource utilization.

In this paper, we take a first step towards interactive debugging of rule-based entity matching. Typically, rule-based entity matching is accomplished by evaluating a boolean matching function for each candidate record pair (for example, $B1$ in Figure 2). In this paper, we follow the approach we have encountered in practice in which this matching function is in Disjunctive Normal Form (DNF). Each disjunction is a rule, and each rule is a conjunction of a set of predicates that evaluate the similarity of two records on one or more attributes, using a similarity function (such as Jaccard or TF-IDF). For example, $\text{Jaccard}(a.name, b.name) > 0.7$ is a predicate, where $\text{Jaccard}(a.name, b.name)$ is a feature. A record pair is a match if it matches at least one rule.

As was pointed out by Benjelloun et al. [2], and confirmed in our experiments, computing similarity function values dominates the matching time. In view of this, our basic idea is to minimize

the number of similarity function computations as the analyst defines new rules and/or refines existing rules. Specifically, we save all computed similarity function values in memory to avoid redundant computation. We then exploit natural properties of DNF/CNF rule sets that enable "early exit" evaluation to eliminate the need for evaluating many rules and/or predicates for a given candidate pair of records. Moreover, we use "dynamic memoing": we only compute (and save the result of) a feature (i.e., a similarity score) if that predicate result is required by the matching function. (Because of early exit, not all features need to be computed.) This "lazy feature computation" strategy can thus save significant computation cost when there are many possible features but only a few of them are really required by the rule set/data set under consideration.

Although techniques such as "early exit" and "dynamic memoing" are straightforward and ubiquitous in computer science, their application in the context of rule-based entity matching raises an interesting, challenging issue: different evaluation orders of the predicates and rules may lead to significant differences in computational cost. It is then natural to ask the question of optimal ordering of predicates and rules. We further study this problem in detail. We show that the optimization problem under our setting is NP-hard, and we propose two greedy solutions based on heuristic optimization criteria. In our experiments with six real-world datasets, we show that the greedy solutions can indeed produce orderings that significantly reduce runtime compared to random ones.

So far, we have been focusing on the "efficiency" aspect of interactive entity matching. Since the elements (e.g., features, predicates, or rules) involved in matching change frequently as the analyst iteratively refines the rule set, the "maintainability" aspect is of equal importance. We therefore further develop incremental matching techniques to avoid rerunning matching from scratch after each change. Specifically, we discuss four fundamental cases: add/remove a predicate and add/remove a rule. We show how easy it is to integrate incremental matching into our framework. We also show via experiments that our incremental solutions can reduce matching time by orders of magnitude.

1.1 Related Work

Our work differs from previous work in several ways. Previous work on efficiently running rule-based entity resolution [2] assumes that each predicate is a black box, and thus memoing of similarity function results is not possible. In our experience in an industrial setting, these predicates are often not black boxes — rather, they are explicitly presented in terms of similarity functions, attributes, and thresholds. On the other hand, the traditional definition of the EM workflow, as described in [3, 5], assumes that *all* similarity values for *all* pairs are *precomputed* before the matching step begins. This makes sense in a batch setting in which a static matching function has been adopted, and the task is to apply this function to a set of candidate record pairs. However, in this paper we are concerned with the exploratory stage of rule generation, where at the outset the matching function is substantially unknown. In such settings the combinatorial explosion of potential attributes pairs, potential similarity functions, and candidate pairs can render such full precomputation infeasible.

Even in small problem instances in which full precomputation may be feasible, it can impose a substantial lag time between the presentation of a new matching task and the time when the analyst can begin working. This lag time may not be acceptable in practical settings where tens of matching tasks may be created every day [7] and the analyst wants to start working on high priority tasks immediately. Finally, during the matching process, an analyst may perform cleaning operations, normalization, and attribute extractions on the two input tables. The analyst might also introduce new similarity functions. In any of these situations, it is not possible to precompute all features a priori.

Table A

Id	Name	Street	Zip	Phone
a1	John	Dayton	54321	123-4567
a2	Bob	Regent	53706	121-1212

Table B

Id	Name	Street	Zip	Phone
b1	John	Dayton	54321	987-6543
b2	John	Bascom	11111	258-3524

Matching function evolution

B1: $(p1_{name} \wedge p_{zip}) \vee (p_{phone} \wedge p2_{name}) \rightarrow$
B2: $(p1_{name} \wedge p_{zip} \wedge p_{street}) \vee (p_{phone} \wedge p2_{name})$

Figure 2: Tables A, B to be matched and example matching functions. Function B1 evolves to B2.

Previous work on incrementally evaluating the matching function when the logic evolves assumes that we evaluate all predicates for all pairs and materialize the matching result for each predicate [15]. Because we use early exit, our information about the matching results for each predicate is not complete. As a result, this solution is not directly applicable in our setting.

In other related work, Dedoop (abbreviation for "Deduplication with Hadoop") [9] seeks to improve performance for general, large, batch entity matching tasks through the exploitation of parallelism available in Hadoop. By contrast, our work focuses on interactive response for rule-based entity matching where the matching function is composed of many rules that evolve over time. Exploring the application of parallelism as explored in Dedoop to our context is an interesting area for future work.

Our work is also related to [14]. In that work, the user provides a set of rule templates and a set of labeled matches and non-matches, the system then efficiently searches a large space of rules (that instantiate the rule templates) to find rules that perform best on the labeled set (according to an objective function). That work also exploits the similarities among the rules in the space. But it does so to search for the best set of rules efficiently. In contrast, we exploit rule similarities to support interactive debugging.

Finally, our work is related to the Magellan project, also at UW-Madison [10]. That project proposes to perform entity matching in two stages. In the development stage, the user iteratively experiments with data samples to find an accurate EM workflow. Then in the production stage the user executes that workflow on the entirety of data. If the user has decided to use a rule-based approach to EM, then in the development stage he or she will often have to debug the rules, which is the focus of this paper. This work thus fits squarely into the development stage of the Magellan approach.

In the following we start with a motivating example, describe our approach to try to achieve interactive response times, and present experimental results of our techniques on real world data sets.

2. MOTIVATING EXAMPLE

To motivate and give an overview of our approach, consider the following example. Our task is to match Table A and Table B shown in Figure 2 to find records that refer to the same person. We have four candidate pairs of records: $\{a1b1, a1b2, a2b1, a2b2\}$. Assume our matching function is $B1$. Intuitively, $B1$ says that if the name and zipcode of two records are similar, or if the phone number and name of two records are similar, then they match. Here $p1_{name}$ and $p2_{name}$, for example, compute the similarity score $Jaccard(a.name, b.name)$ and then compare this value to different thresholds, respectively, as we will see below.[1] For this ex-

[1]In practice we often compute Jaccard over the sets of q-grams of the two names, e.g., where $q = 3$; here for ease of exposition we

ample, $B1$ will return true for $a1b1$ and false for the rest of the candidate pairs.

A simple way to accomplish matching is to evaluate every predicate for every candidate pair. To evaluate a predicate, we compute the value of the similarity function associated with that predicate and compare it to a threshold. For the candidate pair $a2b1$, we would compute 4 similarity values.

This is unnecessary because once a predicate in a rule evaluates to false, we can skip the remaining predicates. Similarly, once a rule evaluates to true, we can skip the rest of the rules and therefore finish matching for that pair. We call this strategy "early exit," which saves unnecessary predicate evaluations. For instance, consider the candidate pair $a2b1$ again. Suppose that the predicate $p1_{name}$ is

$$\text{Jaccard}(a.name, b.name) \geq 0.9.$$

Since the Jaccard similarity of the two names is 0, $p1_{name}$ will return false for this candidate pair. Further assume that p_{phone} performs an equality check and thus returns 0 as well. We then do not need to evaluate p_{zip} and $p2_{name}$ to make a decision for this pair. Therefore, for this candidate pair, "early exit" reduces the number of similarity computations from 4 to 2.

Since the same similarity function may be applied to a candidate pair in multiple rules and predicates, we "memo" each similarity value once it has been computed. If a similarity function appears in multiple predicates, only the first evaluation of the predicate incurs a computation cost, while subsequent evaluations only incur (much cheaper) lookup costs. We call this strategy "dynamic memoing." Continuing with our example, suppose $p2_{name}$ is

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

Then for $a1b2$ this predicate only involves a lookup cost.

When using *early exit* and *dynamic memoing*, different orders of the predicates/rules will make a difference in the overall matching cost. Once again consider the candidate pair $a2b1$. If we change the order of predicates in $B1$ to

$$(p1_{name} \wedge p_{zip}) \vee (p2_{name} \wedge p_{phone}),$$

the output of the matching function will not change. However, it reduces the matching cost to one computation for $p1_{name}$ plus one lookup for $p2_{name}$. This raises a novel optimization problem that we study in Section 5.

Finally, we take into account the fact that, as the matching function's logic evolves, the changes to the function are often *incremental*. We can then store results of a previous EM run, and as the EM logic evolves, use those to save redundant work for the next EM iterations. As an example, imagine the case where the matching function $B1$ evolves to $B2$. Since $B2$ is *stricter* than $B1$, we only need to evaluate p_{street} for the pairs that were matched by $B1$ to verify if they still match. For our example, this means that we only need to evaluate $B1$ for $a1b1$ among the four pairs.

3. PRELIMINARIES

The input to the entity matching (EM) workflow is two tables A, B with a set of records $\{a_1 \ldots a_n\}$, $\{b_1 \ldots b_m\}$ respectively. The goal of EM is to find all record pairs $a_i b_j$ that refer to the same entity. Given table A with m records and table B with n records, there are $m \times n$ potential matches. Even with moderate-size tables, the total number of potential matches could be very large. Many potential matches obviously do not match and can be eliminated from consideration easily. That is the purpose of a *blocking* step, which typically precedes a more detailed matching phase.

For example, suppose that each product has a category attribute (e.g., clothing or electronics). We can assume that products from different categories are non-matches. This reduces the task to finding matching products within the same category. We refer to the set of potential matches left after the blocking step as the *candidate record pairs* or *candidate pairs* in the rest of this paper.

Each candidate record pair is evaluated by a Boolean *matching function* B, which takes in two records and returns true or false. We assume that B is commutative, i.e.,

$$\forall a_i b_j, B(a_i, b_j) = B(b_j, a_i).$$

We assume that each matching function is in disjunctive normal form (DNF). We refer to each disjunct as a *rule*. For example, our matching function $B1$ (Figure 2) is composed of two rules.

Such a matching function is composed of only "positive" rules, as they say what matches, not what does not match. In our experience, this is a common form of matching function used in the industry. Reasons for using only positive rules include ease of rule generation, comprehensibility, ease of debugging, and commutativity of rule application.

Each rule is a conjunction of a set of *predicates*. Each predicate compares the value of a *feature* for a candidate pair with a threshold. A feature in our context is a similarity function computed over attributes from the two tables. Similarity functions can be as simple as exact equality, or as complex as arbitrary user-defined functions requiring complex pre-processing and logic.

The matching result is composed of the return value of the matching function for each of the candidate pairs. In order to evaluate the quality of matching, typically a sample of the candidate pairs is chosen and manually labeled as match or non-match based on domain knowledge. The matching results for the sample is then compared with the correct labels to get an estimate of the quality of matching (e.g., *precision* and *recall*).

4. EARLY EXIT + DYNAMIC MEMOING

In this section, we first briefly present the details of *early exit* and *dynamic memoing*. Although the ideas are pretty straightforward, we choose to describe them in an algorithmic way for clarity. The notation used in describing these algorithms will also be used throughout the rest of the paper when we discuss optimal predicate/rule ordering and incremental algorithms. To analyze the costs of various algorithms covered in this section, we further develop a cost model. It is also the basis for the next section when we study the optimal predicate/rule ordering problem.

4.1 Baselines

We study two baseline approaches in this subsection. In the following, for a given rule r, we use $\text{predicate}(r)$ and $\text{feature}(r)$ to denote the set of predicates and features r includes.

4.1.1 The Rudimentary Baseline

The first baseline algorithm simply evaluates every predicate in the matching function for every candidate pair. Each predicate is considered as a black box and any similarity value used in the predicate is computed from scratch. The results of the predicates (true or false) are then passed on to the rules, and the outputs of the rules passed on to the matching function to determine the matching status. Algorithm 1 presents the details of this baseline.

will assume that Jaccard scores are computed over the set of words of the two names.

Algorithm 1: The rudimentary baseline.

Input: B, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatch)

```
1  Let R be the CNF rules in B;
2  Mark all c ∈ C with U;
3  foreach c ∈ C do
4      foreach r ∈ R do
5          foreach p ∈ predicate(r) do
6              Evaluate p;
7          end
8          Evaluate r = ⋀_{p∈predicate(r)} p;
9      end
10     Mark c with M if B = ⋁_{r∈R} r is true;
11 end
```

4.1.2 The Precomputation Baseline

This algorithm precomputes all feature values involved in the predicates before performing matching. Algorithm 2 presents the details. As noted in the introduction, full precomputation may not be feasible or desirable in practice, but we present it here as a point of comparison. We store precomputed values as a hash table mapping pairs of attribute values to similarity function outputs.

Algorithm 2: The precomputation baseline.

Input: B, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatch)

```
1  Let R be the CNF rules in B;
2  Let F = ⋃_{r∈R} feature(r);
3  Let Γ = {(c, f, v)} be a |C| × |F| array that stores the value v
     of each f ∈ F for each c ∈ C;
4  foreach c ∈ C do
5      foreach f ∈ F do
6          Compute v and store (c, f, v) in Γ;
7      end
8  end
9  Run Algorithm 1 by looking up feature values from Γ when
     evaluating predicates;
```

4.2 Early Exit

Both baselines discussed above ignore the properties of the matching function B. Given that B is in DNF, if one of the rules returns true, B will return true. Similarly, because each rule in B is in CNF, a rule will return false if one of its predicate returns false. Therefore, we do not need to evaluate all the predicates and rules. Algorithm 3 uses this idea. The "breaks" in lines 8 and 12 are the "early exits" in this algorithm.

4.3 Dynamic Memoing

We can combine the precomputation of the second baseline with early exit. That is, instead of precomputing everything up front, we postpone the computation of a feature until it is encountered during matching. Once we have computed the value of a feature, we store it so following references of this feature only incur lookup costs. We call this strategy "dynamic memoing," or "lazy feature computation." Algorithm 4 presents the details.

4.4 Cost Modeling and Analysis

In this subsection, we develop simple cost models to use in rule and predicate ordering decisions studied in Section 5. In the following discussion, we use $\text{cost}(p)$ to denote the cost of evaluating a predicate p. Let $\mathcal{C}$ be the set of all candidate pairs. Moreover, let F be the set of all features involved in the matching function, and we use $\text{cost}(f)$ to denote the computation cost of a feature f. Furthermore, we use δ to represent the lookup cost.

Algorithm 3: Early exit.

Input: B, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatch)

```
1  Let R be the CNF rules in B;
2  Mark all c ∈ C with U;
3  foreach c ∈ C do
4      foreach r ∈ R do
5          r is true;
6          foreach p ∈ predicate(r) do
7              if p is false then
8                  r is false; break;
9              end
10         end
11         if r is true then
12             Mark c with M; break;
13         end
14     end
15 end
```

4.4.1 The Rudimentary Baseline

The cost of Algorithm 1 can be represented as:

$$C_1 = \sum_{c \in \mathcal{C}} \sum_{r \in \mathcal{R}} \sum_{p \in \text{predicate}(r)} \text{cost}(p).$$

In our running example in the introduction, the cost of making a decision for the pair $a1b2$ is then

$$\text{cost}(p1_{name}) + \text{cost}(p_{zip}) + \text{cost}(p_{phone}) + \text{cost}(p2_{name}).$$

4.4.2 The Precomputation Baseline

Suppose that each feature f appears $\text{freq}(f)$ times in the matching function. Then the cost of the precomputation baseline (Algorithm 2) is

$$C_2 = \sum_{c \in \mathcal{C}} \sum_{f \in F} (\text{cost}(f) + \text{freq}(f)\delta).$$

In our running example this means that, for pair $a1b2$ and matching function $B1$, we would need to precompute three similarity values and look up four. Note that this requires knowing $\text{cost}(f)$ — in our implementation, as discussed in our experimental results, we use an estimate of $\text{cost}(f)$ obtained by evaluating f over a sample of the candidate pairs.

4.4.3 Early Exit

To compute the cost of early exit (Algorithm 3), we further introduce the probability $\text{sel}(p)$ that the predicate p will return true for a given candidate pair (i.e., the *selectivity* of p). In our implementation, we use an estimate of $\text{sel}(p)$ obtained by evaluating p over a sample of the candidate pairs.

Given this estimate for $\text{sel}(p)$, suppose that we have a rule r with m predicates p_1, ..., p_m. The expected cost of evaluating r for a (randomly picked) candidate pair is then

$$\begin{aligned} \text{cost}(r) \quad = \quad & \text{cost}(p_1) + \text{sel}(p_1)\,\text{cost}(p_2) + \cdots \\ & + \text{sel}(\textstyle\bigwedge_{j=1}^{m-1} p_j)\,\text{cost}(p_m), \end{aligned} \tag{1}$$

Algorithm 4: Early exit with dynamic memoing.

Input: B, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ (M means a match and U means an unmatch)

```
Let R be the CNF rules in B;
Let Γ be the feature values computed; Γ ← ∅;
Mark all c ∈ C with U;
foreach c ∈ C do
    foreach r ∈ R do
        r is true;
        foreach p ∈ predicate(r) do
            Let f be the feature in p;
            if f ∉ Γ then
                Compute f; Γ ← Γ ∪ {f};
            else
                Read the value of f from Γ;
            end
            if p is false then
                r is false; break;
            end
        end
        if r is true then
            Mark c with M; break;
        end
    end
end
```

because we only need to evaluate p_j if p_1, ..., p_{j-1} are all evaluated to be true. Similarly, we can define the selectivity of the rule r as

$$\text{sel}(r) = \text{sel}(\bigwedge_{j=1}^{m} p_j).$$

Suppose that we have n rules r_1, ..., r_n. The expected cost of the early exit strategy (Algorithm 3) is then

$$\begin{aligned} C_3 &= \text{cost}(r_1) + (1 - \text{sel}(r_1))\,\text{cost}(r_2) + \cdots \\ &\quad + (1 - \text{sel}(\bigvee_{i=1}^{n-1} r_i))\,\text{cost}(r_n). \end{aligned}$$

4.4.4 Early Exit with Dynamic Memoing

The expected cost of early exit with dynamic memoing (Algorithm 4) can be estimated in a similar way. The only difference is that we need to further know the probability that a feature is present in the memo. Specifically, suppose that a feature can appear at most once in a rule. Let $\alpha(f, r_i)$ be the probability that a feature f is present in the memo after evaluating r_i. The expected cost of computing f when evaluating r_i is then

$$E[\text{cost}(f)] = (1 - \alpha(f, r_{i-1}))\,\text{cost}(f) + \alpha(f, r_{i-1})\delta. \quad (2)$$

The expected cost C_4 of Algorithm 4 is obtained by replacing all $\text{cost}(p)$'s in Equation 1 by their expected costs in Equation 2.

Let $\text{prev}(f, r_i)$ be the predicates in the rule r_i that appear before f. We then have

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1}))\,\text{sel}(\bigwedge_{p \in \text{prev}(f, r_i)} p) + \alpha(f, r_{i-1}).$$

Based on our assumption, different predicates in the same rule contain different features. If we further assume that predicates with different features are independent, it then follows that

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1})) \prod_{p \in \text{prev}(f, r_i)} \text{sel}(p) + \alpha(f, r_{i-1}).$$

Notation	Description
$\text{cost}(X)$	cost of X (X is a feature/predicate/rule)
δ	the lookup cost
$\text{freq}(f)$	frequency of feature f
$\text{predicate}(r)$	predicates of rule r
$\text{feature}(X)$	features of X (X can be a predicate/rule)
$\text{sel}(X)$	selectivity of X (X can be a predicate/rule)
$\text{prev}(f, r)$	features/predicates in rule r before feature f
$\text{predicate}(f, r)$	predicates in rule r that have feature f
$\text{reduction}(r)$	overall cost reduction by execution of rule r
$\text{cache}(f, r)$	chance that f is in the memo after running r

Table 1: Notation used in cost modeling and optimal rule ordering.

Note that the initial condition satisfies

$$\alpha(f, r_1) = \prod_{p \in \text{prev}(f, r_1)} \text{sel}(p).$$

We therefore have obtained an inductive procedure for estimating $\alpha(f, r_i)$ ($1 \le i \le n$). Clearly, $\alpha(f, r_i) = \alpha(f, r_{i-1})$ if $f \notin \text{feature}(r_{i-1})$. So we can focus on the rules that contain f.

5. OPTIMAL ORDERING

Our goal in this section is to develop techniques to order rule and predicate evaluation to minimize the total cost of matching function evaluation. This may sound familiar, and indeed it is — closely related problems have been studied previously in related settings (see, for example, [1, 8]). However, our problem is different and unfortunately more challenging due to the interaction of early exit evaluation with dynamic memoing.

5.1 Notation

Table 1 summarizes notation used in this section. Some of the notation has been used when discussing the cost models.

5.2 Problem Formulation

We briefly recap an abstract version of the problem. We have a set of rules $\mathcal{R} = \{r_1, ..., r_n\}$. Each rule is in CNF, with each clause containing exactly one predicate. A pair of records is a match if any rule in $\mathcal{R}$ evaluates to true. Therefore, $\mathcal{R}$ is a disjunction of rules:

$$R = r_1 \vee r_2 \vee \cdots \vee r_n.$$

Consider a single rule

$$r = p_1 \wedge p_2 \wedge \cdots \wedge p_m.$$

We are interested in the minimum expected cost of evaluating r with respect to different orders (i.e., permutations) of the predicates p_1, ..., p_m.

Given a specific order of the predicates, the expected cost of r can be expressed as

$$\begin{aligned} \text{cost}(r) &= \text{cost}(p_1) + \text{sel}(p_1)\,\text{cost}(p_2) + \cdots \\ &\quad + \text{sel}(\bigwedge_{j=1}^{m-1} p_j)\,\text{cost}(p_m). \end{aligned} \quad (3)$$

Similarly, given a specific order of the rules, the expected cost of evaluating R, as was in Section 4.4.3, is

$$\begin{aligned} \text{cost}(R) &= \text{cost}(r_1) + (1 - \text{sel}(r_1))\,\text{cost}(r_2) + \cdots \\ &\quad + (1 - \text{sel}(\bigvee_{i=1}^{n-1} r_i))\,\text{cost}(r_n). \end{aligned} \quad (4)$$

We want to minimize $\text{cost}(R)$.

5.3 Independent Predicates and Rules

The optimal ordering problem is not difficult when independence of predicates/rules holds. We start by considering the optimal ordering of the predicates in a single rule r. If the predicates are independent, Equation 3 reduces to

$$\begin{aligned}\text{cost}(r) &= \text{cost}(p_1) + \text{sel}(p_1)\,\text{cost}(p_2) + \cdots \\ &\quad + \text{sel}(p_1)\cdots\text{sel}(p_{m-1})\,\text{cost}(p_m).\end{aligned}$$

The following lemma is well known for this case (e.g., see Lemma 1 of [8]):

LEMMA 1. *Assume that the predicates in a rule r are independent. $\text{cost}(r)$ is minimized by evaluating the predicates in ascending order of the metric:*

$$\text{rank}(p_i) = (\text{sel}(p_i) - 1)/\,\text{cost}(p_i) \qquad (\textit{for } 1 \le i \le m).$$

We next consider the optimal ordering of the rules by assuming that the rules are independent. We have the following similar result.

THEOREM 1. *Assume that the predicates in all the rules are independent. $\text{cost}(R)$ is minimized by evaluating the rules in ascending order of the metric:*

$$\text{rank}(r_j) = -\frac{\text{sel}(r_j)}{\text{cost}(r_j)} = -\frac{\prod_{p\in\text{predicate}(r_j)}\text{sel}(p)}{\text{cost}(r_j)}.$$

Here $\text{cost}(r_j)$ is computed by using Equation 3 with respect to the order of predicates specified in Lemma 1.

PROOF. By De Morgan's laws, we have

$$R = r_1 \vee \cdots \vee r_n = \neg(\bar{r}_1 \wedge \cdots \wedge \bar{r}_n).$$

Define $r'_j = \bar{r}_j$ for $1 \le j \le n$ and $R' = \neg R$. It follows that

$$R' = r'_1 \wedge \cdots \wedge r'_n.$$

This means, to evaluate R, we only need to evaluate R', and then take the negation. Since R' is in CNF, based on Lemma 1, the optimal order is based on

$$\text{rank}(r'_j) = (\text{sel}(r'_j) - 1)/\,\text{cost}(r'_j) \qquad (\text{for } 1 \le j \le n).$$

We next compute $\text{sel}(r'_j)$ and $\text{cost}(r'_j)$. First, we have

$$\text{sel}(r'_j) = 1 - \text{sel}(r_j) = 1 - \prod\nolimits_{p\in\text{predicate}(r_j)}\text{sel}(p),$$

by the independence of the predicates. Moreover, we simply have $\text{cost}(r'_j) = \text{cost}(r_j)$, because we can evaluate r'_j by first evaluating r_j and then taking the negation. Therefore, it follows that

$$\text{rank}(r_j) = \text{rank}(r'_j) = -\frac{\text{sel}(r_j)}{\text{cost}(r_j)} = -\frac{\prod_{p\in\text{predicate}(r_j)}\text{sel}(p)}{\text{cost}(r_j)}.$$

This completes the proof of the theorem. □

Recall that in our implementation we compute feature costs and selectivity by sampling a set of record pairs and compute the costs and selectivities on the sample. So far, we have implicitly assumed that memoing is not used.

5.4 Correlated Predicates and Rules

We now consider the question when memoing is used. This introduces dependencies so Lemma 1 and Theorem 1 no longer hold.

Let us start with one single rule r. We introduce a *canonical form* of r by "grouping" together predicates that share common features. Formally, for a predicate p, let $\text{feature}(p)$ be the feature it refers to. Furthermore, define

$$\text{feature}(r) = \cup_{p\in\text{predicate}(r)}\{\text{feature}(p)\}.$$

Given a rule r and a feature $f \in \text{feature}(r)$, let

$$\text{predicate}(f, r) = \bigwedge\nolimits_{p\in\text{predicate}(r)\wedge\text{feature}(p)=f} p.$$

We can then write the rule r as

$$r = \bigwedge\nolimits_{f\in\text{feature}(r)} \text{predicate}(f, r). \tag{5}$$

Since we only consider predicates of the form $A \ge a$ or $A \le a$ where A is a feature and a is a constant threshold, it is reasonable to assume that each rule does not contain redundant predicates/features. As a result, each group $\text{predicate}(f, r)$ can contain at most one predicate of the form $A \ge a$ and/or $A \le a$. Based on this observation, we have the following simple result.

LEMMA 2. $\text{cost}(\text{predicate}(f, r))$ *is minimized by evaluating the predicates in ascending order of their selectivities.*

PROOF. Remember that $\text{predicate}(f, r)$ contains at most two predicates p_1 and p_2. Note that, the costs of the predicates follow the pattern c, c' if memoing is used, regardless of the order of the predicates in $\text{predicate}(f, r)$. Here c and c' are the costs of directly computing the feature or looking it up from the memo ($c > c'$). As a result, we need to decide which predicate to evaluate first. This should be the predicate with the lower selectivity. To see this, without loss of generality let us assume $\text{sel}(p_1) < \text{sel}(p_2)$. The overall cost of evaluating p_1 before p_2 is then

$$C_1 = c + \text{sel}(p_1)c',$$

whereas the cost of evaluating p_2 before p_1 is

$$C_2 = c + \text{sel}(p_2)c'.$$

Clearly, $C_1 < C_2$. This completes the proof of the lemma. □

Since the predicates in different groups are independent, by applying Lemma 1 we get the following result.

LEMMA 3. $\text{cost}(r)$ *is minimized by evaluating the predicate groups in ascending order of the following metric:*

$$\text{rank}(\text{predicate}(f, r)) = \frac{\text{sel}(\text{predicate}(f, r)) - 1}{\text{cost}(\text{predicate}(f, r))}.$$

Here $\text{cost}(\text{predicate}(f, r))$ is computed by using Equation 3 with respect to the order of predicates specified in Lemma 2.

Now let us move on to the case in which there are multiple rules whose predicates are not independent. Unfortunately, this optimization problem is in general NP-hard. We can prove this by reduction from the classic traveling salesman problem (TSP) as follows. Let the rules be vertices of a complete graph G. For each pair of rules r_i and r_j, define the cost $c(i, j)$ of the edge (r_i, r_j) to be the execution cost of r_j if it immediately follows r_i. Note that here we have simplified our problem by assuming that the cost of r_j only depends on its predecessor r_i. Under this specific setting, our problem of finding the optimal rule order is equivalent to seeking a Hamiltonian cycle with minimum total cost in G, which is NP-hard. Moreover, it is known that a constant-factor approximation algorithm for TSP is unlikely to exist unless P equals NP (e.g., see Theorem 35.3 of [4]). Therefore, in the following we seek heuristic approaches based on various greedy strategies.

Algorithm 5: A greedy algorithm based on expected costs of rules.

Input: $\mathcal{R} = \{r_1, ..., r_n\}$, a set of CNF rules
Output: $\mathcal{R}^\pi$, execution order of the rules

```
1  Let Q be a priority queue ⟨(cost(r), r)⟩ of the rules;
2  R^π ← ∅;
3  foreach r ∈ R do
4  |   Order predicate(r) according to Lemma 3;
5  |   Compute cost(r) based on this order;
6  |   Insert (cost(r), r) into Q;
7  end
8  while Q is not empty do
9  |   r_min ← ExtractMin(Q);
10 |   Add r_min into R^π;
11 |   foreach (cost(r), r) ∈ Q do
12 |   |   Update cost(r) by assuming that r immediately
   |   |     follows r_min;
13 |   end
14 end
15 return R^π;
```

5.4.1 Greedy Algorithms

We now need to further order the rules by considering the overhead that can be saved by memoing. By Lemma 3, the predicates in each rule can be locally optimally ordered. Note that each order of the rules induces a global order over the (bag of) predicates. However, the selectivities of the predicates are no longer independent, because predicates in different rules may share the same feature. Furthermore, the costs of predicates are no longer constants due to memoing. In fact, they even depend on the positions of the predicates in their global order. In other words, the costs of predicates depend on the order of the rules (recall the cost model in Section 4.4.4). Hence we are not able to apply Lemma 1 or Theorem 1 in this context.

Nonetheless, intuitively, a predicate should tend to have priority if it is very selective (returns true for very few pairs) and small cost, since it will eliminate many pairs cheaply. On the other hand, a rule should tend to have priority if it is not very selective (returns true for many candidate pairs) and small cost, since it contributes many matches cheaply. Our first algorithm then uses this intuition in a greedy strategy by picking the rule with the minimum expected cost. The details of this algorithm are presented in Algorithm 5. Note that when we update $\mathrm{cost}(r)$ at line 12, we use the cost model developed in Section 4.4.4, which considers the effect of memoing, by assuming that r will be the immediate successor of $r_{\min}$.

Algorithm 5 only considers the expected costs of the rules if they are the first to be run among the remaining rules. Some rules may have slightly high expected costs but significant long-term impact on overall cost reduction. Algorithm 5 does not consider this and thus may overlook these rules. We therefore further consider a different metric that is based on the rules that can be affected if a rule is executed. This gives our second greedy algorithm.

In the following, we use $\mathrm{reduction}(r)$ to represent the overall cost that can be saved by the execution of the rule r, and use $\mathrm{cache}(f, r)$ to represent the probability that a feature f is in the memo after the execution of r. For two features f_1 and f_2 in r, we write $f_1 < f_2$ if f_1 appears before f_2 in the order of predicate groups specified by Lemma 3. Following Section 4.4.4, we redefine $\mathrm{prev}(f, r)$ to be the *features* that appear before f in r, namely,

$$\mathrm{prev}(f, r) = \{f' \in \mathrm{feature}(r) \wedge f' < f\}.$$

If we write r as it is in Equation 5, then

$$\mathrm{sel}(\mathrm{prev}(f, r)) = \prod\nolimits_{f' \in \mathrm{prev}(f,r)} \mathrm{sel}(\mathrm{predicate}(f', r)) \quad (6)$$

is the selectivity of (conjunction of) the predicates appearing before f in r. Here we have abused notation because $\mathrm{prev}(f, r)$ is a set of features rather than a predicate. Basically, $\mathrm{sel}(\mathrm{prev}(f, r))$ is the chance that the feature f needs to be computed (by either direct computation or cache lookup) when executing r. We further define $\mathrm{prev}(r)$ to be the rule executed right before r. It then follows that

$$\begin{aligned}\mathrm{cache}(f, r) &= (1 - \mathrm{cache}(f, \mathrm{prev}(r)))\,\mathrm{sel}(\mathrm{prev}(f, r)) \\ &\quad + \mathrm{cache}(f, \mathrm{prev}(r)).\end{aligned}$$

Next, define $\mathrm{contribution}(r', r)$ to be the reduced cost of r' by executing the rule r before the rule r'. Define $\mathrm{contribution}(r', r, f)$ to be the reduced cost due to the feature f. Let $\mathrm{feature}(r', r) = \mathrm{feature}(r') \cap \mathrm{feature}(r)$. Clearly,

$$\mathrm{contribution}(r', r) = \sum\nolimits_{f \in \mathrm{feature}(r', r)} \mathrm{contribution}(r', r, f).$$

We now consider how to compute $\mathrm{contribution}(r', r, f)$. If we do not run r before r', the expected cost of evaluating f in r' is then

$$\begin{aligned}\mathrm{cost}_1(f, r') &= \mathrm{sel}(\mathrm{prev}(f, r'))\big[\,\mathrm{cache}(f, \mathrm{prev}(r))\delta \\ &\quad + (1 - \mathrm{cache}(f, \mathrm{prev}(r)))\,\mathrm{cost}(f)\big],\end{aligned}$$

whereas if we run r before r' the cost becomes

$$\begin{aligned}\mathrm{cost}_2(f, r') &= \mathrm{sel}(\mathrm{prev}(f, r'))\big[\,\mathrm{cache}(f, r)\delta \\ &\quad + (1 - \mathrm{cache}(f, r))\,\mathrm{cost}(f)\big].\end{aligned}$$

It then follows that

$$\begin{aligned}\mathrm{contribution}(r', r, f) &= \mathrm{cost}_1(f, r') - \mathrm{cost}_2(f, r') \\ &= \mathrm{sel}(\mathrm{prev}(f, r'))\Delta(\mathrm{cost}(f) - \delta),\end{aligned}$$

where $\Delta = \mathrm{cache}(f, r) - \mathrm{cache}(f, \mathrm{prev}(r))$.

Based on the above formulation, we have

$$\mathrm{reduction}(r) = \sum\nolimits_{r' \neq r} \mathrm{contribution}(r', r).$$

Our second greedy strategy simply picks the rule r that maximizes $\mathrm{reduction}(r)$ as the next rule to be executed. Algorithm 6 presents the details of the idea. It is more costly than Algorithm 5 because update of $\mathrm{reduction}(r)$ at line 21 requires $O(n)$ rather than $O(1)$ time, where n is the number of rules.

Note that the computations of $\mathrm{cost}(r)$ and $\mathrm{reduction}(r)$ are still based on local decisions, namely, the *immediate* effect if a rule is executed. The actual effect, however, depends on the actual ordering of all rules and cannot be estimated accurately without finishing execution of all rules (or, enumerating all possible rule orders).

5.4.2 Discussion

If we only employ early exit without dynamic memoing, the optimal ordering problem remains NP-hard when the predicates/rules are correlated. However, we can have a greedy 4-approximation algorithm [1, 11]. The difference in this context is that the costs of the predicates no longer depend on the order of the rules. Rather, they are constants so approximation is easier. One might then wonder if combining early exit with precomputation (but not dynamic memoing) would make the problem even tractable, for now the costs of the predicates become the same (i.e., the lookup cost). Unfortunately, the problem remains NP-hard even for uniform costs when correlation is present [6].

Algorithm 6: A greedy algorithm based on expected overall cost reduction.

Input: $\mathcal{R} = \{r_1, ..., r_n\}$, a set of CNF rules
Output: $\mathcal{R}^\pi$, execution order of the rules

```
Let Q be a priority queue ⟨(reduction(r), r)⟩ of the rules;
R^π ← ∅;
foreach r ∈ R do
    Order predicate(r) according to Lemma 3;
end
foreach r ∈ R do
    reduction(r) ← 0;
    foreach r' ∈ R such that r' ≠ r do
        foreach f ∈ feature(r') do
            if f ∈ feature(r) then
                reduction(r) ←
                  reduction(r) + contribution(r', r, f);
            end
        end
    end
    Insert (reduction(r), r) into Q;
end
while Q is not empty do
    r_max ← ExtractMax(Q);
    Add r_max into R^π;
    foreach (reduction(r), r) ∈ Q do
        Update reduction(r) by assuming that r immediately
          follows r_max;
    end
end
return R^π;
```

5.4.3 Optimization: Check Cache First

We have proposed two greedy algorithms for ordering rules and predicates in each rule. The order is computed before running any rule and remains the same during matching. However, the greedy strategies we proposed are based on the "expected" rather than actual costs of the predicates. In practice, once we start evaluating the rules, it becomes clear that a feature is in the memo or not. One could then further consider dynamically adjusting the order of the remaining rules based on the current content of the memo. This incurs nontrivial overhead, though: we basically have to re-run the greedy algorithms each time we finish evaluating a rule. So in our current implementation we do not use this optimization. Nonetheless, we are able to reorder the predicates inside each rule at runtime based on the content of the memo. Specifically, we first evaluate predicates for which we have their features in the memo, and we still rely on Lemma 3 to order the remaining predicates.

5.5 Putting It All Together

The basic idea in this section is to order the rules such that we can decide on the output of the matching function with lowest computation cost for each pair. To order the rules we use a small random sample of the candidate pairs and estimate feature costs and selectivities for each predicate and rule. We then use Algorithm 5 or Algorithm 6 to order the rules. These two algorithms consider two different factors that affect the overall cost: 1) the expected cost of each rule, and 2) the expected overall cost reduction that executing this rule will have if the features computed for this rule are repeated in the following rules. We further evaluate the performance of both algorithms in our experiments.

6. INCREMENTAL MATCHING

So far we have discussed how to perform matching for a fixed set of fixed rules. We now turn to consider incremental matching in the context of an evolving set of rules.

6.1 Materialization Cost

To perform incremental matching, we materialize the following information during each iteration:

- For each pair: For each feature that was computed for this pair, we store the calculated score. Note that because we use lazy feature computation, we may not need to compute all feature values.
- For each rule: Store all pairs for which this rule is true.
- For each predicate: Store all pairs for which this predicate evaluated to false.

We show in our experiments that if we use straightforward techniques such as storing bitmaps of pairs that pass rules or predicates, the total memory needed to store this information for our data sets is less than 1GB.

6.2 Types of Matching Function Changes

An analyst often applies a single change to the matching function, re-runs EM, examines the output, then applies another change. We study different types of changes to the matching function and present our incremental matching algorithm for each type.

6.2.1 Add a Predicate / Tighten a Predicate

Algorithm 7: Add a predicate.

Input: $\mathcal{R}$, the set of CNF rules; r, the rule that was changed; p, the predicate added to r

```
Let M(r) be the previously matched pairs by r;
Let X be the unmatched pairs by p; X ← ∅;
foreach c ∈ M(r) do
    if p returns false for c then
        X ← X ∪ {c};
    end
end
Let R' be the rules in R after r;
foreach c ∈ X do
    Mark c as an unmatch;
    foreach r' ∈ R' do
        if r' returns true for c then
            Mark c as a match; break;
        end
    end
end
```

If a matching result contains pairs that should not actually match, the analyst can make the rules that matched such a pair more "strict" by either adding predicates, or making existing predicates more strict. For example, consider the following predicate

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

We can make this more strict by changing it to

$$\text{Jaccard}(a.name, b.name) \geq 0.8.$$

In this case, we can obtain the new matching results incrementally by evaluating this modified predicate only for the pairs that were

evaluated and matched by the rule we made stricter. Consider such a previously matched pair:

- If the modified predicate returns true, the pair is still matched.
- If the modified predicate returns false, the current rule no longer matches this pair. However, other rules in the matching function may match this pair, so we must evaluate the pair with the other rules until either a rule returns true or all rules return false.

We can use the same approach for adding a new predicate to a rule, because that can be viewed as making an empty predicate that always evaluates to true more strict. Algorithm 7 illustrates the procedure for adding a predicate.

6.2.2 Remove a Predicate / Relax a Predicate

Algorithm 8: Make a predicate less strict.

Input: r, the rule that was changed; p, the predicate of r that was made less strict
Let $U(p)$ be the pairs for which p returned false;
Let Y be the pairs p now returns true; $Y \leftarrow \emptyset$;
foreach $c \in U(p)$ **and** c *was an unmatch* **do**
  if p *returns* true *for* c **then**
    $Y \leftarrow Y \cup \{c\}$;
  end
end
foreach $c \in Y$ **do**
  Mark c as a match;
  foreach $p' \in \text{predicate}(r)$ **and** $p' \neq p$ **do**
    if p' *returns* false *for* c **then**
      Mark c as an unmatch; **break**;
    end
  end
end

In the case where pairs that should match are missing from the result, we might be able to fix the problem by either removing a predicate or making an existing predicate less strict. Consider again the predicate

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

We can make it less strict by changing it to

$$\text{Jaccard}(a.name, b.name) \geq 0.6.$$

In both cases, all pairs for which this predicate returned false need to be re-evaluated. Consider such a previously unmatched pair:

- If the new predicate is false, the pair remains unmatched.
- If the new predicate is true, we will evaluate the other predicates in the rule.[2] If any of these predicates returns false, then the pair remains a non-match. Otherwise, this rule will return true for this pair, and it will be declared a match.

Algorithm 8 illustrates the details of the procedure for updating the matching result after making a predicate less strict. Removing a predicate follows similar logic and is omitted for brevity.

6.2.3 Remove a Rule

[2]Note that, because we use the "check-cache-first" optimization, the order of the predicates within the rule is no longer fixed. In other words, different pairs may observe different orders. So we cannot just evaluate predicates that "follow" the changed one.

Algorithm 9: Remove a rule.

Input: $\mathcal{R}$, the set of CNF rules; r, the rule removed
Let $M(r)$ be the previously matched pairs by r;
Let $\mathcal{R}'$ be the rules in $\mathcal{R}$ after r;
foreach $c \in M(r)$ **do**
  Mark c as an unmatch;
  foreach $r' \in \mathcal{R}'$ **do**
    if r' *returns* true *for* c **then**
      Mark c as a match; **break**;
    end
  end
end

We may decide to remove a rule if it returns true for pairs that should not match. In such a case, we can re-evaluate the matching function for all pairs that were matched by this rule. Either another rule will declare this pair a match or the matching function will return false. Algorithm 9 illustrates this procedure.

6.2.4 Add a Rule

Algorithm 10: Add a rule.

Input: $\mathcal{R}$, the set of CNF rules; r, the rule added
Let $U(r)$ be the previously unmatched pairs by $\mathcal{R}$;
foreach $c \in U(r)$ **do**
  Mark c as a match;
  foreach $p \in \text{predicate}(r)$ **do**
    if p *returns* false *for* c **then**
      Mark c as an unmatch; **break**;
    end
  end
end

One way to match pairs that are missed by a current matching function is to add a rule that returns true for them. In this case, inevitably, all non-matched pairs need to be evaluated by this rule. However, note that only the newly added rule will be evaluated for the non-matched pairs, which can be substantial savings over re-evaluating all rules. Algorithm 10 demonstrates this procedure.

7. EXPERIMENTAL EVALUATION

In this section we explore the impact of our techniques on the performance of various basic and incremental matching tasks. We ran experiments on a Linux machine with eight 2.80 GHz processors (each with 8 MB of cache) and 8 GB of main memory. We implemented our algorithms in Java. We used six real-world data sets as described below.

7.1 Datasets and Matching Functions

We evaluated our solutions on six real-world data sets. One data set was obtained from an industrial EM team. The remaining five data sets were created by students in a graduate-level class as part of their class project, where they had to crawl the Web to obtain, clean, and match data from two Web sites. Table 2 describes these six data sets. For ease of exposition, and due to space constraints, in the rest of this section we will describe experiments with the first (and largest) data set. Experiments with the remaining five data sets show similar results and are therefore omitted.

We obtained the Walmart/Amazon data set used in [7] from the authors of that paper. The dataset domain is electronics items from

Data set	Source 1	Source 2	Table1 size	Table2 size	Candidate pairs	Rules	Used features	Total features
Products	Walmart	Amazon	2554	22074	291649	255	32	33
Restaurants	Yelp	Foursquare	3279	25376	24965	32	21	34
Books	Amazon	Barnes & Noble	3099	3560	28540	10	8	32
Breakfast	Walmart	Amazon	3669	4165	73297	59	14	18
Movies	Amazon	Bestbuy	5526	4373	17725	55	33	39
Video games	TheGamesDB	MobyGames	3742	6739	22697	34	23	32

Table 2: Real-world data sets used in the experiments.

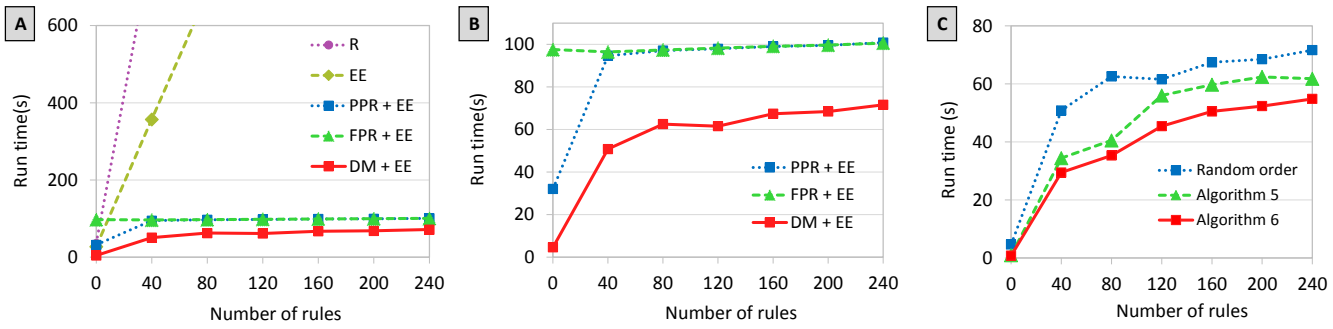

Figure 3: **(A)** Run time for different sizes of matching function for rudimentary baseline (R), early exit (EE), production precomputation baseline + early exit (PPR + EE), full precomputation baseline + early exit (FPR + EE), and dynamic memoing + early exit (DM + EE). **(B)** Zoom-in of A to compare methods that use precomputation/dynamic memoing. **(C)** Run time for different orderings of the set of rules/predicates: Random ordering, order by Algorithm 5, and order by Algorithm 6.

Function	Walmart	Amazon	µs
Exact Match	modelno	modelno	0.2
Jaro	modelno	modelno	0.5
Jaro Winkler	modelno	modelno	0.77
Levenshtein	modelno	modelno	1.22
Cosine	modelno	title	3.37
Trigram	modelno	modelno	4.79
Jaccard	modelno	title	6.75
Soundex	modelno	modelno	8.77
Jaccard	title	title	10.54
TF-IDF	modelno	title	12.18
TF-IDF	title	title	18.92
Soft TF-IDF	modelno	title	21.89
Soft TF-IDF	title	title	66.06

Table 3: Computation costs for features in the products data set

Walmart.com and Amazon.com. After the blocking step, we have 291, 649 candidate pairs. Gokhale et al. [7] have generated the ground truth for these pairs.

We generated 33 features using a variety of similarity functions based on heuristics that take into account the length and type of the attributes. Table 3 shows a subset of these features and their associated average computation times. The computation times of features vary widely.

Using a combination of manual and semi-automatic approaches, analysts from the EM team that originally created the data set have created a total of 255 matching rules. We will use this rule set as a basis from which to create and evaluate a variety of matching functions. Figure 4 shows two sample rules for this data set.

$R1$ Jaro Winkler(m, m) $\geq 0.97 \wedge$ Jaro(m, m) ≥ 0.95
$\wedge$ Soft TF-IDF(m, t) < 0.28
$\wedge$ TF-IDF(m, t) $< 0.25 \wedge$ Cosine(t, t) ≥ 0.69

$R2$ Jaccard(t, t) $< 0.4 \wedge$ TF-IDF(t, t) < 0.55
$\wedge$ Soft TF-IDF(t, t) $\geq 0.63 \wedge$ Jaccard ≥ 0.34
$\wedge$ Levenshtein(m, m) < 0.72
$\wedge$ Jaro Winkler(m, m) < 0.05

Figure 4: Sample rules extracted from the random forest. m, t stand for modelno and title respectively.

7.2 Early Exit + Dynamic Memoing

Figure 3A shows the effect of early exit and precomputing (memoing) feature values on matching time as we use an increasingly larger rule set. For example, to generate the data point corresponding to 20 rules, we randomly selected 20 rules and measured the time to apply them to the data set. For each data point we report the average running time over three such random sets of rules.

We compare the run time for baseline, early exit, production precomputation + early exit, full precomputation + early exit, and dynamic memoing + early exit. For production precomputation, which we described as one of our baselines in Section 4.1.2, we assume that we know all the features used in the rules. We call this "production precomputation" because it is feasible only if the set of rules for matching is already finalized. In full precomputation, we know a superset of features that the analyst will choose from to make the rule set. In such a case, we may precompute values for features that will never be used. We compare these approaches with dynamic memoing + early exit proposed in this paper.

We can see that the rudimentary baseline has a very steep slope, and around 20 rules, it takes more than 10 minutes to complete. The early exit curve shows significant improvement over baseline, however, it is still slow compared to either the precomputation baseline or early exit with dynamic memoing. Figure 3B zooms in and shows the curves for the full and production precomputation baselines and dynamic memoing. We can see that using dynamic memoing + early exit can significantly reduce matching time.

In this subsection, we have not considered the optimal ordering problem, and we ran dynamic memoing with a random ordering of the rules and predicates in each rule. In the next subsection, we further study the effectiveness of our greedy algorithms on optimizing orderings of predicates/rules.

7.3 Optimal Ordering

Figure 3C shows runtime as we increase the number of rules for "dynamic memoing + early exit" with random ordering of predicates/rules, as well as that with orderings produced by the two greedy strategies presented in Algorithm 5 and Algorithm 6. Each data point was generated using the same approach described in the previous subsection. We used a random sample consisting of 1% of the candidate pairs for estimating feature costs and predicate selectivities. We can see that the orderings produced by both of these algorithms are superior to the random ordering.

We further observe that Algorithm 6 is faster than Algorithm 5, perhaps due to the fact that its decision is based on a global optimization metric that considers the overall cost reduction by placing a rule before other rules. As the number of rules increases, the impact is less significant, because most of the features have to be computed. Nonetheless, matching using Algorithm 6 is still faster even when we use 240 rules in the matching function.

7.4 Memory Consumption

We store the similarity values in a two dimensional array. We assign each pair an index based on their order in the input table. Similarly, we assign each feature a random order and an index based on the order. In the case of the precomputation baseline, this memo is completely filled with feature values before we start matching. In the case of dynamic memoing, we fill in the memo as we run matching and the analyst makes changes to the rule set. Therefore, the memory consumption of both approaches is the same. For this dataset, if we use all rules, the two-dimensional array takes 22 MB of space, which includes the space for storing the actual floats as well as the bookkeeping overhead for the array in Java. For incremental matching, we store a bitmap for each rule as well as for each predicate. In our implementation, we use a boolean array for each bitmap. For this dataset, we have 255 rules and a total of $1,688$ predicates. These bitmaps occupy 542 MB.

For our dataset, the two-dimensional array and bitmaps fit comfortably in memory. For a data set where this is not true, one could consider avoiding an array and using a hash-map for storing similarity values. Since we do not compute all the feature values, this would lead to less memory consumption, although the lookup cost for hash-maps would be more expensive.

7.5 Cost Modeling and Analysis

To illustrate accuracy of our cost models, in Figure 5A we compare the actual run time of "dynamic memoing + early exit" versus run time estimated by the cost model for random ordering of rules as well as rules ordered by Algorithm 6. The two curves follow each other closely.

To compute the selectivity of each predicate, we select a sample of the candidate pairs, evaluate each predicate for the pairs in the sample and compute the percentage of pairs that pass each predicate. In our experiments, we observed that using a 1% sample can give relatively accurate estimates of the selectivity, and increasing the sample size did not change the rule ordering in a major way. We used the same small sample approach to estimate feature costs.

Figure 5B shows the actual matching time when we use all the rules for the data set as we increase number of pairs. As we assumed in our cost modeling, the matching cost increases linearly as we increase number of pairs. Given this increase proportional to the number of pairs (which is itself quadratic in the number of input records), the importance of performance enhancing techniques to achieve interactive response times increases with larger data sets.

7.6 Incremental Entity Matching

Our first experiment examines the "add rule" change. Adding a rule can be expensive for incremental entity matching because we need to evaluate the newly added rule for all the unmatched pairs.

To test how incremental matching performs for adding a new rule, we conducted the following experiment. We start from an empty matching function without any rules. We then add the first rule to the matching function, run matching with this single-rule matching function, and materialize results. Next, we add the second rule and measure the time required for incremental matching. In general, we run matching based on k rules, and then run incremental matching for the $(k+1)$-th rule when it is added. We repeat this for $1 \leq k \leq 240$.

We consider two variations of incremental algorithm. In the *precomputation* variation, all the rules in the matching function are evaluated. Note that we use early exit and the optimization discussed in Section 5.4.3 with this variation to reduce unnecessary lookups. The second variation is *fully incremental*. In this case we not only lookup the stored feature values, but also only evaluate part of the matching function for the subset of candidate pairs that will be affected by this operation. In particular, for the "add rule" operation, all the non-matched pairs need to be evaluated by just the new rule that is added, and all the rules in the matching function do not need to be evaluated.

Figure 5C shows the results for the add-rule experiment. We can see that in the first iteration, both variations are slow. This is because there is no materialized result to use (i.e. the memo is empty). However, from the second iteration onwards we can see that the cost of the precomputation baseline steadily increases whereas the cost of fully incremental is mostly constant and significantly smaller than that of the precomputation baseline. This is because the precomputation baseline performs unnecessary lookups and evaluates all the rules in the matching function. The incremental approach just evaluates the newly added rule and thus it does not slow down as the number of rules increases.

In certain runs both of the variations experience a sudden increase in the running time. These are the cases in which the new rule requires many feature computations, because either there was a new feature, or the feature was not in the memo, and this feature was "reached" in the rule evaluation (it might not be reached, for example, if a predicate preceding the feature evaluates to false.)

Figure 6 shows run time of incremental EM for different changes to the matching function. To illustrate how the numbers were generated, assume that we want to measure the incremental run time for adding a predicate. We randomly selected 100 predicates, removed the predicate, ran EM and materialized the results, then added the predicate to the rule, and measured the run time. The rest of the numbers in the table were generated in a similar manner.

For tightening the thresholds, we randomly selected a predicate, and for that predicate we randomly chose one of the values in

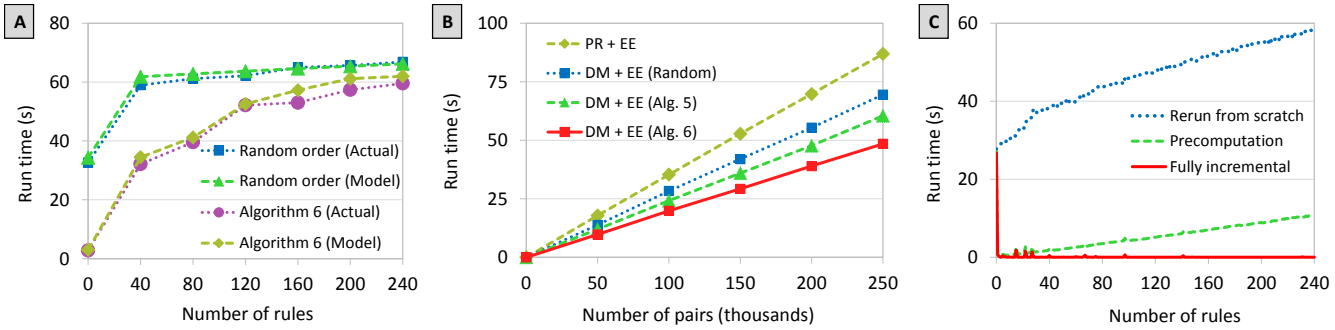

Figure 5: (A) Actual run time versus run time estimated by the cost model for random ordering of rules and rules ordered by Algorithm 6. **(B)** Run time as we increase number of pairs for production precomputation + early exit (PPR + EE), and dynamic memoing + early exit (DM + EE) for random ordering of rules, rules ordered by Algorithm 5, and ordered by Algorithm 6. **(C)** Run time with dynamic memoing + early exit as we add rules one by one to the matching function in three cases: 1) Rerun matching from scratch, 2) Precomputation: lookup memoed feature values but evaluate all rules 3) Fully incremental: lookup memoed feature values but only evaluate the newly added rule.

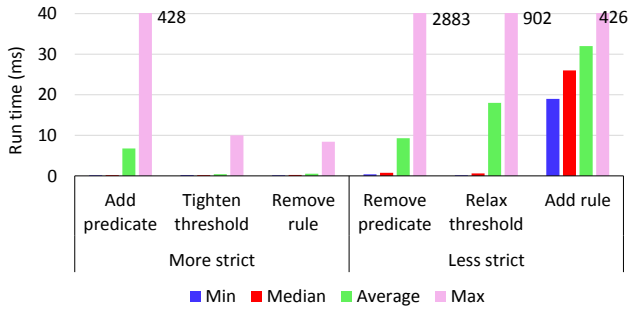

Figure 6: Incremental EM run time for different changes to the matching function that make it more/less strict.

$\{0.1, 0.2, 0.3, 0.4, 0.5\}$ that could be applied to the predicate. For example, assume that the predicate is $\text{Jaccard}(a.name, b.name) \geq 0.6$. To tighten the threshold, we add a random value to the threshold from $\{0.1, 0.2, 0.3, 0.4\}$, because adding 0.5 makes the threshold larger than 1. If the predicate uses a $\geq$ operation we add the value to the current threshold, and if it uses a $\leq$ operation we subtract the value from the current threshold. The procedure is similar for relaxing thresholds.

We can see that making the matching function more strict by adding a predicate, tightening the threshold, and removing a rule on average takes no more than about 6 milliseconds. On the other hand, making the function less strict could take up to 34 milliseconds on average. This cost is due to the fact that we may need to calculate new features for a fraction of candidate pairs.

8. CONCLUSIONS

We have focused on scenarios where an analyst iteratively designs a set of rules for an EM task, with the goal of making this process as interactive as possible. Our experiments with six real-world data sets indicate that "memoing" the results of expensive similarity functions is perhaps the single most important factor in achieving this goal, followed closely by the implementation of "early-exit" techniques that stop evaluation as soon as a matching decision is determined for a given candidate pair.

In the context of rule creation and modification it may not be desirable or even possible to fully precompute similarity function results in advance. Our just-in-time "memoing" approach solves this problem, dynamically storing these results as needed; however, the interaction of the on-demand memoing and early-exit evaluation creates a novel rule and predicate ordering optimization problem. Our heuristic algorithms to solve this problem provide significant further reductions in running times over more naive approaches.

Finally, in the context of incremental rule iterative development, we show that substantial improvements in running times are possible by remembering the results of previous iterations and on the current iteration only computing the minimal delta required by a given change.

From a broader perspective, this work joins a small but growing body of literature which asserts that for matching tasks, there is often a "human analyst in the loop," and rather than trying to remove that human, attempts to make him more productive. Much room for future work exists in integrating the techniques presented here with a full system and experimenting with its impact on the analyst.

9. REFERENCES

[1] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, 2004.

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

[3] P. Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer, 2012.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[5] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Morgan Kaufmann, 2012.

[6] U. Feige, L. Lovász, and P. Tetali. Approximating min-sum set cover. In *APPROX*, 2002.

[7] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD Conference*, 2014.

[8] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[9] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *VLDB*, 2012.

[10] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. Magellan: Toward building entity matching management systems. *VLDB*, 2016.

[11] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *ICDT*, 2005.

[12] J. Nielsen. *Usability engineering*. Academic Press, 1993.

[13] P. Suganthan et al. Why big data industrial systems need rules and what we can do about it. In *SIGMOD Conference*, 2015.

[14] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *VLDB*, 2011.

[15] S. E. Whang and H. Garcia-Molina. Incremental entity resolution on rules and data. *The International Journal on Very Large Data Bases*, 23(1):77–102, 2014.