Smurf: Self-Service String Matching Using Random Forests

Paul Suganthan G. C., Adel Ardalan, AnHai Doan, Aditya Akella
University of Wisconsin-Madison
{paulgc, adel, anhai, akella}@cs.wisc.edu

ABSTRACT

We argue that more attention should be devoted to developing *self-service string matching (SM) solutions*, which lay users can easily use. We show that Falcon, a self-service entity matching (EM) solution, can be applied to SM and is more accurate than current self-service SM solutions. However, Falcon often asks lay users to label many string pairs (e.g., 770-1050 in our experiments). This is expensive, can significantly compound labeling mistakes, and takes a long time. We developed Smurf, a self-service SM solution that reduces the labeling effort by 43-76%, yet achieves comparable F_1 accuracy. The key to make Smurf possible is a novel solution to efficiently execute a random forest (that Smurf learns via active learning with the lay user) over two sets of strings. This solution uses RDBMS-style plan optimization to reuse computations across the trees in the forest. As such, Smurf significantly advances self-service SM and raises interesting future directions for self-service EM and scalable random forest execution over structured data.

PVLDB Reference Format:
Paul Suganthan G. C., Adel Ardalan, AnHai Doan, Aditya Akella. Smurf: Self-Service String Matching Using Random Forests. *PVLDB*, 12(3): 278-291, 2018.
DOI: https://doi.org/10.14778/3291264.3291272

1. INTRODUCTION

String matching (SM) finds strings from two given sets that refer to the same real-world entity (see Figure 1). This problem is critical for many data science tasks, such as data exploration, cleaning, and entity matching, among others.

As a result, SM has received much attention. Many solutions have been proposed [50, 19], most of which declare two strings a and b matched if $sim[t(a), t(b)] \geq \epsilon$, where $sim(a, b)$ is a string similarity measure (e.g., Jaccard, TF/IDF), t is a tokenizer (e.g., word-based), and ϵ is a threshold. Most existing works focus on developing string similarity measures and efficiently executing matching condition $sim[t(a), t(b)] \geq \epsilon$ over large sets of strings [50, 19, 45].

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291272

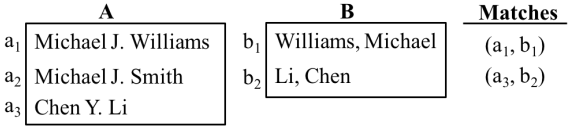

Figure 1: Matching two sets of strings.

While much progress has been made, current SM works are still limited in that *they are not well suited for lay users*, such as domain scientists, journalists, and business users [28]. Most such users are not familiar with the SM literature, and do not know how to select $sim(a, b)$ nor ϵ. As data science applications proliferate, more and more lay users need to perform SM. Thus, *it is increasingly critical that we develop SM solutions that they can easily use.* Such solutions have been called "self-service" and "hands-off", among others [21, 51, 24, 25].

In this paper we develop a self-service solution for SM. Most current SM solutions are not self-service because lay users do not know how to select $sim(a, b)$ and ϵ. To address this problem, recent work [8, 2, 12] asks lay users to label string pairs as match/no-match (in a batch mode or an active learning mode), then uses the labeled data to automatically select $sim(a, b)$ and ϵ. These solutions however achieve only limited accuracy (see Section 7.1), because they consider matching conditions that are a *single predicate* (i.e., $sim([t(a), t(b)] \geq \epsilon)$. In practice, using *multiple predicates* can significantly improve the SM accuracy.

EXAMPLE 1. *Consider matching two sets of person names that contains both long names (e.g., Shivaram Venkataraman) and short names (e.g., Dave Maier). A single predicate such as $jaccard[2gram(a), 2gram(b)] \geq \epsilon$ does not work well because it is difficult to set ϵ properly. A high value for ϵ helps match long names accurately, but can be too high for short names, incorrectly predicting many matching short names as non-matches. Conversely, a low ϵ helps match short names accurately, but can be too low for long names. Intuitively, we should use two predicates of the form $jaccard[2gram(a), 2gram(b)] \geq \epsilon$, but one with a high ϵ for long names, and the other with a lower ϵ for short names. We can check if a name is long using a predicate such as $length(a) > 9$, which returns true if the number of non-space characters in a exceeds 9.*

To address this problem, we consider entity matching (EM) solutions [10, 22, 19]. These solutions have traditionally been used to match *tuples with multiple attributes* [10, 22]. However, we believe that using them to match *strings* can achieve much higher accuracy (than single-predicate SM solutions), because they employ powerful matching conditions,

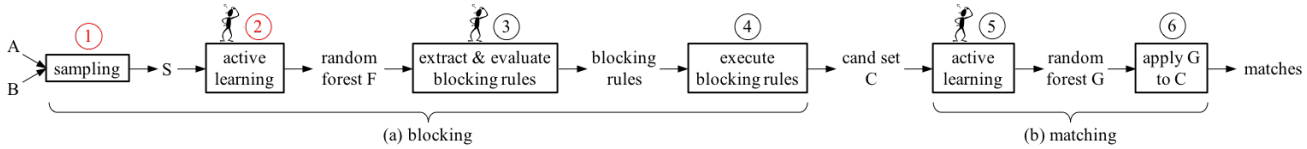

Figure 2: The workflow of Falcon, which requires a lay user to label tuple pairs as match/no-match in Steps ②, ③, and ⑤.

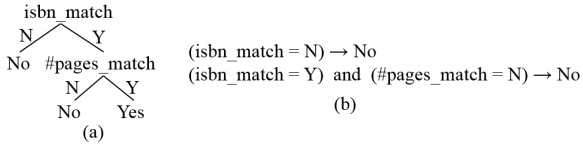

Figure 3: (a) A decision tree learned by Falcon and (b) blocking rules extracted from the tree.

such as a random-forest classifier that uses multiple predicates (e.g., $jaccard[2gram(a), 2gram(b)] \geq \epsilon$, $length(a) > 9$) [24, 14].

Most current EM solutions, however, are not well suited for lay users [29, 18]. For example, they often require a developer to write heuristic rules, called *blocking rules*, to reduce the number of candidate pairs to be matched, then train and apply a *matcher* to the remaining pairs to predict matches. The developer must know how to code (e.g., to write rules in Python) and match entities (e.g., to select learning models and features). Lay users clearly cannot perform these tasks.

The Falcon Solution: In response, we have developed Falcon, a self-service EM solution [14]. To match two tables A and B, *like most current EM solutions,* Falcon *performs blocking and matching, but it makes both stages self-service* (see Figure 2). In the blocking stage (Figure 2.a), it takes a sample S of tuple pairs (Step ①), then performs active learning with the lay user on S (in which the user labels tuple pairs as match/no-match) to learn a random forest F (Step ②), which is a set of n decision trees [7]. The forest F declares a tuple pair p a match if at least αn trees in F declare p a match (where α is pre-specified).

In Step ③, Falcon extracts all tree branches from the root of a tree (in random forest F) to a "No" leaf as candidate blocking rules. For example, the tree in Figure 3.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Figure 3.b shows two blocking rules extracted from this tree. Falcon enlists the lay user to evaluate the extracted blocking rules, and retains only the precise rules. In Step ④, Falcon executes these rules on tables A and B to obtain a set of candidate tuple pairs C. This completes the blocking stage (Figure 2.a). In the matching stage (Figure 2.b), Falcon performs active learning with the lay user on C to obtain another random forest G, then applies G to C to predict matches (Steps ⑤ and ⑥).

Thus Falcon is well suited for lay users, who only have to label tuple pairs as match/no-match. It has been deployed as a cloud service at UW-Madison, and many domain scientists have successfully used it to match strings [25]. Section 7.1 shows that Falcon indeed significantly improves matching accuracy compared to single-predicate SM solutions.

Limitations & Proposed Smurf Solution: Our UW deployment, however, reveals a major problem with Falcon: it can ask a lay user to label many string pairs (e.g., 770-1050 in our experiments). This raises three major problems. First, it can incur a lot of user expense. In certain contexts (e.g., drug matching), one needs to hire experts to label, and these experts are expensive. Second, if users make mistakes, say mislabeling every 10th pair, then labeling many pairs will significantly compound the mistakes. Finally, when crowdsourcing is used to label, labeling can take a long time (e.g., 2 days to label 800 pairs). Thus, reducing the number of pairs to be labeled can significantly reduce the overall SM time.

To address these problems, *we seek to significantly reduce the labeling effort of* Falcon*, while achieving the same SM accuracy.* Our key observation is that in the blocking stage Falcon learns a random forest F (Steps ①-②, highlighted in red), but uses F only to derive blocking rules. Yet F is a full-fledged matcher by itself, i.e., it can classify each string pair as match/no-match. So why not apply F directly to A and B to match the string pairs? This way we can discard the rest of the Falcon workflow (Steps ③-⑥), thus saving the labeling effort in Steps ③ and ⑤.

While appealing, this raises a difficult technical challenge: *how to effectively execute random forest F over two tables of strings A and B without enumerating all pairs in $A \times B$?* Our solution decomposes the process of executing all decision trees in the random forest F into executing a subset of trees in a *pruning step*, then the remaining trees in a *verification step*. It then uses RDBMS-style plan optimization to efficiently execute sets of decision trees in both steps, by *reusing computation across the trees*. We call this solution Smurf (String matching using random forest).

Contributions: As described, this paper significantly advances the state of the art in string matching. Specifically:

- We argue for the increasing importance of self-service SM, and show that current single-predicate self-service SM solutions achieve limited accuracy.
- We develop Smurf, a self-service SM solution that uses multiple-predicate matching conditions (in form of random forests).
- We describe extensive experiments showing that Smurf indeed can significantly outperform single-predicate SM solutions, by 1.15-22.4% F_1. Further, Smurf achieves F_1 accuracy comparable to Falcon, the best current self-service EM solution, yet drastically reduces the number of pairs to be labeled by 43-76%.
- At the heart of Smurf is a novel solution to efficiently execute a random forest over two large sets of strings, using RDBMS-style plan optimization.

Many existing SM works focus on efficiently executing matching conditions (using indexes [50, 19, 45], see Section 3). But they have considered only single-predicate conditions (except [32], which we compare with in Section 7.2). *Our work on efficiently executing a random forest builds on these, but can be viewed as a logical next step*, in that it considers more

powerful matching conditions (in form of random forests using multiple predicates). As such, our work significantly advances this important SM research direction.

Our work also raises two interesting future research directions: *Can* Smurf *be extended to perform EM? And can our solution of executing a random forest over sets of strings be extended to more general settings, such as a set of tables linked via foreign-key joins?* In Section 7.3 we explore these questions. In particular, we provide preliminary evidence showing that Smurf can indeed be extended to EM, and can achieve comparable F_1 accuracy at a far lower labeling cost (than Falcon). But a more rigorous future study is necessary to examine how best to extend it.

Finally, Smurf has been developed as a part of a larger project at UW-Madison [29, 20], which builds Magellan, an ecosystem of interoperable tools for EM and SM, and more details about Smurf can be found in a technical report [42].

2. RELATED WORK

Self-Service Data Solutions: This topic has received increasing attention [21, 51, 24]. Its goal is to develop solutions (e.g., in cleaning, matching, wrangling, etc.) that are very easy for lay users to use. Several self-service SM solutions have been proposed [8, 2, 12]. The most advanced solution [8] learns a single-predicate SM condition using active learning. In contrast, Smurf learns a random forest SM condition, and thus can achieve higher SM accuracy. Self-service EM solutions include [24, 14]. Among these, the Falcon system is the most advanced and can be applied to SM, but often requires a large user labeling effort. Smurf drastically reduces this labeling effort, while achieves comparable accuracy.

String & Entity Matching: String matching (SM), also called *string similarity joins (SSJs)*, has been widely studied [48, 6, 27, 33] (see [50] for a survey). To avoid examining all pairs of strings, prior works use inverted indexes [38, 17], prefix filter [9], size filter [3, 6], position filter and suffix filter [48], etc. Work has examined SM within a database [26, 9, 4, 44] and developed scalable parallel solutions (e.g., using MapReduce [46, 34, 15, 16]). Recent work has also examined top-k SSJs [47, 52]. Current SM work however has only examined single-predicate matching conditions (e.g., [48, 46]). In contrast, Smurf considers more powerful multi-predicate matching conditions in form of random forests. Entity matching (EM) has also received much attention [10, 22, 19, 30, 35, 11], and a wide range of EM solutions have been developed. Most current EM solutions however are not self-service. Recent self-service EM solutions include [24, 14]. Smurf uses Falcon [14] to perform self-service SM, but significantly reduces its user labeling effort.

Scalable Machine Learning over Structured Data: This topic has received growing attention. Most works have focused on *efficiently learning an ML model over structured data*. For example, the works [31, 39] learn generalized linear models over a join without having to materialize the join output. The works [36, 41, 49] learn random forests over large datasets. In contrast, Smurf focuses on the complementary problem of *efficiently applying an ML model over structured data* (specifically on applying a random forest to two sets of strings). This problem has received little attention. A work related to Smurf is [23], which develops pruning techniques for reducing the prediction time of ensemble models, but this work assumes a set of feature vectors as input and thus is not applicable to our SM context.

Rule Execution & Multi-Query Optimization: Executing a decision tree means executing the matching rules of the trees (see Section 4). The works [14, 32] have examined how to efficiently execute *a single matching rule*, e.g., [32] in effect performs a similarity join using a single rule with multiple predicates. In contrast, Smurf examines how to efficiently execute *a set of rules*, by reusing computation. Sec. 7.2 shows that Smurf outperforms [14, 32] for SM, because it can reuse computations across rules.

This *combined execution of rules* is reminiscent of multi-query optimization in RDBMSs, which optimizes the execution of a set of queries [40]. Similar to Smurf, prior works on multi-query optimization also represent each query as a DAG and combine the DAGs into a single DAG by exploiting the common sub-expressions in the queries [40, 53, 37]. However, Smurf's reuse rules exploit the semantics of the string matching operators that we define and hence enable more reuse opportunities; in contrast, existing work would treat the features as blackboxes. Further, Smurf often executes a large number of rules (48-127 in our experiments), necessitating an incremental search strategy to reduce the search time (see Sec. 5.3). In contrast, existing works consider fewer queries (e.g., less than 20 in [53, 37]) and hence use more expensive search strategies.

Additional Related Work: The problem of selecting a subset of trees for the pruning step is reminiscent of selecting an optimal set of SSJ filters (e.g., size filter, prefix filter) when executing a single-predicate SSJ condition [43]. However, the SSJ filters considered in [43] form a conjunction, whereas in our pruning step the trees form a disjunction (i.e., we need to output the string pairs predicted as a match by *at least* one tree). Finally, finding an optimal tree sequence for the verification step is similar to ordering pipelined filters [5]. However, our problem is more complex, a special case of which is the problem in [5] (see the tech report [42]).

3. PROBLEM DEFINITION

Features & Predicates: We define a *feature* to be a function that takes two strings a and b and returns a numeric value. A *predicate* $p(a, b)$ is of the form $f(a, b)$ *op* ϵ, where f is a feature, *op* is a comparison operator (e.g., $\geq, \leq$), and ϵ is a threshold. Predicate $p(a, b)$ evaluates to true iff strings a and b satisfy the comparison, and to false otherwise.

For example, feature $jaccard\_3gram(a, b)$ tokenizes strings a and b into sets of 3-grams S_a and S_b, then returns the Jaccard score $|S_a \cap S_b|/|S_a \cup S_b|$. Predicate $jaccard\_3gram(a, b) > 0.8$ evaluates to true iff the Jaccard score exceeds 0.8. In SM contexts, features often involve string similarity measures, e.g., edit distance, Jaccard, overlap, etc. [50].

String Matching & String Similarity Joins: Given two sets of strings A and B, *string matching (SM)* finds all pairs $(a \in A, b \in B)$ that refer to the same real-world entity [50, 19]. Most current solutions return as matches all pairs $(a, b) \in A \times B$ that satisfy a single predicate, e.g., $jaccard\_3gram(a, b) > 0.8$. [50, 19]. These solutions are said to perform a *string similarity join (SSJ)* between A and B, using this predicate as the join condition [50].

Using Indexes to Execute SSJs: Applying the join predicate to all pairs in $A \times B$ is often impractical because

$A \times B$ can be very large. To address this, prior work typically builds an index I over a table, say A (sometimes multiple indexes are built, over both tables). For each string $b \in B$, it consults I to locate only a (relatively small) set of strings in A that can potentially match with b. It then applies the join predicate only to these string pairs. Numerous indexing techniques have been developed, e.g., inverted index, size filtering, prefix filtering, etc. [50, 19]. Smurf uses these indexes, as we will see in Section 4.

The Falcon Entity Matching Solution: As discussed in Section 1, single-predicate SM solutions achieve only limited accuracy. So we consider EM solutions, which often employ powerful matching conditions with multiple predicates. Specifically, we consider Falcon, a self-service EM solution [14]. For space reasons, we now describe only Steps ① and ② of the Falcon workflow (see Figure 2), because we will use those steps in Smurf.

Step ① - Creating a Sample S: Given two tables A and B to match, Falcon takes a small sample S of tuple pairs from $A \times B$ (without materializing this product). This is because learning directly on $A \times B$ is difficult as $A \times B$ is often too large. Randomly sampling from A and B, however, will not work because S is likely to contain very few matching tuple pairs, rendering learning ineffective.

To address this, Falcon first randomly selects a set of x tuples from table B (assumed to be the larger table). Next, for each tuple b selected from B, Falcon pairs it with (1) $y/2$ tuples from A that are likely to match (see below), and (2) $y/2$ random tuples from A. Thus Falcon tries to get a reasonable number of matches into sample S yet keep it as representative of $A \times B$ as possible. To find tuples from A that are likely to match tuple $b \in B$, Falcon builds an inverted index I on the smaller table A, then uses it to quickly find tuples in A that share many tokens with b. The resulting sample S contains xy tuple pairs, where x and y are tunable parameters (see [14] for details).

Step ② - Active Learning on Sample S: Falcon first examines the schemas of A and B to create a set of features. For example, if it detects that attribute city is of type string, it creates many features that use string similarity measures, e.g., $edit\_dist(A.city, B.city)$, $jaccard\_2gram(A.city, B.city)$, etc., as well as features that capture string properties, such as $length(A.city)$ (see [14]). Next, Falcon uses these features to convert each tuple pair in sample S into a feature vector. This produces a set of feature vectors S'. Next, Falcon trains an initial random forest F (by asking the user to supply two positive and two negative examples), uses F to select a set of "most informative" examples in S', asks the user to label these examples as match/no-match, uses them to retrain F, and so on, until a stopping condition is met.

The Proposed Smurf Solution: To match two sets of strings A and B, we propose that Smurf execute only Steps ①-② of Falcon to learn a random forest F (discarding Steps ③-⑥, thus saving a significant amount of user labeling effort). Smurf then executes a novel last step: apply F to match string pairs (a, b) in $A \times B$ (without materializing $A \times B$). In the rest of the paper we focus on efficiently executing this last step. Specifically:

DEFINITION 2 (SM USING RANDOM FORESTS). *Given two sets of strings A and B, perform active learning with a user U (by executing Steps ①-② of Falcon) to learn a random forest F. Given any two strings $a \in A, b \in B$, F can predict if they match. Now efficiently apply F to A and B to obtain all pairs $(a \in A, b \in B)$ predicted matched.*

As a first step, in this paper we will develop a solution to the above problem *that runs on a single machine*. This solution is easy for lay users to download, install, and run. It is well suited for scenarios where lay users (e.g., domain scientists, journalists) do *not* know how to use, or want to use, or have access to a machine cluster [28, 25]. Solving the above problem on a machine cluster is ongoing work.

4. EXECUTING A RANDOM FOREST

We now provide an overview of our solution to efficiently execute a random forest over two sets of strings.

Suppose we have learned the random forest F of three decision trees (DTs) t_1, t_2, t_3 in Figure 4.a (here, for simplicity, we show each predicate such as $edit\_dist(a, b) < 3$ only as $edit\_dist < 3$). We now consider how to efficiently execute F over two sets of strings A and B. Note that given a string pair $(a \in A, b \in B)$, each tree in F will predict the pair as match or non-match (see Figure 4.a). We refer to the set of all string pairs a tree (or a random forest) predicts to be matches as *the output* of that tree (or that random forest).

Pruning and Verification: Suppose the random forest F outputs a pair (i.e., declaring it a match) only if at least two out of the three trees also output the pair. Naively, we can execute F on two sets of strings A and B by executing each tree t_i on A and B to obtain an output C_i, then output all pairs that appear in the outputs of at least two trees (see Figure 4.b). This however is very time consuming.

A better idea is to execute just two trees, say t_1, t_2 on A and B, to obtain outputs C_1 and C_2 (see Figure 4.c). The set $I = C_1 \cap C_2$ consists of all pairs predicted match by both t_1 and t_2, and so can be output immediately as a part of output of the random forest F.

The set $J = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$ consists of all pairs predicted match by only one tree (either t_1 or t_2). *It is easy to see that we need to apply the remaining tree t_3 only to set J.* Let K be the set of pairs in J predicted match by t_3. Clearly, any such pair is also a match for the random forest F, because it is matched by exactly two trees (either t_1 or t_2, together with t_3). The output of random forest F is thus $I \cup K$ (see Figure 4.c). Any other pair (i.e., neither in I nor in J) is *not* predicted match by both t_1 and t_2 and hence cannot be a match for F.

In practice, the set J tends to be relatively small (see Section 7.2). Thus, applying tree t_3 to J tends to be much faster than applying it to the original sets of strings A and B. This time saving is significant when F is large, say 10 trees. Suppose in this case we need at least five trees to match in order for F to match. Then we can apply six trees to A and B to obtain sets I and J, then apply the remaining four trees to just the relatively small set J.

Smurf uses the above idea. We refer to the first step of applying a subset of trees as *pruning*, and the second step of applying the remaining trees to J as *verification*.

Executing a Tree by Executing Its Matching Rules: The pruning step must execute a set of trees over A and B. Continuing with the example in Figure 4, suppose this step must execute tree t_1 (Figure 4.a).

We refer to each path from the root of t_1 to a "match" node as a *matching rule*. Figure 5.a shows the two rules

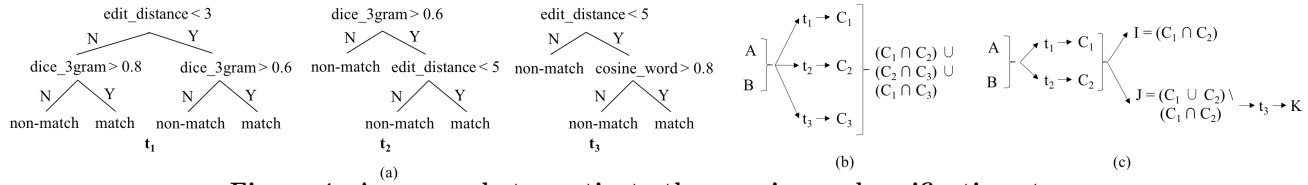

Figure 4: An example to motivate the pruning and verification steps.

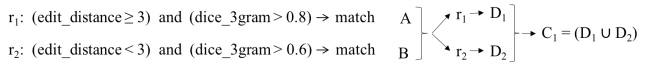

Figure 5: Rules extracted from tree t_1 in Figure 4.a.

extracted from t_1. Each rule is a conjunction of two predicates. Rule r_1 for instance captures the first path leading to a "match" node in the tree. Note that since on this path the predicate ($edit\_dist < 3$) takes value "N", i.e, false, in rule r_1 we capture this as ($edit\_dist \geq 3$).

Executing tree t_1 then reduces to executing these two rules on A and B, then combining their output. Figure 5.b illustrates this step. Note that the output of t_1 is the *union* of the outputs of the rules, because any pair predicted match by a rule is also predicted match by the tree.

Prior Work on Rule Execution: We now consider how to execute a matching rule efficiently. *No prior work has addressed this problem explicitly for string matching.* But two recent works [14, 32] have addressed it for record matching and can be applied to this context.

The first work, Falcon [14], proposes two solutions: ApplyAll and ApplyGreedy. To execute a rule such as r_1 in Figure 5.a, ApplyAll builds indexes for all predicates in the rule, i.e., for both ($edit\_dist \geq 3$) and ($dice\_3gram > 0.8$). It then consults all indexes and take the intersection of the outputs of these indexes to be the set H of record pairs that can possibly be in the rule output. Finally, it evaluates the rule on H. ApplyGreedy builds an index for just one predicate in the rule, consults the index to find a set H of record pairs, then applies the rule to H. The key challenge is to select a predicate that is highly selective to build an index on, to minimize the size of H.

The second work [32] proposes a solution that we will refer to as RAR (Rule Applied to Record Pairs). RAR analyzes the entire rule, builds a single index covering all predicates in the rule (using prefix filtering ideas), uses the index to find a set H, then applies the original rule to H.

Limitations of Prior Work & Our Solution: As described, prior work has developed efficient solutions to execute a *single* matching rule. Executing a tree however requires executing a *set* of rules. As far as we can tell, *no work has examined efficiently executing a set of matching rules.* (Falcon [14] does consider a set of rules, but those are blocking rules, not matching rules as considered here. When adapting Falcon to matching rules, it is easy to show that Falcon is capable of executing only a single such rule.)

Executing a set of rules by executing each rule individually can be very time consuming. We observe that these rules *often share a lot of computation.* So we seek to execute them jointly, in a "multi-query optimization" style. Specifically, we define a set of core operators, use them to generate a plan encoding the rules, then optimize and execute the plan.

EXAMPLE 3. *Consider again the two rules r_1, r_2 in Figure 5.a. Predicates ($dice\_3gram > 0.8$) of r_1 and ($dice\_3gram > 0.6$) of r_2 share computation. The plan in Figure 6.a executes both rules jointly, by reusing this computation. It uses three operators: join, select, and filter (see Section 5.1). Specifically, the plan first performs a single-predicate join between A and B, using join condition $dice\_3gram > 0.6$. Next, it selects from the output of this join all pairs where feature $dice\_3gram$ (which has been computed) exceeds 0.8 (see the left path of the plan), then it computes feature $edit\_dist$ for these pairs and selects only those with $edit\_dist \geq 3$. This produces the set D_1, the output of rule r_1. Similarly, the right path of the plan produces D_2, the output of rule r_2. The plan returns $C_1 = D_1 \cup D_2$, the output of tree t_1. Note how computing the feature $dice\_3gram$ is done only once in this plan (in the "join" node).*

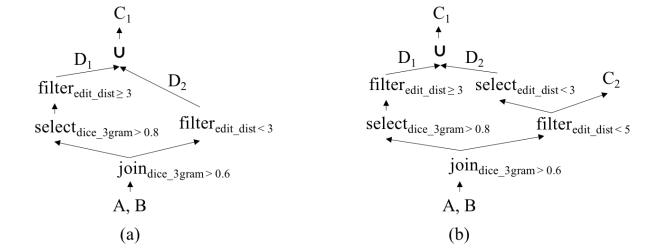

Figure 6: Executing rules in a joint fashion.

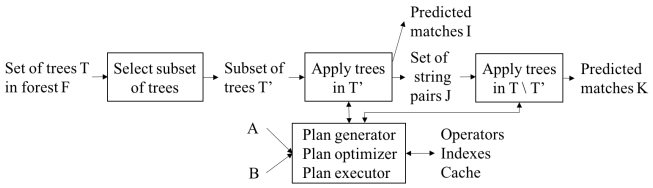

Figure 7: The process of executing a random forest in Smurf.

So far we have discussed executing a single tree. The pruning step executes a set of trees. It is easy to see that the above idea generalizes to this case: *we simply extract all matching rules from the trees, then execute them in a joint fashion.* (We do need to make sure that we know which output pair comes from which tree.) For example, to execute trees t_1, t_2 (Figure 4.a), we extract their three matching rules, then combine the rules to form the plan in Figure 6.b. Note how this plan returns both C_1 and C_2, the outputs of trees t_1 and t_2, respectively. The verification step also executes trees and thus will also use the above joint execution idea.

Solution Architecture: The above ideas lead to the overall architecture for executing a random forest F in Figure 7. Given the set of trees T in F, we first select a subset T' of trees to perform pruning. This produces a set of predicted matches I and a set of candidate pairs J. Next, in the verification step, we apply the remaining trees $T \setminus T'$ to J, to obtain a set of matches K. The output of F is then $I \cup K$. *Both the pruning and verification steps rely on a module that provides efficient execution of a set of trees, us-*

ing operators, indexes, cache, plan generation, optimization, and execution. The key challenges are: (1) how to select a subset of trees for the pruning step? (2) how to execute the remaining trees in the verification step? and (3) how to execute a set of trees, by extracting the rules, defining operators, then generating, optimizing, and executing a plan? We now discuss our solutions to these challenges. First we discuss (3), then build on it to discuss (1) and (2).

5. OPTIMIZING AND EXECUTING A SET OF DECISION TREES

We now describe how to efficiently executing a set of trees. We consider the following concrete problem (and show later how it can be used for pruning and verification):

DEFINITION 4. *[Executing trees over two sets of strings] Let G be a set of decision trees. Given two sets of strings A and B, return all pairs $(a, b) \in A \times B$ that is a match output by at least a tree in G, and associate with each such pair the IDs of all trees in G that output that pair.*

To solve the above problem, we begin by extracting each path from the root of a tree in G to a "match" node as a *matching rule.* Each rule r_i is of the form $p_1^i(a,b) \wedge \ldots \wedge p_{m_i}^i(a,b) \rightarrow$ *predict* (a,b) *as match*, where each $p_j^i(a,b)$ is a predicate (see Section 3).

Let R be the set of all matching rules extracted from the trees in G. Executing G reduces to executing the rules in R, then union their outputs. As discussed earlier, executing the rules in isolation is inefficient. So we seek to execute them jointly, by sharing computation. To do so, we define a set of operators, convert the set of rules into a plan composed of these operators, develop optimization techniques, then search a large plan space to select a good plan. We now discuss these steps.

5.1 Operators and Default Plan Generation

We define then motivate the following four operators:

$join_p(A, B)$: This operator takes two sets of strings A and B and a predicate p, and returns all pairs $(a, b) \in A \times B$ that satisfies p. For example, given predicate $jaccard\_word(a, b) > 0.5$, this operator returns all pairs (a, b) with Jaccard score above 0.5.

$filter_p(C)$: This operator returns all string pairs $c \in C$ that satisfies predicate p. It assumes that feature f in p has *not* been computed for the pairs in C. So given a pair $c \in C$, it computes f for c, then outputs c if c satisfies p. For example, given $jaccard\_word(a, b) > 0.5$, this operator computes feature $jaccard\_word$ for each pair $(a, b) \in C$, then outputs (a, b) if it satisfies the predicate.

$select_p(C)$: This operator is the same as $filter_p(C)$, but it assumes feature f in predicate p *has already been computed* for all pairs in C. So it simply evaluates p for each $c \in C$ and outputs c if p evaluates to true.

$feature_f(C)$: This operator assumes feature f has *not* been computed for pairs in C. So it computes f for all pairs in C then returns those pairs.

Motivations: Operator $join_p(A, B)$ performs a single-predicate SSJ, and has been studied intensively [50]. Operator $filter_p(C)$ is typically applied to string pairs coming out of a $join_q(A, B)$ operator, as we will see below. To motivate operators $select_p(C)$ and $feature_f(C)$, suppose in a plan (defined below) we execute a $join_q(A, B)$ operation to obtain a set of pairs C, then execute both $filter_{jac\_word>0.6}$ and $filter_{jac\_word<0.8}$ on C. Then we would compute feature jac_word twice. To avoid this, we can execute operation $feature_{jac\_word}$ once, followed by two select operations $select_{jac\_word>0.6}$ and $select_{jac\_word<0.8}$.

Default Plan: We now discuss how to convert a set of rules into a default plan (later we show how to rewrite this plan into a set of plans, then select a good one). First, we convert each rule into a plan: given each rule $p_1(a,b) \wedge \ldots \wedge p_m(a,b) \rightarrow$ *match*, we construct a plan $A, B \rightarrow join_{p_1} \rightarrow filter_{p_2} \rightarrow \ldots \rightarrow filter_{p_m} \rightarrow C$. This plan performs $join_{p_1}$ on sets of strings A and B (using indexes), applies $filter_{p_2}$ to the output of the join, applies $filter_{p_3}$ to the output of $filter_{p_2}$, etc., until producing the output C. We then merge the individual plans by adding a node to union their outputs, to obtain a "global" default plan. For example, the set of two rules $r_1 : (edit\_dist < 5) \wedge (jac\_2g > 0.5) \rightarrow$ *match* and $r_2 : (dice\_3g > 0.7) \wedge (edit\_dist < 7) \rightarrow$ *match* is converted into the default plan in Figure 8.a (ignore the dotted boxes and the notations P_1, P_2 for now).

A plan is thus a directed acyclic graph, where the root nodes (those with no incoming edges) denote input data (e.g., A, B), the leaf nodes (those with no outgoing edges) denote output data (e.g., C), the remaining nodes denote the four operators described above plus the set union operator, and the edges denote the flow of data among the operators.

5.2 Strategies for Reusing Computation

Given a plan G many possible strategies exist for reusing computation within G. As a first step, in this paper we propose four such strategies. The key idea is to identify plan fragments that often share computation, then analyze how to merge them to enable reuse. We focus in particular on a common kind of fragment called reusable paths:

DEFINITION 5 (REUSABLE PATH). *Given a plan G, which is a DAG, a reusable path P is a path in graph G of the form $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_n$, such that (a) each node o_i is an operator and has exactly one incoming edge and one outgoing edge in graph G, and (b) P is the longest such path, i.e., we cannot extend path P before o_1 or after o_m to obtain a longer path that still satisfies (a).*

When there is no ambiguity, we will use "path" instead of "reusable path". We refer to nodes o_1 and o_m as the root and leaf nodes of a path, and the node with an edge leading to o_1 as the parent node of the path. Figure 8.a shows two reusable paths P_1 and P_2 (denoted with dotted boxes).

We now describe four reuse strategies for paths: join reuse, inter-path filter reuse, intra-path filter reuse, and filter ordering. For space reasons, we only describe the key idea of each strategy, deferring the details to a technical report [42].

1. Join Reuse: This strategy merges two paths with joins to enable join reuse. Consider the two paths P_1 and P_2 in Figure 8.a. Path P_1 performs $join_{edit\_dist<5}$, while P_2 performs $join_{dice\_3g>0.7}$. These two joins are different and cannot be shared. Observe however that P_2 contains a node $filter_{edit\_dist<7}$. We can push this node down to become the root node of P_2, then merge it with the root node $join_{edit\_dist<5}$ of P_1, to obtain the plan fragment in Figure 8.b, which reduces the number of joins from two to one. To realize this idea, we first define

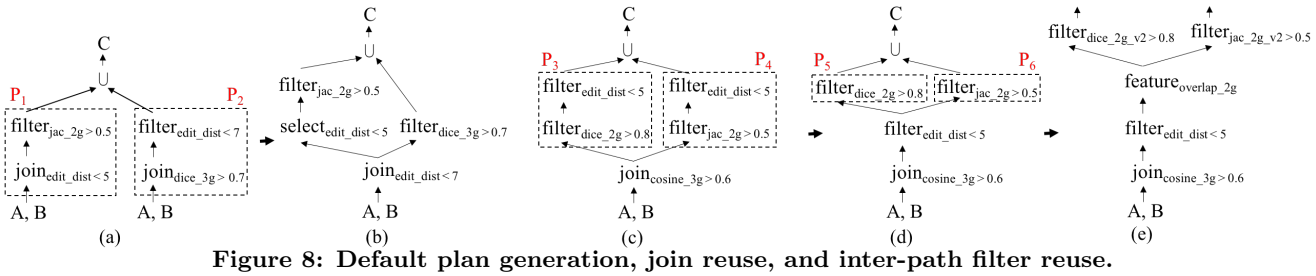

Figure 8: Default plan generation, join reuse, and inter-path filter reuse.

DEFINITION 6 (PREDICATE CONTAINMENT). *Let p_1 and p_2 be two predicates defined over the same feature f, E denote a set of string pairs and $p_i(E)$ denote the result of applying p_i over E. We say that (a) p_1 is contained in p_2, denoted $p_1 \sqsubseteq p_2$, iff for any E, $p_1(E) \subseteq p_2(E)$, and (b) p_1 is equivalent to p_2, denoted $p_1 \equiv p_2$, iff $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_1$.*

For example, for predicates $p_1 : jac\_3g(a, b) > 0.5$ and $p_2 : jac\_3g(a, b) > 0.7$, we have $p_2 \sqsubseteq p_1$. Join reuse then works as follows. It takes as input two paths P_1 and P_2 such that (a) both roots are join operations, and (b) the parents are the same input (e.g., two sets of strings A and B). We first find a node n_i containing predicate $p(n_i)$ in P_1, and a node n_j containing predicate $p(n_j)$ in P_2 such that either $p(n_i) \equiv p(n_j)$, $p(n_i) \sqsubseteq p(n_j)$ or $p(n_j) \sqsubseteq p(n_i)$. If n_i, n_j exist, then we push them down the paths to become the two new roots (the old roots become new filter nodes). Then we merge the two paths. Specifically, if $p(n_i) \equiv p(n_j)$, then we delete n_j and append the rest of path P_2 to n_i. If $p(n_i) \sqsubseteq p(n_j)$, then we modify n_i to be a selection operator and append it as a child of n_j (see Figure 8.b), and so on.

Note that the above describes *one* join reuse rule. If multiple combinations of n_i, n_j exist, then each combination gives rise to a join reuse rule. Later we use these rewrite rules to generate a space of alternative plans.

2. Inter-path Filter Reuse: In this strategy we consider two paths that share the same parent, identify filters (across the paths) that perform common computation, then merge/modify them to reuse the computation. We distinguish two cases:

(a) Reusing filters with the same feature: To motivate, consider paths P_3 and P_4 in Figure 8.c, which share the same parent $join_{cosine\_3g>0.6}$. Both paths execute $filter_{edit\_dist<5}$. So we can push this filter down to be the root of each path, then merge them, to produce the plan fragment in Figure 8.d, which performs the above filter only once.

More generally, this strategy works as follows. Given two paths P_1 and P_2 sharing the same parent, if we find a filter node n_i containing predicate $p(n_i)$ in P_1, and a filter node n_j containing predicate $p(n_j)$ in P_2 such that $p(n_i)$ and $p(n_j)$ are defined over the same feature f, then we rewrite P_1 and P_2 by pushing n_i and n_j down to be the root nodes of the paths. Next, we merge n_i and n_j. If $p(n_i) \equiv p(n_j)$, then we delete n_j and append the rest of P_2 to n_i. If $p(n_i) \sqsubseteq p(n_j)$, then we modify n_i to be a selection operator and append it as a child of n_j. If none of these holds, then we add a new feature node n_f that computes the feature f as a child of the parent node, move n_i and n_j to be n_f's children, then make n_i and n_j into select nodes.

(b) Reusing filters with correlated features: To motivate, consider again the plan in Figure 8.d. Consider the path P_5 consisting of the sole operation $filter_{dice\_2g>0.8}$ and the path P_6 consisting of the sole operation $filter_{jac\_2g>0.5}$. These two filters do not share the same feature, and hence cannot benefit from the reuse strategy in Case (a). However, these features are *correlated*, in that they perform some common computation. Indeed, $dice(X, Y) = 2|X \cap Y|/(|X| + |Y|)$ and $jac(X, Y) = |X \cap Y|/|X \cup Y|$. So they both compute the overlap feature $|X \cap Y|$.

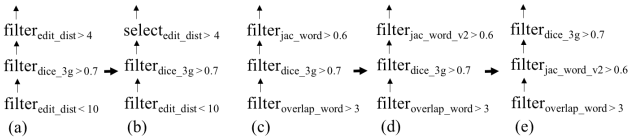

Figure 9: Intra-path filter reuse and ordering.

To reuse this computation, we modify the fragment in Figure 8.d into that in Figure 8.e (we omit the union operator at the top for space reasons), where we first execute $feature_{overlap\_2g}$, then execute the above two filters. However, we rewrite these filters with new features. Consider filter $filter_{dice\_2g>0.8}$. Feature *dice_2g* of this filter performs a full computation of the Dice score, i.e., computing the overlap, among others. But now $feature_{overlap\_2g}$ already computes the overlap. So we define a new feature *dice_2g_v2*, which also computes the Dice score, but assumes that the overlap information already exists (and stored with the incoming string pair). So it does not compute the overlap again, saving time compared to the old feature *dice_2g*. Thus, we rewrite $filter_{dice\_2g>0.8}$ into $filter_{dice\_2g\_2v>0.8}$, and rewrite $filter_{jac\_2g>0.5}$ into $filter_{jac\_2g\_2v>0.5}$.

3. Intra-path Filter Reuse: This strategy is similar to inter-path filter reuse, but applies to filters within a single path. Here we can also distinguish two cases:

(a) Reusing filters with the same feature: Within a single path, we also often have multiple filters with the same feature. (Such paths encode rules extracted from decision trees, and these rules often have multiple predicates with the same feature.) In such cases, we can reuse computation across these filters. For example, the path in Figure 9.a has two filters involving feature *edit_dist*. Clearly we can rewrite the second filter as a select operation, because *edit_dist* has been computed in the first filter (see Figure 9.b).

(b) Reusing filters with correlated features: Within a single path, we also often have filters that have different, but *correlated* features. We can also share computation among these filters, in a way similar to the case of inter-path filter reuse. Consider for example the path in Figure 9.c. Here features *overlap_word* and *jac_word* are different, but correlated: computing the Jaccard score requires computing the overlap. As a result, we can rewrite operation $filter_{jac\_word>0.6}$ as $filter_{jac\_word\_v2>0.6}$ (see Figure 9.d), where feature *jac_word_v2* is a new feature that also computes the Jaccard score, but assumes that the overlap has been computed and stored with the incoming string pair.

4. Filter Ordering: Within a path the filters can be re-ordered (i.e., moved around) without affecting the output of the path. Different orderings however can significantly affect the runtime of the path. Consider again the path in Figure 9.d. Here $filter_{jac\_word\_v2>0.6}$ is quite fast, because it assumes the overlap information has been computed (by the upstream $filter_{overlap\_word>3}$). On the other hand, $filter_{dice\_3g>0.7}$ is slow. If we re-order these two filters, to obtain the path in Figure 9.e, then the slow $filter_{dice\_3g>0.7}$ is applied to fewer string pairs, and thus the entire path may execute much faster. As a result, in this strategy given a path we seek to find a good ordering of its filters. This raises two challenges: how to estimate the runtime of an ordering and how to search the large space of possible orderings. We have adapted the 4-approximation greedy solution in [5] to this problem (see the tech report [42]).

5.3 Searching for a Good Plan

We now describe how to search for a good plan. The reuse strategies in the previous section give rise to a set of rewrite rules, each of which rewrites a plan into a potentially better plan (see [42] for the pseudo code of the rules). So a simple search strategy is to start with the default plan G (see Section 5.1), apply all possible rewrite rules repeatedly, until we cannot apply any more rules, to obtain a place space $\mathcal{G}$. We then estimate the runtime of each plan in $\mathcal{G}$ (see Section 5.4), and select the fastest plan. $\mathcal{G}$ however is often huge, rendering this strategy impractical.

Staged Search: As a result, we explore the following staged search strategy. Given the default plan G, we apply (a) all possible *join reuse* rules repeatedly (until we cannot apply any further), then (b) all possible inter-path filter reuse rewrite rules, then (c) all possible filter ordering rules, and finally (d) all possible intra-path filter reuse rules. The tech report [42] gives the pseudo code of this search process.

The rationale for this ordering of the rules is as follows. First, joins are very expensive. So we want to do (a) first, to consider all possible join reuse opportunities. We can delay (c) and (d) to the end, because they are *local* rules and do not increase the estimated runtime of any target plan (as we discuss below). This leaves inter-path filter reuse rules to be executed in (b). Finally, we do (c) before (d) because it is not difficult to prove that if there is any intra-path filter reuse we want to perform for a path, we can always perform it (or another reuse with equivalent effect) *after* we have performed filter ordering for the path.

Let $\mathcal{U}$ be the resulting plan space. We can reduce $\mathcal{U}$ somewhat, by observing that applying filter ordering or intra-path filter reuse rules does *not* increase the estimated time of the plan. Formally, suppose applying a filter ordering or intra-path filter reuse rule as described in Section 5.2 to a plan P yields a new plan P'. Then the runtime of P' does not exceed that of P, where the runtimes are estimated using the procedure in Section 5.4.

As a result, if we rewrite a plan P into P' using one of the above rules, we can drop P from $\mathcal{U}$. We then estimate the runtime for each plan in $\mathcal{U}$ and select the fastest plan.

Incremental Staged Search: Unfortunately, the plan space $\mathcal{U}$ is still huge (e.g., 100+M plans in our experiments). As a result, we perform an incremental staged search that explores a much smaller space yet still finds good plans (see Section 7). Specifically, let R be a set of n matching rules to be executed. We first sort the rules in R in some order $r_1, \ldots, r_n$ (discussed below). Next, we convert the set of the first two rules r_1 and r_2 into a default plan P_{12}, then perform staged search (as described earlier) on it to find the best plan P^*_{12}. Next, we convert rule r_3 into a default plan P_3, merge it with plan P^*_{12} (by adding a node that unions their output), to form a new plan P_{123}. Then we perform staged search on P_{123}, to find the best plan P^*_{123}. During this search, however, we fix the plan fragment P^*_{12}, applying rewrite rules only to the rest of plan P_{123}. Next, we convert rule r_4 into a default plan P_4, then merge it with P^*_{123}, etc., until we have processed the last rule r_n.

We now discuss how to sort the rules in R. The key idea is to give the maximal amount of freedom in selecting a join operator to the rule whose minimal estimated runtime is higher than that of other rules. As a result, *we sort the rules in the decreasing order of their minimal estimated runtime.* Specifically, for each rule $r_i \in R$, we first enumerate all plans where a predicate in r_i becomes a join operator and the remaining predicates become filter operators (producing at most k plans, assuming r_i has k predicates, see below). Then for each such plan P we perform all filter ordering and intra-path filter reuse rewriting, which can only help reduce P's runtime. Finally, we estimate the runtimes of these plans (see Section 5.4), then select the lowest runtime to be the minimal estimated runtime of rule r_i. For space reasons we defer the complexity analysis to [42].

5.4 Plan Cost Estimation

We now estimate plan runtime. A plan P is a DAG of operators. To execute P, we read the sets of strings A and B from disk into memory, execute the DAG in memory, then write the output to disk. So we will only estimate the CPU time of executing the DAG (the I/O time is the same for all plans), which is the sum of the CPU times of all operations in the DAG. There are five types of operator: select, feature, filter, join, and union. Since unions take negligible time, we only need to consider the first four types of operators. For each operator type, we need to estimate its *runtime* as well as *the size of the output relative to the size of the input* (which we need for estimating the runtime of any operator that consumes the output of this operator).

***select*$_p(C)$:** applies a predicate p to each pair in C to obtain an output C_{out}. We estimate the output size as $|C_{out}| = \rho_p \cdot |C|$, where ρ_p is a selectivity factor for predicate p. We estimate the runtime of this operator as $\alpha \cdot |C|$, where α is the average time to apply p to a pair (this time involves just a single comparison, hence it is very small and assumed to be the same regardless of p). The cost model of this operator thus requires estimating ρ_p and α (see below).

***feature*$_f(C)$:** computes feature f for each pair in C. Thus the output size is the same as the input size. The runtime is estimated as $\beta_f \cdot |C|$, where β_f is the average time to compute feature f for a string pair.

***filter*$_p(C)$:** computes a feature f (specified by predicate p) and applies p to each pair in C, then output only those pairs satisfying p. We estimate the output size to be $\rho_p \cdot |C|$, where ρ_p is a selectivity factor for predicate p, and the runtime to be $(\beta_f + \alpha) \cdot |C|$, because for each pair in C it takes time β_f to compute feature f and time α to apply p.

***join*$_p(A, B)$:** returns all pairs in $A \times B$ that satisfy predicate p. Thus, we estimate the output size as $\rho_p \cdot |A \times B|$,

where ρ_p is the selectivity factor of predicate p. Estimating the runtime of this operator is more involved. Given two sets of strings A and B, this operator first builds an index I on A. Then for each string $b \in B$, it probes I to obtain a relatively small set of strings $Q(b)$ in A. Finally, it processes each pair (b, q), where $q \in Q(b)$, by computing feature f for the pair (the feature mentioned in predicate p), applying predicate p, then outputting the pair if it satisfies p.

Thus, the runtime of this operator consists of the times for index building, index probing, and processing of string pairs. We estimate the index building time to be $\delta_p \cdot |A|$ and the index probing time to be $\mu_p \cdot |B|$. Let $Q = \cup_{b \in B}\ Q(b)$. Then the processing time is $(\beta_f + \alpha) \cdot |Q|$ (because for each pair in Q it takes β_f time to compute feature f, then α time to apply predicate p). Finally, we estimate $Q = \gamma_p \cdot |A \times B|$, where γ_p is a reduction factor showing how much the index-based probing "shrinks" the set of string pairs $A \times B$.

We now describe how we estimate cost model parameters $\alpha, \beta_f, \gamma_p, \rho_p, \delta_p$, and μ_p. We begin by taking a small random sample of string pairs X (of size currently set at 30K) from the sample S used when learning the join condition.

Estimating α: The average selection time per string pair α is a constant which is independent of the predicate being applied. We estimate α by measuring the time to apply an arbitrary predicate (with feature precomputed) over each pair in X, and taking the average.

Estimating β_f: We estimate the time factor β_f for each feature f by measuring the time to apply f to each pair in X, and taking the average.

Estimating γ_p: We estimate the reduction factor due to index-based probing γ_p for each predicate p as follows. We begin by applying the prefix filter for p to each pair in X (i.e., for a given pair (a, b), check if the prefix of a and b share at least one token) to obtain a set of string pairs Y (that satisfy the filter). We then estimate γ_p as $\mid Y \mid / \mid X \mid$.

Estimating ρ_p: We now discuss how to estimate the selectivity of each predicate p, ρ_p. We do not precompute ρ_p for each predicate p, as we do not assume the predicates are independent. For example, if we apply a sequence of two filters containing predicates p_1 and p_2, respectively, over an input set of pairs C, we cannot compute the output size of applying the sequence of filters as $\rho_{p_1} * \rho_{p_2} * \mid C \mid$ since p_1 and p_2 may not be independent.

To address this, for each predicate p_i we compute the coverage of p_i over sample X, $cov(p_i, X)$, which is the set of pairs in X that p_i would satisfy. Then we can estimate the selectivity of p_i, ρ_p, to be $\mid cov(p_i, X) \mid / \mid X \mid$. And we can compute the selectivity of applying a sequence of predicates p_1, p_2, to be $\mid cov(p_1, X) \cap cov(p_2, X) \mid / \mid X \mid$. Hence we only keep track of the coverage of each predicate and estimate the selectivities of predicates on the fly. To estimate selectivities efficiently, Smurf maintains the coverages of predicates in the form of bitmaps.

Estimating δ_p: To estimate the index building time, we need to estimate δ_p which is the average time spent per string in A when building the index. To do so, we take a small random sample of strings Y from A (where $\mid Y \mid$ is $min\{0.1 * \mid A \mid, 1K\}$), then measure the time it takes to index each string $a \in Y$ (i.e., the time taken to insert each token in the prefix of a into the index), and take the average.

Estimating μ_p: To estimate the index probing time, we need to estimate μ_p which is the average time spent per string in B when probing the index. To do so, we take a small random sample of strings Z from B (where $\mid Z \mid$ is $min\{0.1 * \mid B \mid, 1K\}$), then measure the time it takes to probe each string $b \in Z$ (i.e., the time taken to probe each token in the prefix of b) in the index built over strings in sample Y (used for estimating δ_p), and take the average.

6. PRUNING AND VERIFICATION

Recall that to match two sets of strings A and B, we interact with the user to learn a random forest F, then execute F over A and B. In particular, we break the execution of F into two steps: pruning and verification. We now describe how to perform these two steps efficiently.

The Pruning Step: Suppose random forest F has n trees. For ease of exposition, suppose we need at least $\lceil n/2 \rceil$ trees in F to match, in order for F to match. We can easily prove that if the pruning step executes at least $(\lfloor n/2 \rfloor + 1)$ trees, then any string pair not output by this step (i.e., not output by any of these trees) cannot be a match.

To minimize the run time of pruning, we will execute exactly $(\lfloor n/2 \rfloor + 1)$ trees in this step (Section 7.1 shows that executing more trees incurs longer total join execution time.)

Specifically, let T be the set of all trees in random forest F. We will select a subset $T' \subseteq T$ of $(\lfloor n/2 \rfloor + 1)$ trees for pruning, such that the time taken to apply the trees in T' to A and B to produce a set of pairs J, plus the time taken to apply the remaining trees in $T \setminus T'$ to set J is minimized. Let these two times be $time(T')$ and $time(T \setminus T')$, respectively. We estimate $time(T)$ by applying the procedure in Section 5.3 to generate a good execution plan P for the set of trees T, then take the estimated runtime of P (using the cost estimation procedure in Section 5.4) to be $time(T)$. We estimate $time(T \setminus T')$, the verification time, as described later in this section.

The problem is that there are too many possible subsets of trees of size $(\lfloor n/2 \rfloor + 1)$. So we cannot enumerate and estimate $time(T') + time(T \setminus T')$ for all of them, then select the one with the lowest total time. As a result, we select T' using greedy search. First, we assign T' to be the set of all trees T. Then in each iteration we remove the tree t from T' that results in the largest reduction of $time(T') + time(T \setminus T')$, until we have removed $(\lceil n/2 \rceil - 1)$ trees. Let T'_* be the remaining set of trees. We perform pruning using T'_*, i.e., we generate an efficient execution plan for T'_* (see Section 5) then execute it on A and B.

The Verification Step: Suppose that executing T'_* trees in the pruning step produces a set of pairs J. We now consider how to execute the remaining trees on J. We begin by noting that the optimization procedure in Section 5, which finds a good plan to execute a set of trees over *two sets of strings A and B*, can easily be adapted to find a good plan to execute a set of trees over *a set of string pairs J*.

Now let U be the set of the remaining trees to be executed on set J. Similar to how we execute trees in the pruning step, here we can simply use the above optimization procedure to generate a single plan P that executes all the trees in U in a *combined* fashion (i.e., reusing computation). A better solution however is to apply the trees *sequentially* to avoid applying all trees in U to all pairs in J.

EXAMPLE 7. *Consider a forest F of 10 trees, where at least 5 trees must match in order for F to match. Then the*

Table 1: Overall performance of Smurf vs. Falcon on six datasets.

Dataset	A	B	# Matches	Falcon					Smurf					Reduction in # Pairs (in %)
				P	R	F_1	# Pairs	Runtime (sec)	P	R	F_1	# Pairs	Runtime (sec)	
Addresses	24,650	29,531	9,850	99.92	99.90	99.91	1,050	114	98.42	99.60	99.01	440	91	610 (58.1)
Researchers	8,342	43,549	4,556	99.95	95.76	97.81	820	46	97.08	97.87	97.47	200	64	620 (75.6)
Citations	2,616	64,263	5,347	91.90	90.08	90.98	990	126	87.07	91.15	89.06	340	160	650 (65.6)
Names	10,341	15,396	5,132	95.54	95.91	95.72	830	170	90.35	99.79	94.84	360	63	470 (56.6)
Products	2,554	22,074	1,154	93.23	52.51	67.18	1,020	320	66.16	64.04	65.08	400	100	620 (60.7)
Songs	1,000,000	1,000,000	1,292,023	99.99	85.20	92.00	770	10,140	85.23	99.79	91.94	440	11,700	330 (42.8)

pruning step executes 6 trees to produce a set of pairs J. Consider a pair $p_1 \in J$ matched by 4 trees in pruning. Then we can declare p_1 a match as soon as one of the remaining 4 trees matches p_1. Consider a pair $p_2 \in J$ matched by just one tree in pruning. Then we can declare p_2 a non-match as soon as one of the remaining 4 trees predicts it a non-match. Thus we will order and execute the trees in U sequentially. In particular, we want to find the tree sequence that minimizes the total execution time. This problem is NP-hard (see [42]). As a result, we employ a greedy approach. Specifically, let M be the output of applying a tree sequence $\langle t_1, \ldots, t_i \rangle$ to set J, i.e., (a) M contains all pairs in J for which we still cannot make a match/non-match decision, and (b) $J \setminus M$ contains all pairs in J for which we have already made a match/non-match decision (after executing sequence $\langle t_1, \ldots, t_i \rangle$, see Example 7). Then we refer to $|J \setminus M|/|J|$ as the pruning rate of the sequence $\langle t_1, \ldots, t_i \rangle$ and denote this rate as $d(\langle t_1, \ldots, t_i \rangle)$. Let $w(t_i)$ be the average runtime of tree t_i on a string pair.

Intuitively, we want to be able to make match/non-match decisions as soon as possible, for as many pairs as possible. So we want to start with the trees with the highest pruning rates. But we need to balance this against the runtimes of those trees. Thus, we select the tree sequence as follows. First, we select a tree t_i that maximizes $d(\langle t_i \rangle)/w(t_i)$. Then we select another tree t_j that maximizes $d(\langle t_i, t_j \rangle)/w(t_j)$, etc., until we have selected all trees in U. This forms the sequence $\bar{U}$ to be executed in the verification step.

Recall from Section 6 that when considering whether to select a subset of trees T' for pruning, we need to estimate the total runtime of executing the remaining trees, $U = T \setminus T'$, in the verification step. To do this, we first find a good tree sequence $\bar{U}$, as described above. Let $\bar{U} = \langle t_1, \ldots, t_k \rangle$. Then we estimate the runtime of $\bar{U}$ on set J as $|J| \cdot z$, where $z = w(t_1) + (1 - d(\langle t_1 \rangle)) \cdot w(t_2) + (1 - d(\langle t_1, t_2 \rangle)) \cdot w(t_3) + \cdots + (1 - d(\langle t_1, \ldots, t_{k-1} \rangle)) \cdot w(t_k)$.

7. EMPIRICAL EVALUATION

We experiment with the six datasets described in the first four columns of Table 1. Addresses describes street addresses from Yelp and Yellow Pages. Researchers describes the names of researchers at a university. Citations is derived from the dataset used in [13]. Names describes full names from the US Census Bureau [1], and Products and Songs are derived from the datasets used in [14]. The column "# Matches" lists the number of gold matches in each dataset. Smurf was implemented in Cython. All experiments were run on a machine with Ubuntu 14.04.4, two Intel Xeon E5-2630 CPUs (8 cores, 2.4GHz), and 32GB of memory.

7.1 Overall Performance

We begin by showing that (a) Smurf achieves comparable F_1 accuracy, yet drastically reduces the user labeling effort, by 42.8-75.6%, compared to Falcon, and (b) Smurf is significantly more accurate than single-predicate SM solutions, by 1.15-22.4% in absolute F_1.

Smurf vs Falcon: In the experiments below, both Smurf and Falcon use random forests of 10 trees (the default in many learning packages, e.g., scikit-learn), and learn the forests using synthetic users with 0% error rate in labeling. (We experiment with real users below, and Section 7.2 experiments with a varying number of trees and with error rates of up to 20% in user labeling.)

In Table 1, the columns under "Falcon" and "Smurf" show the accuracy in P, R, F_1. They show that Falcon and Smurf achieve comparable F_1 accuracy across the six data sets (67.18-99.91% F_1 vs. 65.08-99.01% F_1). Interestingly, Smurf achieves higher recall than Falcon (e.g., 99.79% vs. 85.2% on Songs, 64.04% vs. 52.51% on Products), which in turn achieves higher precision than Smurf. This is because Smurf considers many more paths in the random forest before deciding to prune a string pair, whereas Falcon prunes pairs in the blocking step by considering only a few paths in the random forest.

Columns "# Pairs" (in red color) show the number of string pairs that the user has to label: Smurf dramatically reduces the total number of labeled pairs by 42.8-75.6% compared to Falcon (see the last column). For example, on Addresses Falcon requires 1050 labeled pairs, whereas Smurf requires only 440, a 58.1% reduction. On Researchers, it is 820 vs. 200, a 75.6% reduction. Finally, columns "Runtime" show that Falcon and Smurf are comparable across the six data sets (46-10,140 secs vs. 63-11,700 secs).

Experiments with Domain Scientists: In a recent project, a team of economists at UW-Madison led by Dr. Brent Hueth had to match two sets of organization names of size 21,531 and 2,617. Using Falcon, they had to label 680 string pairs, achieving 97.7% F_1 (96.4% precision and 99.2% recall), with a machine time of 11 min. In contrast, Smurf only required labeling 300 pairs (a 55.8% reduction in labeling effort), achieving 97.9% F_1 (98.3% precision and 97.7% recall), with a machine time of 1 min 32 sec.

Smurf vs Single-Predicate Solutions: Recall that the most common prior SM solution is to apply a single-predicate join condition, e.g., $jacc\_3gram(a, b) > 0.8$, to A and B. We now examine how this solution compares to Smurf. To do so, for each dataset we find the best single-predicate join condition by an exhaustive search. Specifi-

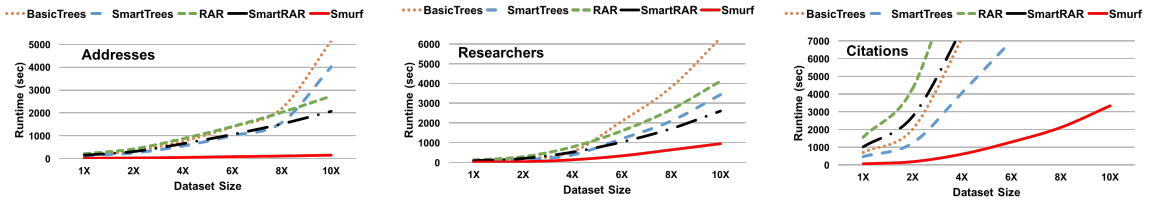

Figure 10: Runtimes of Smurf versus baselines that use existing solutions on rule execution.

Table 2: Selecting a subset of trees for pruning.

Dataset	Smurf	$\text{Smurf}_{\text{rand}}$	$\text{Smurf}_{\text{sel}}$	$\text{Smurf}_{\text{time}}$
Addresses	158	771	158	164
Researchers	956	1123	1146	1050
Citations	3333	6167	6899	5391

cally, we consider 25 features, each created by pairing one of six common tokenization method (e.g., 2-gram, 3-gram, word, numeric, etc.) with one of five common similarity measures (e.g., Jaccard, edit distance, cosine, etc.). For each feature f, we consider all predicates of the form $f \geq t$, where t ranges from 0.1 to 1 in increments of 0.01 (edit distance was converted into a similarity measure for this purpose). We then find the predicate with the highest F_1 accuracy.

The results (not shown for space reasons) show that Smurf consistently outperforms the single-predicate solution, e.g., by 1.15% in absolute F_1 for Citations and by 10.5-22.4% for the other datasets. The tech report [42] shows that this improvement is indeed due to the fact that Smurf can use multiple predicates in the join condition.

7.2 Executing Random Forests

We now evaluate the performance of executing a random forest, which is the main technical contribution of this work.

Comparing to Prior Work: As discussed in Section 4, no published work has optimized the execution of *a set of matching rules*. But two recent works, Falcon and RAR [14, 32], have optimized the execution of *a single rule*. We use these works to build four baselines for comparison. BasicTrees does not do pruning. It executes all trees in the random forest on A and B. To do so, it extracts the matching rules of the trees, executes all of them, then merges their outputs. To execute a rule r, it creates and executes an optimized plan for r as described in Section 5.1. As such, BasicTrees is equivalent to the Falcon variation that uses ApplyGreedy [14]. But it is *not* Falcon; rather, it uses the rule execution part of Falcon to execute all matching rules extracted from the random forest. (We also experimented with the ApplyAll variation [14] but it was outperformed by ApplyGreedy in our settings and is not discussed further.)

RAR is similar to BasicTrees, but when executing an individual rule it uses the holistic prefix index solution of [32]. SmartTrees and SmartRAR are versions of BasicTrees and RAR that use pruning. They first execute a set of $(\lfloor n/2 \rfloor + 1)$ trees (the same set of trees used by Smurf for blocking) to A and B to obtain a set of pairs J, then apply the remaining trees to J. They differ from Smurf only in that they do not execute the rules of the trees in an optimized fashion (i.e., no reuse, see Section 4).

Figure 10 compares the runtimes as we increase the dataset size. Here a value 4x on the x-axis means that we replicate the original dataset 4 times, by using random perturbations (e.g., inserting/deleting characters) of the original strings. For space reasons we only show the results for three datasets.

The results show that Smurf significantly outperforms the four baselines, and this gap increases as the dataset size increases, e.g., at dataset size of 10x, Smurf performs 6-32 times better than BasicTrees (i.e., Falcon), 3-25 times better than SmartTrees, 4-17 times better than RAR, and 2-13 times better than SmartRAR. It is clear that executing the trees in a joint fashion (to reuse computations), as Smurf does, is absolutely critical for scaling.

Performance of the Components: We now "zoom in" to examine the random forest execution in more details.

(a) Pruning: Pruning drastically reduces the number of pairs to be considered, from 56M-727M for $A \times B$ to 4,887-25,763 pairs. Using pruning, SmartTrees significantly outperforms BasicTrees, and similarly SmartRAR outperforms RAR (see Figure 10).

Table 2 examines how well Smurf selects a subset of trees for pruning. It shows the runtime (in secs) of Smurf vs three Smurf variations. Smurf_{rand} uses a random subset of $(\lfloor n/2 \rfloor + 1)$ trees for pruning. Smurf_{sel} selects the first $(\lfloor n/2 \rfloor + 1)$ trees in decreasing order of their pruning power. Smurf_{time} selects the first $(\lfloor n/2 \rfloor + 1)$ trees in increasing order of their average execution time (as we want to reduce the pruning time). The results show that Smurf always outperforms the three variants, often by a large margin (e.g., by 38-51% for Citations), suggesting that Smurf selects good subsets of trees for pruning.

(b) Verification: We found that executing the trees in the verification step in a sequential, instead of combined, fashion reduces runtime by 8-20%, suggesting that sequential execution is effective. To examine how well Smurf orders the trees, we compare it with three variations that order the trees (a) randomly, (b) in decreasing order of their pruning power, and (c) in increasing order of average execution time. For Addresses and Citations, Smurf is the best (9-15% faster compared to the second best). For Researchers Smurf is the second best (11% slower than the best). This suggests that Smurf selects a reasonable sequence of trees.

(c) Optimization: Table 3 examines the effect of executing a set of trees in a joint optimized fashion, on the 10x versions of three datasets. Columns "BT" and "ST" show that BasicTrees and SmartTrees incur significant runtimes, and that optimization (i.e., Smurf) drastically reduces these times to 158-3,333 secs (see Column "O"), a major reduction of 37-88%. The next four columns show the runtimes when we turn off each type of optimization: join reuse (O_1), inter-path filter reuse (O_2), ordering filters (O_3), and intra-path filter reuse (O_4). Comparison with Column "O" shows that all four optimization types are useful, and that the effects

Table 3: Runtimes of the components (in seconds).

Dataset	BT	ST	O	O - O_1	O – O_2	O – O_3	O – O_4
Addresses	5145	4007	158	956	178	169	185
Researchers	6302	3428	956	1164	1015	1211	1020
Citations	43890	23526	3333	6270	3544	4076	3553

of some are quite significant (e.g., O_1 on all three data sets, O_3 on Researchers and Citations).

Sensitivity Analysis: We have performed extensive sensitivity analysis and found that Smurf is robust to changes in the sample size (from 50K to 1M pairs), the error rate of user labeling (0%-20%), the number of trees in the RF (1-25), and number of trees in the pruning step (6-10). For space reasons we defer a detailed discussion to [42].

7.3 Discussion & Additional Experiments

Extending Smurf to Entity Matching: We found that (1) Smurf can be extended to EM, (2) it can achieve accuracy comparable to that of Falcon (while drastically reduces the labeling effort), but (3) more future work is required to examine how best to extend Smurf to EM.

Specifically, we performed Steps ①-② of Falcon to learn a random forest F. Then we executed F directly over the tables A and B. The solution in Sections 4-6 can be minimally modified to do this, by revising the reuse strategies to be "attribute aware", e.g., in join reuse, we can merge two nodes $join_{edit\_dist<5}$ and $join_{edit\_dist<7}$ only if the two "edit_dist" features refer to the same attributes.

Surprisingly, we achieved significantly lower EM accuracies than Falcon (which executes Steps ①-⑥), e.g., 74.04% vs. 81.51% F_1 on Products. We found that this was mainly because the current sampling strategy is suboptimal. For string matching, the sample S on which Smurf does active learning is quite representative of the matches between A and B. However, for EM, S is not as representative (since it is taken in a way that is not "attribute aware", by concatenating all attributes and treating the entire tuple as a string), so the random forest F learned on S is not a very good matcher, resulting in lower EM accuracy. (This problem does not arise in Falcon; there we do not use F as a matcher; we use it only to generate blocking rules, and we can generate good rules even if F is not a good matcher.)

To address this problem, we modified the sampling strategy to be "attribute aware" (see [42] for details), so that sample S has more matches and is more representative of the matches between A and B. The result is highly promising. On the three datasets Products, Songs, and Citations (considered by the Falcon paper [14]), Falcon achieves 81.51%, 98.8%, 94.5% F_1, whereas Smurf achieves 80.04%, 97.16%, 92.43% F_1, only 1.47-2.07% lower than Falcon.

The above results suggest that Smurf can achieve accuracies comparable to Falcon (while reducing labeling effort). But more work is required to examine how best to extend Smurf to EM. First, we need to examine if the sampling strategy can be improved; that can raise Smurf's accuracy even more. Second, in the execution step Smurf creates and loads multiple indexes into memory. In the EM context, however, we found that the number of indexes and their sizes can be quite large (compared to the SM context), and not all of these indexes can fit into memory, raising the challenge of how to manage them effectively. Finally, we need to empirically evaluate how well the modified Smurf performs compared to existing baselines (e.g., SmartTrees, SmartRAR) runtime-wise for EM. We leave these as future work outside the scope of this paper.

Executing a Random Forest over Structured Data: We are examining extending the solution in Section 4-6 to the general setting of executing a RF over structured data. We found that this is indeed possible, but that more work is required to examine how best to do this extension.

Specifically, we consider the general setting of *multiple tables, where some are linked via key-foreign key (FK) relationships*, e.g., A(x,y), B(u,v), C(p,q), where A.y and B.u are linked via the FK constraint. We define an *instance* to be a set of tuples (one from each table) that satisfy the constraints, e.g., (t_A, t_B, t_C), where tuple t_A is from A and linked to t_B from B via the FK constraint, and t_C is an arbitrary tuple from C. We define a *feature* to be a function that takes as input an instance and outputs a value (e.g., computing a score between A.x and B.v). Let F be a binary RF classifier whose predicates involve such features (e.g., learned using [36, 41, 49]).

We consider how to execute F fast over the above tables. (The output of F is the set of instances classified as true.) In adapting the solution in Sec. 4-6, we observe that the idea of pruning and verification (Sec. 4) still applies. Similar to the case of EM, here the reuse strategies (Sec. 5) must be modified to be "attribute aware". *But unlike EM, here the features can be more general (e.g., not string similarity measures), and so the modifications will be more complex.* For example, when can a feature f be used in a join operator (see Sec. 5.1)? We propose that when such features are defined, the developer should supply indexes, if any (similar to the way we supply indexes for string similarity measures). Then feature f can be used in a join operator if (a) an index exists for f, or (b) no index exists but the join combines two or more tables linked via FK constraints (in this case we can use the indexes created by the FK constraints to avoid enumerating all tuple combinations). Reuse strategies need to take these into consideration. The search strategy (to find a good plan) remains the same, but the estimation of cost model parameters and plan execution will have to be modified (e.g., to manage out-of-memory indexes). While we are working on these issues, we consider them outside the scope of this paper.

8. CONCLUSIONS & FUTURE WORK

We argued for self-service string matching, and showed that single-predicate solutions achieve limited accuracy. We showed that Falcon, the best current self-service EM solution, can achieve higher accuracy, but that it often asks lay users to label too many string pairs. We developed Smurf, a self-service SM solution that drastically reduces the number of pairs to be labeled, yet achieves comparable accuracy. At the heart of Smurf is a novel method to efficiently execute a random forest over two large sets of strings. Going forward, we plan to explore more optimization/execution techniques for random forests, the machine cluster setting, and the future directions discussed in Section 7.3.

Acknowledgments: We thank Guoliang Li and Jian He for helping with the experiments. This work is supported by UW-Madison UW2020 grant and NSF grant IIS-1564282.

9. REFERENCES

[1] *Names dataset*. https://catalog.data.gov/dataset/.

[2] A. Arampatzis and A. van Hameran. The score-distributional threshold optimization for adaptive binary classification tasks. In *SIGIR*, 2001.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[4] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. In *SIGMOD*, 2014.

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

[6] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[7] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[8] L. Büch and A. Andrzejak. Approximate string matching by end-users using active learning. In *CIKM*, 2015.

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[10] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[11] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE*, 24(9):1537–1555, 2012.

[12] R. da Silva, R. Stasiu, V. M. Orengo, and C. A. Heuser. Measuring quality of similarity functions in approximate data matching. *Journal of Informetrics*, 1(1):35–46, 2007.

[13] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda. *The Magellan Data Repository* *https://sites.google.com/site/anhaidgroup/projects/data*.

[14] S. Das, P. S. G.C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[15] A. Das Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[16] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, 2014.

[17] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, 2018.

[18] A. Doan, A. Ardalan, J. Ballard, S. Das, Y. Govind, P. Konda, H. Li, S. Mudgal, E. Paulson, G. C. P. Suganthan, and H. Zhang. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, 2017.

[19] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 1st edition, 2012.

[20] A. Doan, P. Konda, P. S. G. C., A. Ardalan, J. R. Ballard, S. Das, Y. Govind, H. Li, P. Martinkus, S. Mudgal, E. Paulson, and H. Zhang. Toward a system building agenda for data integration (and data science). *IEEE Data Eng. Bull.*, 41(2):35–46, 2018.

[21] X. L. Dong. *Challenges and Innovations in Building a Product Knowledge Graph*. Talk, AKBC @ NIPS 2017.

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

[23] W. Fan, F. Chu, H. Wang, and P. S. Yu. Pruning and dynamic scheduling of cost-sensitive ensembles. In *AAAI*, 2002.

[24] C. Gokhale et al. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[25] Y. Govind, E. Paulson, P. Nagarajan, P. S. G. C., A. Doan, Y. Park, G. M. Fung, D. Conathan, M. Carter, and M. Sun. Cloudmatcher: A hands-off cloud/crowd service for entity matching. *PVLDB*, 11(12):2042–2045, 2018.

[26] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[27] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[28] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012.

[29] P. Konda, S. Das, P. Suganthan G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[30] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.

[31] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, 2015.

[32] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, 2015.

[33] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[34] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[35] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool, 2010.

[36] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[37] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[38] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[39] M. Schleich, D. Olteanu, and R. Ciucanu. Learning

linear regression models over factorized joins. In *SIGMOD*, 2016.

[40] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[41] J. C. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *VLDB*, 1996.

[42] P. Suganthan, A. Ardalan, A. Doan, and A. Akella. Efficiently executing random forests over tables to build hands-off string matching services. 2018. UW-Madison Technical Report, available at `www.cs.wisc.edu/~anhai/smurf-tr.pdf`.

[43] C. Sun, J. F. Naughton, and S. Barman. Approximate string membership checking: A multiple filter, optimization-based approach. In *ICDE*, 2012.

[44] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *PVLDB*, 10(12):1925–1928, 2017.

[45] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.

[46] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[47] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, 2009.

[48] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.

[49] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM*, 2009.

[50] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: A survey. *Front. Comput. Sci.*, 10(3):399–417, 2016.

[51] E. Zaidi, R. Sallam, and S. Vashisth. *Market Guide for Data Preparation*. `https://www.gartner.com/doc/3838463/`.

[52] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.

[53] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.