Transformation-based Framework for Record Matching

Arvind Arasu Surajit Chaudhuri Raghav Kaushik
Microsoft Research
{arvinda, surajitc, skaushi}@microsoft.com

***Abstract*— Today's record matching infrastructure does not allow a flexible way to account for synonyms such as "Robert" and "Bob" which refer to the same name, and more general forms of string transformations such as abbreviations. We propose a programmatic framework of record matching that takes such user-defined string transformations as input. To the best of our knowledge, this is the first proposal for such a framework. This transformational framework, while expressive, poses significant computational challenges which we address. We empirically evaluate our techniques over real data.**

I. Introduction

Record matching [1] is an essential step in order to use data warehouses for accurate data analysis. For example, owing to various errors in data, the customer name in a sales record may not match exactly with the name of the same customer in the registration table. A critical component of record matching involves determining whether two strings are similar or not. String similarity is typically captured via a similarity function that, given a pair of strings returns a number between 0 and 1, a higher value indicating a greater degree of similarity with the value 1 corresponding to equality.

As reviewed by Koudas, Sarawagi, and Srivastava [1], previously proposed similarity functions predominantly focus on the syntactic difference between strings in measuring their similarity. While this is indeed an indicator of similarity, there are many cases where strings that are syntactically far apart can still represent the same real-world object. This happens in a variety of settings such as street names (the street name *SW 154th Ave* is also known as *Lacosta Dr E* in the postal area corresponding to zipcode *33027*), the first names of individuals (*Robert* can also be referred to as *Bob*), conversion from strings to numbers (*second* to *2nd*) and abbreviations (*Internal Revenue Service* being represented as *IRS*).

Prior work addresses this issue primarily using *token standardization* [1], [2]. The idea is to pre-process the input data so that all variations of a string are converted into a canonical (standard) representation. Thus, for example, the input could be pre-processed so that all occurrences of *Bob* are converted to *Robert*. However, this approach is inadequate as illustrated by the following example.

Example 1: Suppose we have three strings *Jeff Jones*, *J Jones* and *John Jones* and suppose that record matching is performed using string equality. If we adopt the approach of token standardization, then there are only three possible outcomes: (1) no two of the above strings join (for example, if we do not change any token), (2) all three pairs of strings join say by standardizing *John* and *Jeff* to *J*, and (3) *J Jones* joins with exactly one of *Jeff Jones* and *John Jones*. It is impossible to achieve the following output: *J Jones* joins with both *Jeff Jones* and *John Jones* but the latter two do not join. This is clearly a limitation of token standardization since there are scenarios where this is desirable. □

The key issue illustrated in Example 1 is that equivalences are only one form of string variations. We also have variations such as abbreviations that lose information. For example, a first initial (such as *J*) can be expanded in multiple ways (*John*, *Jeff*, etc.) that are clearly not equivalent.

We argue that it is impractical to expect a generic similarity function to be cognizant of such domain-dependent variations. Rather the matching framework has to be customizable to take these variations as *explicit input*. In this paper, we propose a simple programmatic framework based on *transformation rules* to capture non-syntactic notions of string similarity. Informally, transformation rules generate a set of new strings for any given string; two strings are considered similar if *some* pair of strings generated from the original strings is similar. The following example illustrates this.

Example 2: Consider the names in Example 1. Our transformation rules are of the form $J \rightarrow Jeff$ and $J \rightarrow John$. Informally, the first rule means that an occurrence of *J* can be replaced with *Jeff*. Under these rules, *J Jones* generates three strings: *J Jones*, *Jeff Jones*, and *John Jones*. The strings *Jeff Jones* and *John Jones* do not generate any new string other than themselves. Therefore, the strings *J Jones* and *Jeff Jones* would be considered similar, since they both generate the string *Jeff Jones*; on the other hand, the strings *Jeff Jones* and *John Jones* would not be considered very similar. □

The similarity between strings generated using transformation rules is captured using one of the traditional similarity functions such as jaccard or edit distance. Thus, our framework is orthogonal to the choice of the underlying similarity function.

As Example 2 suggests, and as we will argue in the rest of the paper, transformation rules provide a highly expressive framework for declaratively capturing a wide variety of non-syntactic string similarity such as synonymous words or names, acronyms and first name initialization. To the best of our knowledge, this is the first proposal that takes such transformations as explicit input. We show using an empirical study on real data that we can leverage our transformation rule-based framework to significantly improve the quality of

record matching.

As part of our empirical study, we also consider the ease of specifying transformation rules. We argue using two representative data cleaning domains, addresses and academic publications, that there often exist rich sources of domain knowledge that can be leveraged to semi-automatically derive a large number of useful transformation rules. These include, for example, data published by USPS [3] for addresses, manually curated sites such as Wikipedia [4] and DBLP [5].

While the transformational framework is expressive, it also introduces significant computational challenges. A fundamental operation in record matching is *similarity join*, which identifies all pairs of strings (records) that are similar. The naive approach suggested by our semantics is to generate all derived strings using transformation rules and compute similarity join over the derived strings. However, this approach is not efficient since the number of derived strings could be very large, as illustrated by the following example.

Example 3: Consider the citation:

> *N Koudas, S Sarawagi, D Srivastava. Record linkage: Similarity Measures and Algorithms. Proceedings of the 2006 ACM SIGMOD International Conference. Chicago, IL, USA.*

Suppose that we consider the set of rules $\{N \rightarrow$ *Nick*, $S \rightarrow$ *Sunita*, $D \rightarrow$ *Divesh*, *SIGMOD* $\rightarrow$ *Special Interest Group on Management of Data*, *ACM* $\rightarrow$ *Association for Computing Machinery*, *IL* $\rightarrow$ *Illinois*$\}$. The number of strings generated by this citation under these rules is $2^6 = 64$. □

We address the computational challenges thus posed by our framework. We present general techniques applicable to a large subset of common similarity functions, and some specializations optimized for set-based similarity functions such as jaccard similarity and its variants. Our experimental results suggest that our new techniques provide at least one or two orders of magnitude performance improvements over the naive approach.

II. Related Work

Record matching has been a thriving area of database research surveyed in [1] and [2]. Substantial portions of this work have focused on designing suitable similarity functions. Some of these functions are domain-specific, such as the Jaro distance [6] and Jaro Winkler distance [7] for person names, and functions used by tools such as Trillium [8] for addresses. However, by and large, the focus of prior research has been on domain-independent similarity functions. These can be broadly classified into:

- *core functions* that are based on the syntactic difference between two records, such as edit distance and its variations [9], jaccard similarity, tf-idf based cosine similarity [10], functions based on language models from information retrieval and for example Hidden Markov Models [11]
- *machine-learning* based approaches that use positive and negative examples to identify a combination of core functions [12], [13], [14], [15], and
- *linkage-based functions* [16], [17], [18], [19] that exploit the relationship among attributes to measure the similarity between records (such as joining two papers based on the corresponding sets of authors).

Our main contribution is to the class of core functions which we enrich with a table of user-specified transformations. Our goal is to be domain-independent. The limitations of the core functions in the absence of transformations have already been discussed in Section I. Extensions of these core functions have been studied that attempt to account for equivalences between strings that are syntactically far apart. Affine edit distance [12], [9] is a variant of classic edit distance that allows for prefix-based abbreviations (*Corp* as a short-form of *Corporation*). However, these extensions only account for special cases such as prefix-based abbreviations. For instance, affine edit distance does not account for equivalences of first names and street aliases. These special cases are implicit. In contrast, our approach takes transformation rules as explicit input. We compare the record matching quality of our approach empirically against these approaches [12]. Our technique complements machine-learning based approaches that use examples to learn the best combination of core functions.

Algorithms that perform linkage-based record matching [16], [17] work iteratively jointly deducing equivalences among various attributes (e.g., papers, authors) and using the equivalences derived in the current iteration in the next iteration. In principle, it is possible to seed these algorithms with an initial set of equivalences that are user-defined. However, this is inadequate since not all string variations correspond to equivalences as shown in Example 1. Further, these algorithms model equivalences only at the attribute level, not at the intra-attribute level. For instance, the fact that *Robert* is the same as *Bob* is not used to deduce that *Robert Jones* is the same as *Bob Jones*. Such a deduction is one of our contributions and thus our techniques can be used to complement linkage-based algorithms.

III. String Transformations

As noted in Section I, string matching is a critical component of record matching and is the focus of this paper. We model strings as a sequence of *tokens* where each token is a (smaller) string. We assume the existence of procedures that convert a string into a token sequence and vice versa. A given string may be tokenized for instance by splitting it based on delimiters such as white spaces. By this method, the string *Internal Revenue Service* is converted to the sequence of tokens <*Internal*,*Revenue*,*Service*>. Henceforth, we use the term *string* to refer to a sequence of tokens and the term *substring* to refer to a (contiguous) subsequence of tokens.

A. Transformation Rules

A *transformation rule* consists of a pair $(context, Production)$ where $Production$ is of the form $lhs \rightarrow rhs$. Each of $context$, lhs, rhs is a string. The sequence lhs cannot be empty, but rhs can be. A production without a corresponding context is called a *context-free*

transformation and we use the production itself to denote the rule.

Example 4: Some example transformation rules are:

- *IL → Illinois*
- *ACM → Association for Computing Machinery*
- (*33027, SW 154th Ave → Lacosta Dr E*)

The first two rules are context-free, while the third is context-sensitive. □

We now describe how a string s can be transformed given a set of transformation rules $\mathcal{T}$. The transformations are driven by the productions. The context is used to identify a set of productions that are applicable to s. Let $\mathcal{P}(s)$ denote all productions where $context$ is a substring of s (thus, all context-free transformations are included in $\mathcal{P}(s)$). The productions in $\mathcal{P}(s)$ can be used to transform s. A production $P = lhs \rightarrow rhs$ in $\mathcal{P}(s)$ can be applied to s if lhs is a substring of s; the result of applying the production is the string s' obtained by replacing the substring matching lhs with rhs.

Example 5: We can use the transformation (*33027, SW 154th Ave → Lacosta Dr E*) to go from the string *SW 154th Ave, Miramar FL 33027* to *Lacosta Dr E, Miramar FL 33027*. However, the string *SW 154th Ave, Miramar FL* does not derive any new string since the context is not a substring. □

We can apply any number of productions in $\mathcal{P}(s)$ one after another. However, a token that is generated as a result of production application cannot participate in a subsequent production. We provide the rationale for this restriction in Section III-D. Briefly, without this restriction, the transformation framework becomes intractable.

Example 6: We can use the production *Drive → Dr* to generate the string *Lacosta Dr E* from the string *Lacosta Drive E*. However, we cannot further convert *Lacosta Dr E* to *Lacosta Doctor E* using the production *Dr → Doctor*. □

The set of strings *generated* by s is the set of all strings obtained by applying zero or more productions in $\mathcal{P}(s)$ to s.

Example 7: To illustrate a more complex example, we consider the set of productions: (*B C → X*), (*C D → Y*), (*D E → Z*), (*X Z → U*) as applied to the string *A B C D E*. Note that overlapping portions of this string match different rules. Each such match derives a different string. The set of strings that are generated is {*A B C D E, A X D E, A B Y E, A B C Z, A X Z*}. □

B. String Similarity

We now discuss how we extend the notion of string similarity in the presence of transformation rules. A similarity function f and a threshold $0 < \theta \leq 1$ together define a *similarity predicate* (f, θ) that is satisfied by strings (s_1, s_2) if $f(s_1, s_2) \geq \theta$.

Given a set of transformation rules $\mathcal{T}$, two strings s_1 and s_2 satisfy a similarity predicate (f, θ) *under* $\mathcal{T}$ if there are strings s_1' and s_2' respectively generated by s_1 and s_2 using $\mathcal{T}$ such that $f(s_1', s_2') \geq \theta$.

Example 8: Suppose that the similarity predicate is equality. The string *SW 154th Ave Florida 33027* is equal to the string *Lacosta Dr E, FL 33027* under the transformations {*FL → Florida*, (*33027, SW 154th Ave → Lacosta Dr E*)}. This happens since both of the above strings generate the same string *Lacosta Dr E, Florida 33027*. □

Following from the above discussion, we can assign a similarity score to two strings given a similarity function f and a set of transformations $\mathcal{T}$. The similarity between strings s_1 and s_2 *under* $\mathcal{T}$ is defined to be the maximum similarity (as measured by f) among all pairs s_1' and s_2' respectively generated by s_1 and s_2 using $\mathcal{T}$. With this definition, adding transformations can only increase the similarity between two strings.

Notice that the above definition as stated does not penalize the application of a transformation rule for computing similarity. For instance, for the rules of Example 7, the similarity of both pairs (*A B C D E, A B C D E*) and (*A B C D E, A X Z*) is the same for any similarity function, although the first pair contains exactly identical strings, while the second does not. We can imagine scenarios where penalizing or costing the application of transformation rules is useful, and this can be done in several ways. One example is simply counting the number of rules applied. We can incorporate costs simply by taking as input a budget on the total cost of rules to restrict the number of strings generated by a given string. This methodology is independent of the underlying similarity function. Of course, this is not the only way to incorporate costs—we could for example aggregate the rule application costs with the final similarity value returned. We defer a detailed study of these alternatives to future work.

C. Generating Transformations

One question that arises from our framework is how we can obtain transformations. We now discuss various ways of doing this.

First, we may obtain transformations through specialized tables that are readily available for various domains. We illustrate with the following examples.

1) *US Addresses:* The United States Postal Service (USPS) [3] publishes extensive information about the format of US addresses, which can be used to get transformation rules for address strings. The published information contains a comprehensive list of alternate representations of common strings such as *street* and *st*, and *NE* and *North East*. It also contains a list of about 176,000 street aliases. Our example in Section I (the street name *SW 154th Ave* is also known as *Lacosta Dr E* in the postal area corresponding to zipcode *33027*) was drawn from this table. This set of aliases holds for a given zipcode which constitutes the context.
2) *Academic Publications:* The DBLP site [5] provides a list of conferences and their abbreviations, which can be used to derive transformation rules via screen-scraping. We can also use the list of authors in DBLP

to get transformation rules corresponding to first name abbreviations such as those illustrated in Example 2.

3) There are also several useful online resources such as `www.acronymfinder.com`, which contains extensive lists of abbreviations, and Wikipedia [4].

Second, transformations can be programmatically generated. For example, we can generate rules that connect the integer and textual representation of numbers such as *25th* and *Twenty-Fifth*.

Third, the set of transformation rules need not be explicitly specified, but could be specified implicitly using regular expressions. For example, the regular expression *Rural Route \d+ → RR \d+* conceptually specifies an infinite set of transformation rules of the form *Rural Route* $i \rightarrow$ *RR* i, for $i = 1, 2, \ldots$. We can apply such transformation rules to derive strings using standard techniques from regular expression matching.

We use transformation rules obtained as described above in our experiments in Section V.

D. Relationship to Formal Grammars

We now relate our method of applying productions to formal grammars [20]. The goal of this is two-fold. It illustrates the principle behind our method, and also shows why we do not permit a token that is generated as a result of production application from participating in a subsequent production.

Suppose we assign one terminal for each distinct token t and a non-terminal N_t corresponding to t. Given string s, let $N(s)$ denote the sequence of non-terminals corresponding to the sequence of tokens in s. Consider the formal grammar defined using the following rules:

1) $Start \rightarrow N(s)$
2) $N(lhs) \rightarrow rhs$ for each production $lhs \rightarrow rhs$ in $\mathcal{P}(s)$.
3) $N_t \rightarrow t$ for each token t.

It is easy to see that the set of strings that can be obtained from s as stated above is equivalent to the set of strings generated by the above formal grammar.

Suppose that we allowed tokens generated as a result of production application to participate in subsequent productions any number of times by changing the above grammar to replace rules of the form $N(lhs) \rightarrow rhs$ with $N(lhs) \rightarrow N(rhs)$, then we are left with unrestricted grammars [20] where even deciding membership is undecidable (in our setting, that would mean that given two strings and a set of productions, even the problem of determining whether one string is generated by the other is undecidable.) We thus need to restrict the power of this grammar by requiring that tokens generated as a result of production application participate in subsequent productions only a bounded number of times. While we set this bound to be 0 in this paper, our techniques extend to any bound $k \geq 0$.

IV. Similarity Join with String Transformations

In Section III-B, we define the notion of similarity under transformations (this is based on a given underlying similarity function). Recall that the goal of record matching is to find pairs of records that are similar. In practice, the record matching is implemented in two different settings. The *indexing* setting is where one of the input relations is given in advance. The idea is to pre-process this relation to create an index which is then used at query time to take an input string and look up all records that are similar to it. The *join* setting is where both the input relations are given and the goal is to find pairs of records that are similar. In this paper, we focus on the join setting noting that our techniques are also applicable to indexing.

The formal similarity join problem can be stated as follows: given two input relations R and S, a relation $\mathcal{T}$ containing transformations and a similarity threshold θ, find all pairs of strings $(r, s), r \in R, s \in S$ whose similarity under $\mathcal{T}$ is greater than or equal to θ. There are two components to this problem. First, for each string in R and S, we need to find the set of *matching* transformations, i.e. transformations where both the $context$ and lhs are substrings. Second, we use the set of matched productions to perform the similarity join. We now describe the details of each of these components. We focus on context-free transformations. Extending the techniques presented here to handle context is straightforward.

INPUT: Relations R, S, $\mathcal{T}$, similarity function f and threshold θ

1. Let relations $ExpandR$ and $ExpandS$ be initialized to R and S respectively
2. For each $r \in R$, find every string r' generated by r under $\mathcal{T}$ and add (r, r') to $ExpandR$
3. For each $s \in S$, find every string s' generated by s under $\mathcal{T}$ and add (s, s') to $ExpandS$
4. Perform a similarity join on $ExpandR$ and $ExpandS$ to find tuples (r, r', s, s') such that $f(r', s') \geq \theta$
5. Output all distinct pairs (r, s) from the tuples (r, r', s, s') returned by step 4.

Fig. 1. Baseline Similarity Join

A. Finding Matching Transformations

For a given input string x in either R or S, the goal is to find all transformations where lhs is its substring. We adopt well-known techniques to address this problem. Observe that if l is a substring of x, then l is a prefix of some suffix of x. Under the assumption that the relation $\mathcal{T}$ fits in-memory, we construct a trie over all the distinct lhs in $\mathcal{T}$. We then process every string in R and S and for a given string, use each of its suffixes to look up the trie. For the case when $\mathcal{T}$ does not fit in memory, there exist standard external memory adaptations of the above approach which we can leverage. The details are straightforward and we defer them to the full version of the paper.

B. Baseline Similarity Join

The semantics of computing a similarity join under transformations suggests the algorithm outlined in Figure 1. In steps 2 and 3, we store the original string along with the string generated by it since we use the original strings in step 5. In

our implementation, we store an identifier with every string that we use instead of the string itself.

This is the algorithm we use when we treat the similarity function as a blackbox. There are several similarity functions for which the best known implementation of a similarity join is via a cross-product of the input relations. In such cases, step 4 above involves computing a cross-product. While this is an expensive step, this is not an artifact of the transformation framework, rather a problem intrinsic to the similarity function chosen.

C. Similarity Join using Signatures

Efficient implementations over a relational database system have been proposed for a large class of similarity functions such as jaccard similarity, hamming distance, cosine similarity and edit distance [21], [22], [23] based on *signature-schemes*.

We begin by reviewing the notion of signature schemes. A signature-based algorithm for computing the similarity join between R and S involving similarity predicate (f, θ) operates as follows: It first generates a set of *signatures* for each string in R and S. The signatures have the *correctness* property: if $f(r, s) \geq \theta$, then r and s share a common signature. Signature schemes have been proposed for several similarity functions such as edit distance and jaccard similarity [1]. Based on this property, the signature-based algorithm generates *candidate pairs* by identifying all $(r, s) \in R \times S$ such that the set of signatures of r and s overlap. Since set overlap can be tested using an equi-join, a DBMS is used for evaluating this step. Finally, in a *post-filtering* step, it checks the similarity join condition $f(r, s) \geq \theta$ for each candidate pair (r, s), and outputs those that satisfy the condition.

We illustrate signature schemes through an example. Suppose the similarity function is jaccard similarity obtained by tokenizing the two strings into sets of tokens (for example, the string "Microsoft Corporation" can be tokenized to obtain the set {"Microsoft", "Corporation"}) and computing the ratio of the (weighted) intersection size over the (weighted) union size. One previously proposed signature scheme for computing the similarity join between two relations with the predicate that the jaccard similarity be above threshold θ is *prefix-filtering* [24]. We fix a global ordering over the universe of elements $\{1, \ldots, m\}$ that constitute the sets. The prefix filter of a set s at jaccard threshold θ is defined to be the subset of s containing the $(1 - \theta) \mid s \mid$ smallest elements of s according to the global ordering. For example, if the global ordering of $\{1, \ldots, m\}$ is simply the natural number ordering, the prefix filter of $\{1, 3, 6, 8, 9\}$ for jaccard similarity $\theta = 0.6$ consists of the two smallest elements of s, i.e., $\{1, 3\}$. We can show that prefix filter satisfies the correctness property required of signatures: the θ-prefix filters of two sets with jaccard similarity $> \theta$ have nonempty intersection. In practice, the ordering of elements induced by their frequency in the input sets (rarer elements are smaller) results in the best performance [23].

We now focus on similarity functions that can be implemented using signature schemes. The straightforward way of using signature schemes is to use them in step 4 in Figure 1. This involves generating signatures for each string in $ExpandR$ and $ExpandS$. Since the number of strings generated by each string can be large as illustrated in Example 3, this approach can be prohibitively expensive.

INPUT: Relations R, S, $\mathcal{T}$ and threshold θ
1. For each $r \in R$, compute $Sign(r) = \bigcup_{i=1}^{m} sign(r_i)$ where r generates the set $\{r_1, \ldots, r_m\}$
2. For each $s \in S$, compute $Sign(s) = \bigcup_{j=1}^{n} sign(s_j)$ where s generates the set $\{s_1, \ldots, s_n\}$
3. Generate all candidate pairs $(r, s) \in R \times S$, satisfying $Sign(r) \cap Sign(s) \neq \phi$
4. Output any candidate pair (r, s) whose similarity under $\mathcal{T}$ is $\geq \theta$.

Fig. 2. Signature-Based Algorithm using Compression

Suppose that $r \in R$ generates the set of strings $\{r_1, \ldots, r_m\}$ under $\mathcal{T}$ and that $s \in S$ generates the set $\{s_1, \ldots, s_n\}$. Note that the above approach generates signatures for each r_i and s_j. Denote the signature generated by string r_i as $sign(r_i)$. Observe that:

Property 1: There exist i, j such that $sign(r_i) \cap sign(s_j) \neq \phi$ if and only if $(\bigcup_{i=1}^{m} sign(r_i)) \cap (\bigcup_{j=1}^{n} sign(s_j)) \neq \phi$.

Thus, instead of generating each $sign(r_i)$, it suffices to generate $Sign(r) = \bigcup_{i=1}^{m} Sign(r_i)$ (respectively $Sign(s) = \bigcup_{j=1}^{n} Sign(s_j)$). This set can be significantly smaller since if some element appears in more than one distinct signature, it needs to be represented only once. We refer to this technique as *signature compression*. We illustrate this through an example.

Example 9: Consider the citation record in Example 3. Recall that the number of generated records is $2^6 = 64$. Suppose the set (of last names) {*Koudas, Sarawagi, Srivastava*} is a signature generated by all the 64 generated records. This is not unexpected for several common similarity functions since an overlap on the last names of the authors in a citation is indicative of a high value of similarity. This signature need not be replicated 64 times; one copy suffices.

The overall signature-generation step is illustrated in Figure 2.

D. Optimizations for Jaccard variants

The signature compression technique illustrated in Section IV-C applies to any similarity function that can be supported via signature schemes. Clearly, a better knowledge of the similarity function is likely to open up even more opportunities for improving the execution efficiency of similarity joins. We now present optimizations that focus on jaccard similarity (denoted $JaccSim$) — computed by tokenizing two strings into (multi)sets of tokens and taking the size of the intersection divided by the size of the union. We choose jaccard similarity since it has been shown that supporting jaccard similarity efficiently leads to efficient implementations of several other similarity functions [22], [24]. Further, the optimizations we describe below can be applied to any set-based similarity function that has the property that for sets of a given size, the similarity score is monotonic with the

INPUT: Relations R, S, $\mathcal{T}$, threshold θ and a clustering of the tokens

1. For each $r \in R$:
   Rename each matching transformation $lhs \rightarrow rhs$ to obtain $cluster(lhs) \rightarrow cluster(rhs)$
   Compute $ClusteredSign(r) = \bigcup_{i=1}^{m} sign(r_i)$ where $cluster(r)$ generates the set $\{r_1, \ldots, r_m\}$ using the renamed transformations
2. For each $s \in S$, compute $ClusteredSign(s)$ similarly
3. Generate all candidate pairs $(r, s) \in R \times S$, satisfying $ClusteredSign(r) \cap ClusteredSign(s) \neq \phi$
4. Output any candidate pair (r, s) whose similarity under $\mathcal{T}$ is $\geq \theta$.

Fig. 3. Signature-Based Algorithm with Token Clustering

intersection size (examples include hamming distance and trivially, set intersection.)

A well-known signature scheme for jaccard similarity is based on the idea of *locality-sensitive hashing (LSH)*. The compressed LSH signatures corresponding to the set of all strings generated from a given input string, can be constructed without explicitly materializing the set. The details while straightforward are intricate and we defer it to the full version of the paper. Before discussing the postprocessing step (Step 4, Figure 2), we first present a technique for further reducing the number of signatures.

1) Token Clustering: The signature compression technique described in Section IV-C exploits the resemblance among the strings generated by a given input string — the more the resemblance, the greater the opportunities for compression. We further enhance this resemblance by clustering all the tokens. Suppose we assign a new token corresponding to each cluster of tokens. For a token t, let $cluster(t)$ denote the token corresponding to the cluster to which t is assigned. For a string x, let $cluster(x)$ denote the string obtained by replacing every token t with $cluster(t)$. We can see that the following property holds.

Property 2: Consider two strings $r \in R, s \in S$ along with the transformations that match them. The jaccard similarity of $JaccSim(cluster(r), cluster(s)) \geq JaccSim(r, s)$.

Thus, the algorithm in Figure 2 could be modified as follows: before finding the set of strings generated by a given string, rename both the string and the matching transformations according to the token clustering, generate signatures and then do the post-filtering step with the *original* (un-renamed) strings. By Property 2, the renaming cannot miss out any pair of strings that satisfy the similarity predicate. Thus, this modification is correct. We outline this in Figure 3.

The advantage yielded by this is that if there are rules where a single token yields a large number of alternatives — such as is the case with abbreviations where for instance the first initial J could lead to a large number of first names beginning with J — these rules would shrink to a single rule if we clustered all of these first names and the letter J. Note that while this is similar to token standardization discussed earlier, it is a performance technique in our context and does not change the semantics. In particular, the procedure outlined in Figure 3 returns the same (correct) result *independent* of the clustering used.

This leaves open the question of what clustering yields the best overall benefit. On the one hand, if we leave every token in its own cluster, then the resemblance among the generated strings is unaffected and we get no benefit whatsoever. On the other hand, if we collapse all tokens into a single cluster, then the resemblance among generated strings is maximized, but the number of false positives returned when we join the signatures is likely to be excessively high. Based on this observation, we define our goal so that tokens that appear on the rhs side of the same lhs are more likely to be in the same cluster, whereas tokens that do not appear together on the rhs side of the same lhs are less likely to be in the same cluster. We capture this intuition by using the well-known paradigm of correlation clustering [25].

2) Post-Processing: The post-processing step of our signature based algorithm involves checking for a given pair of strings r, s whether the similarity predicate under transformations holds (this step is performed with the original, un-renamed strings and transformations). The straightforward method of checking this condition is to actually compute all strings generated by r and s, compute their cross-product and check exhaustively whether there is some pair satisfying the similarity predicate. The question arises whether we can do better than this.

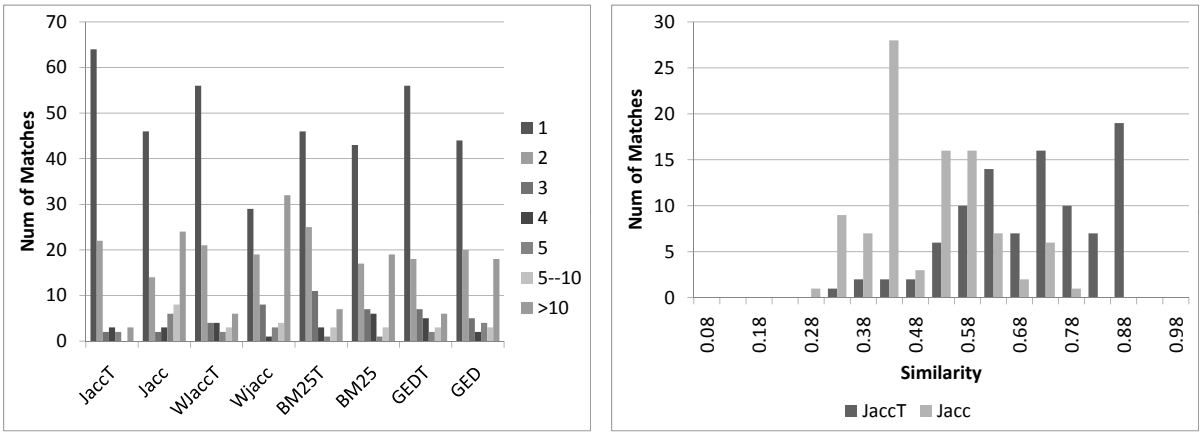
Unfortunately, we show that in the worst case, it is NP-hard to avoid such an exhaustive check. This is formally stated below.

Lemma 1: The problem of computing the jaccard similarity between two input strings r, s given a set of transformations is NP-hard.

However, we observe in practice that a large class of transformations are such that both the lhs and rhs are single tokens (e.g., *St* $\rightarrow$ *Street*). We call them *single-token* transformations. Interestingly, if all transformations are single-token, then it is possible to compute the jaccard similarity between two strings (under the transformations) in polynomial time. We achieve this by reducing this problem to bipartite matching [26].

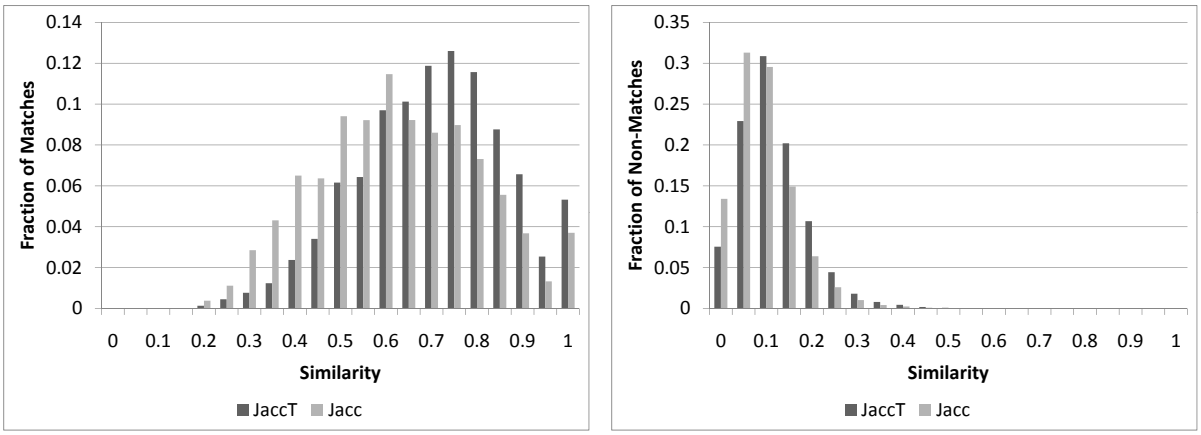
Observe that under single-token transformations, the length of all generated strings is the same and equals the length of the original string. Thus, maximizing the jaccard similarity reduces to maximizing the size of the token-(multi)set intersection. Consider a bipartite graph where we have the set of tokens from r and s on either side. An edge is drawn between two tokens t_r and t_s (on either side) of this graph if (1) they are equal, or (2) there is a rule $t_r \rightarrow t_s$, or (3) there is a rule $t_s \rightarrow t_r$, or (4) there is some t such that there are rules $t_r \rightarrow t$ and $t_s \rightarrow t$. The size of the maximum matching in this graph can be shown to be the maximum intersection size we are seeking.

In the presence of single- and multi-token transformations, we restrict the exhaustive checking to the multi-token transformations.



(a) Rank distribution of matching records (b) Similarity distribution of matching records

Fig. 4. Address dataset: Quality results



(a) Similarity distribution of matching records (b) Similarity distribution of non-matching records

Fig. 5. Cora dataset: Quality results

V. Experiments

The goal of our empirical study is to investigate (1) the impact of transformation rules on record matching quality and (2) the execution efficiency of our algorithms.

A. Datasets and Transformation Rules

We use two datasets for our study. One is a representative dataset based on US addresses that we refer to as the address dataset. It consists of two tables: (1) A reference table USPS of clean US addresses obtained from United States Postal Service [3]. (2) A table `OrgAddress` containing proprietary organization addresses. Each table contains two columns, a column containing address strings, and an integer identity column. An address string encodes information such as the street number, street name, city, state, and zip code. The USPS table has 5 million records while the `OrgAddress` table has 2 million.

We use two types of transformation rules for the address dataset. The rules of the first type are obtained using USPS published domain knowledge as described in Section III-C. There are about 176,000 such transformation rules. We refer to this set of rules as U. The second type of transformation rules are edit-distance based: They are programmatically generated and are of the form *token1* $\rightarrow$ *token2*, where *token1* and *token2* are tokens from the input tables that are within a given edit distance threshold k (e.g., *califarnia* $\rightarrow$ *california* is generated when the threshold is 1). We can vary the edit distance threshold to control the number of input rules and thus study how the performance of our algorithms varies as a function of the number of rules. We generate edit rules corresponding to edit similarity thresholds of 0.8 and 0.85, which we denote E(0.8) and E(0.85), respectively. (Edit similarity of two strings, s_1 and s_2, is defined as $1.0 - ed(s_1, s_2)/\min(|s_1|, |s_2|)$, where $ed(s_1, s_2)$ denotes the edit distance between the strings, and $|s_1|$ and $|s_2|$ denote the length of s_1 and s_2, respectively.)

The second dataset is the `Cora` citation dataset from the RIDDLE repository [27]. This consists of a single relation with about 1300 unsegmented citation strings which we use for record matching against itself (self-join). We use this data set only for quality experiments. For this dataset, we use transformation rules derived from DBLP as described in Section III-C.

B. Quality

We now present our quality experiments to demonstrate the value of transformation rules. We consider four previously proposed similarity functions—jaccard similarity (*Jacc*), a weighted variant of jaccard similarity where tokens are assigned idf weights (*WJacc*), Generalized Edit Distance [28] (*GED*) that has been shown to be more effective than the classic edit distance, and BM25 measure that is based on tf-idf cosine similarity. These four similarity functions are fairly representative of similarity functions used in data cleaning [29]. For both of the datasets chosen above, we study the impact of incorporating transformation rules on the quality of record matching for each of these similarity functions. We denote the similarity under transformations by adding the suffix *T*. Thus *JaccT* denotes Jaccard similarity under transformations.

Address data: For both datasets, we manually construct the "golden truth". For the address dataset, the golden truth specifies, for each record in the `OrgAddress` table, the correct matching "clean" record in the `USPS` table. We obtain a subset of 100 "dirty" records by choosing the 100 records with the least similarity (as measured by GED similarity) to the matching `USPS` record. Therefore, these 100 records do not contain easy matches, such as those that can be found by string equality.

We measure record matching quality in two ways: First, for each record in `OrgAddress`, we measure the rank of the correct matching `USPS` record in terms of similarity. Ideally, the rank of the correct USPS record should be 1, i.e., the correct record should have a higher similarity than any other record in the USPS table. Second, we measure the similarity score of the correct USPS record. Again, it is desirable that this score be high; a higher similarity translates to greater efficiency, since similarity joins and lookups have better performance for higher similarity thresholds than for lower thresholds.

Figure 4(a) shows the distribution of ranks of the correct matching USPS records. For all similarity functions that we consider, the ranks are generally lower (better) with transformation rules than without them. In particular, for all similarity functions, the number of matching records with rank greater than 10 is around 20 without transformation rules; with transformation rules, this number drops to around 5 for all functions. We also observe that in the presence of transformations, the gap between a simple similarity function such as jaccard and a more sophisticated one such as BM25 drops—in fact, JaccT has more correct matches with rank within 10 than BM25T.

Figure 4(b) shows the distribution of similarity of the matching USPS records for jaccard similarity function, with and without transformation rules. We can see that the similarity of the matching records sharply increases when we include transformation rules (the trend is the same for the other similarity functions as well). Not only does this indicate a higher confidence in the matches produced, as mentioned earlier, it also has performance implications since the efficiency of similarity join computation is greater for higher values of the similarity threshold.

Transformation set	*Expansion*	*No. of Rules*
U + E(0.8)	228.7	220,000
U + E(0.85)	15.7	186,000
U	6.89	176,000

Fig. 6. Expansion for different transformation sets

Cora data: The golden truth for the Cora dataset specifies for each pair of records if it represents a valid match or not. Figure 5(a) shows the similarity distribution for matching pairs for jaccard similarity function with and without transformation rules. We observe a behavior similar to the `Address` data, showing the impact of incorporating transformations on record matching quality. The average similarity of matching pairs increases from 0.65 to 0.72 when we add transformations. The results are similar for other similarity functions as well.

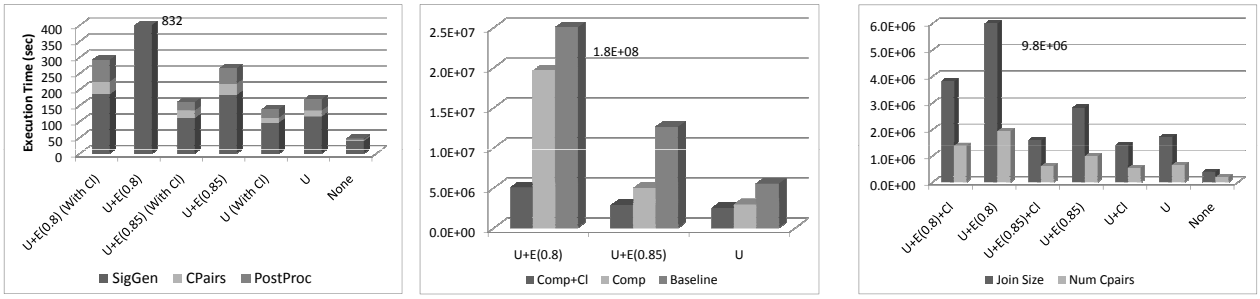
We further observe that incorporating transformations leads to a sharper separation between the similarities of matching and non-matching pairs. Figure 5(b) shows the similarity distribution for non-matching pairs. The average similarity of all matched pairs is 0.72 whereas the average similarity for non-matched pairs is 0.14 in the presence of transformations, whereas without transformations, these numbers are 0.65 and 0.12 respectively. Note that the match-similarity increases by about 0.1 whereas the increase in non-match similarity is negligible.

C. Performance

The goal of this section is to (1) show that our algorithms described in Section IV significantly outperform the naive evaluation of similarity joins, and (2) study the relative contributions of the various techniques proposed earlier for improving the execution efficiency. We compare the performance of the following algorithms: (1) `Baseline` which exploits neither signature compression nor token clustering, (2) `Comp` which leverages signature compression but not token clustering, and (3) `Comp + Cl` which leverages both signature compression and token clustering. All our implementations use the bipartite-matching algorithm for post-processing. We pick subsets of the `OrgAddress` and `USPS` consisting of 100,000 records each for these experiments.

Our implementation is based on jaccard similarity and thus all techniques discussed in Section IV are applicable. We use locality sensitive hashing (LSH) as our signature scheme [30]. We pick the optimal parameters (performance-wise) of LSH such that the accuracy is 0.95 for threshold 0.8 (that is, every record pair with similarity greater than or equal to 0.8 is returned with probability 0.95). Our implementation pushes most computation into a DBMS and is based on the architecture discussed in [23].

1) Expansion from Transformations: Figure 6 shows the number of rules in the different transformation sets and the



(a) Execution Time (sec) (b) No. of Signatures (c) Join Size & No. of Candidate Pairs

Fig. 7. Performance of our algorithms

expansion that results from these sets, measured as the average number of strings generated by a single string from either input relation. In the presence of the transformations U+E(0.8), an input string on average generates 228.7 strings. Even when the edit threshold is 0.85, the expansion is 15.7. This example shows that the baseline algorithm can be prohibitively expensive, motivating the need for the optimizations proposed in this paper.

2) Execution Times: We measure the overall execution time of the above algorithms for computing a similarity join with various subsets of transformations. As noted above, the `baseline` algorithm is significantly more expensive than the other algorithms above and we do not report its execution time here. Thus, only the execution times for `Comp` and `Comp+Cl` are reported. Recall that the similarity join proceeds by first generating signatures for each input relation, joining the two on equality of signatures and for each distinct pair of records returned, checking whether the similarity predicate holds (which we call post-processing). In the presence of transformations, we also have two additional components to our execution: rule-matching and token clustering. The total time taken by rule-matching and token clustering even with the complete set of 200,000 transformations is less than 5 seconds. We thus focus on the total execution time of signature-generation, signature-equi-join and post-processing. Figure 7(a) shows these execution times (in seconds) on the Y-axis for different rule-sets, divided up into the time taken for each of the above components. (Note that some of the values in Figure 7 are too high to show pictorially; we explicitly specify the value for such cases.)

First, we observe that the `baseline` evaluation of the similarity join would proceed with the expanded relations which are a couple of orders of magnitude larger than the input relations (Figure 6) implying a proportional increase in the execution time in the presence of transformations. However, using our techniques we observe that the execution time under transformations is within a factor of 3-6 times of not having any transformations (indicated as *None* on the X-axis). Thus, our techniques yield up to two orders of magnitude improvement in performance.

Figure 7(a) also shows that token clustering yields significant benefits over and above signature compression. For instance, for the transformation set `U+E(0.8)`, invoking token clustering brings down the execution time by a factor of about 3 times.

3) Intermediate Result Size: We also study the benefits yielded by signature compression and token clustering by measuring the intermediate result sizes during the similarity join. For a signature-based algorithm such as ours, there are three measures of the intermediate result size: (1) the number of signatures computed, summed over both the input relations, (2) the size of the equi-join on the signatures, and (3) the number of distinct candidate pairs (this is different from the size of the equi-join since the same pair of strings can have more than one signature in common). These are plotted in Figure 7(b) and (c) (on the Y-axis). We observe that the number of signatures generated decreases by an order of magnitude by using signature compression and another order of magnitude when we also use token clustering. This again shows the benefit of our techniques.

Finally, in Figure 7(c), we also plot the size of the signature join and the number of candidate pairs generated for post-filtering (Y-axis) for various combinations of transformation rule-sets and algorithms used (X-axis). The main observation from this plot is that the benefits of token clustering in reducing the number of signatures generated does not come at the expense of an increase in the size of the signature-join (after all, we could trivially reduce the number of signatures by clustering all tokens into a single cluster).

VI. Summary

In this paper, we proposed a transformation-based framework to capture string variations such as synonyms and abbreviations. Unlike previously proposed approaches to handle these variations, (1) transformations are provided as an explicit input, and (2) the framework is expressive enough to capture not only equivalences but also more general forms of transformations. The semantics we proposed based on expanding the input relations are consistent with any choice of a similarity function. While this framework is powerful enough to capture rich variations between strings, it also exposes significant computational challenges which we addressed for a large class of similarity functions that can be implemented using signature-based algorithms. Our experiments over real

data showed that incorporating transformations significantly enhances record matching quality and that the performance of computing a similarity join is improved by orders of magnitude through our techniques.

VII. Acknowledgments

We thank Misha Bilenko for discussions on learnable string similarity measures.

References

[1] N. Koudas, S. Sarawagi, and D. Srivastava, "Record linkage: similarity measures and algorithms," in *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, June 2006, pp. 802–803.

[2] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. on Knowledge and Data Engg.*, vol. 19, no. 1, pp. 1–16, 2007.

[3] United States Postal Service, http://www.usps.com.

[4] "Wikipedia," http://en.wikipedia.org/.

[5] "DBLP," http://www.informatik.uni-trier.de/~ley/db/index.html.

[6] M. A. Jaro, "Advances in record linkage methodology as applied to matching the 1985 census of tampa," *American Statistical Association*, 1984.

[7] W. E. Winkler, "The state of record linkage and current research problems." *US Bureau of Census*, 1999.

[8] Trillium Software, www.trilliumsoft.com/trilliumsoft.nsf.

[9] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1970.

[10] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing and Management*, 1988.

[11] D. R. H. Miller, T. Leek, and R. M. Schwartz, "A hidden markov model information retrieval system," in *Proc. of the 22nd ACM SIGIR Conf. on Research and Development in Information Retrieval*, Aug. 1999, pp. 214–221.

[12] M. Bilenko and R. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proc. of the 9th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2003, pp. 39–48.

[13] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik, "Example-driven design of efficient record matching queries," in *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, Sept. 2007, pp. 23–27.

[14] S. Tejada, C. Knoblock, and S. Minton, "Learning domain-independent string transformation weights for high accuracy object identification," in *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, July 2002, pp. 350–359.

[15] S. Sarawagi and A. Bhamidipaty, "Interactive deduplication using active learning," in *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, July 2002, pp. 269–278.

[16] R. Ananthakrishna, S. Chaudhuri, and V. Ganti, "Eliminating fuzzy duplicates in data warehouses," in *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, Aug. 2002, pp. 586–597.

[17] X. Dong, A. Y. Halevy, and J. Madhavan, "Reference reconciliation in complex information spaces," in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, June 2005, pp. 85–96.

[18] P. Singla and P. Domingos, "Multi-relational record linkage." in *MRDM*, 2004.

[19] I. Bhattacharya and L. Getoor, "Collective entity resolution in relational data," *IEEE Data Engineering Bulletin*, vol. 29, no. 2, pp. 4–12, 2006.

[20] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[21] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, *et al.*, "Approximate string joins in a database (almost) for free," in *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, Sept. 2001, pp. 491–500.

[22] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004, pp. 743–754.

[23] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proc. of the 32nd Intl. Conf. on Very Large Data Bases*, Sept. 2006, pp. 918–929.

[24] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proc. of the 22nd Intl. Conf. on Data Engineering*, Apr. 2006.

[25] N. Bansal, A. Blum, and S. Chawla, "Correlation clustering," *Mach. Learn.*, vol. 56, no. 1-3, pp. 89–113, 2002.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. McGraw Hill, 2001.

[27] "RIDDLE: Repository of Information on Duplicate Detection, Record Linkage, and Identity Uncertainty," http://www.cs.utexas.edu/users/ml/riddle.

[28] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003, pp. 313–324.

[29] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava, "Benchmarking declarative approximate selection predicates." in *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, June 2007, pp. 353–364.

[30] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. of the 25th Intl. Conf. on Very Large Data Bases*, Sept. 1999, pp. 518–529.