Approximate String Joins with Abbreviations

Wenbo Tao
MIT
wenbo@mit.edu

Dong Deng
MIT
dongdeng@csail.mit.edu

Michael Stonebraker
MIT
stonebraker@csail.mit.edu

ABSTRACT

String joins have wide applications in data integration and cleaning. The inconsistency of data caused by data errors, term variations and missing values has led to the need for approximate string joins (ASJ). In this paper, we study ASJ with abbreviations, which are a frequent type of term variation. Although prior works have studied ASJ given a user-inputted dictionary of synonym rules, they have three common limitations. First, they suffer from low precision in the presence of abbreviations having multiple full forms. Second, their join algorithms are not scalable due to the exponential time complexity. Third, the dictionary may not exist since abbreviations are highly domain-dependent.

We propose an end-to-end workflow to address these limitations. There are three main components in the workflow: (1) a new similarity measure taking abbreviations into account that can handle abbreviations having multiple full forms, (2) an efficient join algorithm following the filter-verification framework and (3) an unsupervised approach to learn a dictionary of abbreviation rules from input strings. We evaluate our workflow on four real-world datasets and show that our workflow outputs accurate join results, scales well as input size grows and greatly outperforms state-of-the-art approaches in both accuracy and efficiency.

PVLDB Reference Format:
Wenbo Tao, Dong Deng, Michael Stonebraker. Approximate String Joins with Abbreviations. *PVLDB*, 11(1): 53-65, 2017.
DOI: 10.14778/3136610.3136615

1. INTRODUCTION

Join is one of the fundamental operations of relational DBMSs. Joining string attributes is especially important in areas such as data integration and data deduplication. However, in many applications the data between two tables is not consistent because of data errors, abbreviations, and missing values. This leads to a requirement for approximate string joins (ASJ), which is the topic of this paper.

1	`mit harvd division of health sciences technology`
	`harvard mit division of hst`

2	`camp activ compl`
	`campus activities complex`

3	`dept of athletics phys ed recreation`
	`daper`

4	`us and p`
	`urban studies and planning`

Figure 1: Some similar department names in the MIT Data Warehouse found by our approach.

In ASJ, the similarity between two strings is typically captured by traditional measures such as Jaccard and edit distance [13]. These measures calculate a value between 0 and 1, with higher values indicating more similarity. The join results are string pairs with similarity values not less than a user-inputted threshold.[1]

However, abbreviations, as a frequent type of term variation in real-world data, pose a significant challenge to the effectiveness of traditional similarity functions. Figure 1 shows several pairs of similar department names with abbreviations we found in the MIT Data Warehouse[2]. We can see none of these pairs have large edit or Jaccard similarity.

1.1 Previous Research and their Limitations

To address the ineffectiveness of traditional measures, two prior works JaccT [4] and SExpand [15] have studied how to find similar strings given a user-inputted dictionary of synonym rules. They both adopt the idea of generating a set of *derived strings* for any given string s, either by applying synonym rewritings to s [4] or appending applicable synonyms to s [15], and consider two strings similar if the similarity of the most similar pair of derived strings is not less than the threshold.

However, they have three common limitations.

Limitation 1: Low precision is likely when the same abbreviation has multiple full forms, as illustrated by the following example.

EXAMPLE 1. *Figure 2 shows two sets of strings and a user-inputted synonym dictionary. The abbreviation* `cs` *has five different synonyms (full forms).*

JaccT generates derived strings of a string by rewriting it using applicable synonym rules. For example, string b_1

[1]This threshold is usually greater than 0.5 to avoid outputting too many dissimilar strings.
[2]http://ist.mit.edu/warehouse

String set 1
a_1 `department of cs`
a_2 `campus security`
a_3 `ctr for control systems`

String set 2
b_1 `dept of computer science`
b_2 `career services`
b_3 `ctr for cs`

The synonym dictionary
$\langle$`dept` $\Leftrightarrow$ `department`$\rangle$
$\langle$`cs` $\Leftrightarrow$ `computer science`$\rangle$
$\langle$`cs` $\Leftrightarrow$ `campus security`$\rangle$
$\langle$`cs` $\Leftrightarrow$ `career services`$\rangle$
$\langle$`cs` $\Leftrightarrow$ `control systems`$\rangle$
$\langle$`cs` $\Leftrightarrow$ `chemical sciences`$\rangle$

Figure 2: Example strings and synonym rules to illustrate prior work [4,15].

= `dept of computer science` has two applicable synonym rules: $\langle$`dept` $\Leftrightarrow$ `department`$\rangle$ and $\langle$`cs` $\Leftrightarrow$ `computer science`$\rangle$. So b_1 has $2^2 = 4$ derived strings, namely `dept of cs`, `dept of computer science`, `department of cs` and `department of computer science`. Given two strings s_1 and s_2, JaccT searches for a derived string s'_1 of s_1 and a derived string s'_2 of s_2 such that $f(s'_1, s'_2)$ is maximized where f is some traditional similarity function[3] such as Jaccard[4] or edit similarity. Consider a_2 = `campus security` and b_2 = `career services`. They represent two different things and thus should not be considered similar. Yet because they have a common derived string `cs`, their JaccT similarity is 1. More generally, any two different full forms of `cs` will have a JaccT similarity of 1 because they share a common derived string `cs`. Therefore, low precision is a possible outcome.

SExpand generates derived strings in a different way by appending applicable synonyms. For instance, b_1 has the following four derived strings:

- *`dept of computer science`*
- *`dept of computer science cs`*
- *`dept of computer science department`*
- *`dept of computer science cs department`*

Then, in the same manner as JaccT, SExpand calculates the similarity between two strings by searching for the most similar pair of derived strings. Consider a_1 = `department of cs` and b_3 = `ctr for cs`. The tokens `cs` in two strings respectively stand for `computer science` and `control systems`. However, because SExpand searches for the most similar pair of derived strings, all five full forms of `cs` will be appended to both strings (e.g. a'_1 =`department of cs computer science campus security career services control systems chemical sciences`), leading to a high similarity of $\frac{11}{15}$ ($a'_1 \cap b'_3 = 11$ and $a'_1 \cup b'_3 = 15$) . More generally, given the dictionary in Figure 2, any two strings both containing `cs` will get a relatively high SExpand similarity.

Limitation 2: The join algorithms are not scalable.

Based on their similarity measures, both JaccT and SExpand propose a filter-verification algorithm to compute the join results. In the *filter* step, they first generate a signature set for each string which guarantees that two strings are similar only if their signature sets intersect. So they select a set of candidate string pairs whose signature sets intersect. In the *verification* step, for each candidate pair, they compute its real JaccT or SExpand similarity. However, to generate the signature set of a string, both JaccT and SExpand need to enumerate all its derived strings, which are $O(2^n)$ where n is number of applicable synonym rules in this string. This makes their join algorithms not scalable when n is large.

Limitation 3: The dictionary may not exist, as term abbreviations are highly domain-dependent.

Some previous works [5,16] learn synonyms from user-provided examples of matching strings. However, it is impractical for a human to manually compile a set of examples that is large enough to make this approach effective (in [5], 100,000 examples are used). To this end, the authors used traditional Jaccard-based ASJ to automatically generate examples (e.g. all string pairs with Jaccard similarity no less than 0.8). But we can see from Figure 1 that good examples of matching strings often have very small Jaccard similarity, making these approaches suffer from low recall. There is also some NLP work [3,8,17,19,21] discovering abbreviations in text. However, these works all rely on explicit indicators of abbreviations in text documents such as parentheses (e.g. `...At MIT (Massachusetts Institute of Technology)...`). In the string join setting, those indicators often do not exist. For example, no string in Figure 1 has parentheses.

1.2 Our Solution

To address these limitations, we propose an end-to-end workflow to effectively and efficiently solve ASJ with abbreviations.

We first propose a **new similarity measure** which is robust to abbreviations having multiple full forms. Instead of searching for the most similar pair of derived strings, we search for a derived string of one string that is the most similar to the other string, i.e., find a derived string s'_1 of s_1 that maximizes $f(s'_1, s_2)$ and vice versa. For example, in Figure 2, no derived string of `campus security` will have a positive Jaccard similarity with `career services` and vice versa. We present the formal description and analysis of this new measure in Section 2. Another big advantage brought by this new measure is that to compute the similarity, it has a **much smaller** search space than JaccT and SExpand, leading to much more efficient join computation (Section 3).

We then design an efficient join algorithm following the filter-verification framework. In contrast to JaccT and SExpand, we propose a polynomial time algorithm to calculate signatures **without iterating through all derived strings**. Calculating the new similarity measure is NP-hard, so we propose an efficient heuristic algorithm to verify all candidate string pairs after the filter step (Section 3).

We also present an **unsupervised** approach to learn an abbreviation dictionary[5] from input strings. We first extract a small set of candidate abbreviation rules by assuming that the length of the Longest Common Subsequence (LCS)[6] between an abbreviation and its full form should be equal or very close to the length of the abbreviation. For example, the LCS between `harvd` and its full form `harvard`

[3]For simplicity, in the following we use Jaccard as the default underlying similarity function, but our techniques can be modified to support other functions.

[4]The Jaccard similarity of two strings is the number of shared tokens divided by the number of all distinct tokens in two strings.

[5]We focus on discovering abbreviations in this paper, but our similarity measure and join algorithm can use other types of user-inputted synonym rules.

[6]A subsequence of a string s can be derived by deleting zero or more characters in s without changing the order of remaining characters. The longest common subsequence between two strings is a subsequence of both strings which has the most number of characters.

is harvd, which is exactly the abbreviation itself. This LCS-based assumption is **expressive**. It can capture a variety of abbreviation patterns such as prefix abbreviations (e.g. ⟨camp ⇔ campus⟩), single term abbreviations (e.g. ⟨harvd ⇔ harvard⟩) and acronyms (e.g. ⟨cs ⇔ computer science⟩). We search all pairs of a token in any string and a token sequence in any string that satisfy the LCS-based assumption. A naive approach which iterates through every pair runs quadratically in the number of strings and thus is not scalable. So we propose an **efficient** algorithm to retrieve LCS-based candidate rules without enumerating all pairs. We then use existing NLP techniques [3,19] to refine the LCS-based candidate rule set to remove unlikely abbreviations. Details are in Section 4.

To summarize, we make the following contributions:

- The first (to our best knowledge) end-to-end workflow solving ASJ with abbreviations.
- A new string similarity measure taking abbreviations into account which is robust to abbreviations having multiple full forms and enables much more efficient join computation than previous measures.
- An efficient join algorithm which generates signatures for strings in PTIME (as opposed to the exponential approach adopted by previous works) and calculates similarities efficiently using a heuristic algorithm.
- An unsupervised approach to efficiently learn a comprehensive abbreviation dictionary from input strings.
- Experimental results on four real-world datasets demonstrating that (1) our join results have high accuracy, (2) the entire workflow scales well as the input size grows and (3) individual parts of the workflow outperform state-of-the-art alternatives in both accuracy and efficiency.

2. STRING SIMILARITY MEASURE AND JOINS WITH ABBREVIATIONS

We start by describing how we formally model strings and abbreviation rules. Next, we introduce a new similarity measure which quantifies how similar two strings are given a dictionary of abbreviation rules[7]. The formal definition of the ASJ problem is presented at the end of this section.

2.1 Strings and Abbreviation Rules

We model a string s as a token sequence[8] $s(1), s(2), \ldots, s(|s|)$ where $|s|$ denotes the number of tokens in s. For example, if the tokenization delimiter is whitespace, then for $s =$ dept of computer science, $s(1) =$ dept, $s(4) =$ science and $|s| = 4$. The token sequence $s(i, j)$ $(1 \leq i \leq j \leq |s|)$ is the token sequence: $s(i), s(i+1), \ldots, s(j)$. For example, for $s =$ dept of computer science, $s(3, 4) =$ computer science.

An abbreviation rule is a pair of the form ⟨*abbr* ⇔ *full*⟩, where *abbr* is a token representing an abbreviation and *full* is a token sequence representing its full form. Example abbreviation rules in Figure 1 include ⟨camp ⇔ campus⟩, ⟨harvd ⇔ harvard⟩ and ⟨hst ⇔ health sciences technology⟩.

An abbreviation rule ⟨*abbr* ⇔ *full*⟩ is applicable in a string s if either *abbr* is a token of s, or *full* is a token subsequence of s. For example, ⟨camp ⇔ campus⟩ is applicable in camp activ compl and ⟨cs ⇔ computer science⟩ is applicable in department of computer science. We use *applicable side* to denote the side of an applicable rule that matches the string and *rewrite side* to denote the other side. For example, the applicable side of ⟨camp ⇔ campus⟩ *w.r.t* camp activ compl is camp whereas the rewrite side is campus.

[7]Section 4 will introduce how to learn this dictionary.
[8]A string can be tokenized by splitting it based on common delimiters such as whitespace.

2.2 Our Measure

We propose a new similarity measure pkduck[9] to address the issue with previous measures, i.e., low precision in the presence of abbreviations having multiple full forms. In contrast to JaccT and SExpand which search for the most similar pair of derived strings, we search for a derived string of one string that is the most similar to the other string.

Formally, we use pkduck$(\mathcal{R}, s_1, s_2)$ to denote the similarity between strings s_1 and s_2 given a dictionary $\mathcal{R}$ of abbreviation rules. pkduck$(\mathcal{R}, s_1, s_2)$ is calculated as follows. For a given string s, we use $A(s)$ to denote the set of applicable rules in s. Applying an applicable rule means rewriting the applicable side with the rewrite side. Following JaccT, we get a derived string s' of s by applying rules in a subset of $A(s)$ such that the applications of rules do not overlap (i.e. each original token is rewrote by at most one rule)[10]. We use $D(s)$ to denote the set of all derived string of s. Then,

$$\mathsf{pkduck}(\mathcal{R}, s_1, s_2) = \max \begin{cases} \max\limits_{s_1' \in D(s_1)} Jaccard(s_1', s_2) \\ \max\limits_{s_2' \in D(s_2)} Jaccard(s_1, s_2') \end{cases} \quad (1)$$

Example 2. *Consider in Figure 1 the fourth similar string pair $s_1 =$ us and p and $s_2 =$ urban studies and planning. Let $\mathcal{R}$ be $\{$⟨us ⇔ urban studies⟩, ⟨p ⇔ planning⟩$\}$. We have $A(s_1) = A(s_2) = \mathcal{R}$. $D(s_1)$ contains the following strings:*

- *us and p*
- *us and planning*
- *urban studies and p*
- *urban studies and planning*

The fourth derived string has the highest Jaccard similarity (1.0) with s_2, thus $\max\limits_{s_1' \in D(s_1)} Jaccard(s_1', s_2) = 1$. Similarly, we can calculate that $\max\limits_{s_2' \in D(s_2)} Jaccard(s_1, s_2') = 1$. Thus pkduck$(\mathcal{R}, s_1, s_2)$ equals 1.

Let us consider again the dissimilar string pairs in Figure 2 which both JaccT and SExpand consider similar. For $a_2 =$ campus security and $b_2 =$ career services, their pkduck similarity is 0. Similarly, for $a_1 =$ department of cs and $b_3 =$ ctr for cs, their pkduck similarity is only $\frac{1}{5}$. We can see this new measure is robust to abbreviations having multiple full forms.

Analysis. Because any string is also a derived string of itself, the pkduck similarity of two strings is always no less than their Jaccard similarity. That is to say, pkduck subsumes Jaccard and other traditional similarity functions that can replace Jaccard in Equation 1.

Compared to JaccT and SExpand, not only can the pkduck measure handle abbreviations having multiple full forms, the

[9]pkduck is an abbreviation of the famous Chinese dish Peking Duck.
[10]As shown in [4], calculating derived string becomes intractable if we allow a token generated by some rule application to participate in other rule applications.

search space is also significantly reduced. Let n denote the number of applicable rules in a string. Both JaccT and SExpand have a search space of $O(2^{2^n})$ whereas the search space of pkduck is only $O(2^n)$. As will be shown in Section 3, this will lead to more efficient join computation. We will also show in Section 3 that calculating pkduck similarity values is NP-hard and propose an efficient heuristic algorithm.

Note that it is not viable to generate derived strings as SExpand does by appending strings with applicable abbreviations/full forms. The reason is that the original tokens are preserved, so if we search for a derived string of one string that is the most similar to the other string, the resulting similarity will often be lower than expected. Consider $s_1 =$ `harvd`, $s_2 =$ `harvard` and a given applicable rule $\langle$`harvd` $\Leftrightarrow$ `harvard`$\rangle$. If we generate derived strings as SExpand does, $D(s_1)$ will have two derived strings `harvd` and `harvd harvard`. So $\max_{s_1' \in D(s_1)} Jaccard(s_1', s_2)$ is only $\frac{1}{2}$ and vice versa. In contrast, their pkduck similarity is 1.

2.3 Problem Formulation

The ASJ problem we study receives as input a set S of strings, and a real number $\theta \in [0, 1]$. We learn from S a dictionary $\mathcal{R}$ of abbreviation rules (Section 4). The output is the set of string pairs $\{s_1, s_2\} \in S \times S$ where pkduck$(\mathcal{R}, s_1, s_2) \geq \theta$. We focus on self-join in this paper for ease of presentation, but our approach easily extends to joining two sets of strings.

3. JOIN ALGORITHM

Our join algorithm follows the filter-verification framework [4,9,15]. In the filter step, we use a signature-based scheme to select from the $O(|S|^2)$ search space a small set of candidate string pairs. In the verification step, we calculate the pkduck similarity for every candidate string pair and output those with pkduck similarities not less than θ.

We review the conventional signature scheme in Section 3.1. Section 3.2 describes a new filtering scheme for pkduck by extending the conventional signature scheme. Naively calculating the new signatures is not scalable, so we introduce a PTIME signature generation algorithm in Section 3.3. In Section 3.4, we propose an efficient heuristic algorithm to calculate pkduck similarities.

3.1 Conventional Signature Schemes

Conventional signature schemes [6,9,14,26] construct a signature set for each input string and filter out those string pairs whose signature sets do not intersect. We use the widely-adopted prefix-filter [9] to briefly illustrate this idea.

We assume in the rest of Section 3 a global ordering of all tokens constituting the strings in S is determined[11]. Given a similarity threshold θ, the prefix-filter signature set of s, denoted as $Sig_{\mathtt{pf}}(s)$, is a prefix token sequence of s after sorting its tokens based on the global ordering. The length of the prefix is determined by a function I_θ of $|s|$: $I_\theta(|s|) = \lfloor (1-\theta) \cdot |s| \rfloor + 1$. This function I_θ ensures the signature property that two strings have a Jaccard similarity not less than θ only if $Sig_{\mathtt{pf}}(s_1)$ and $Sig_{\mathtt{pf}}(s_2)$ intersect[12]. So in traditional Jaccard-based ASJ, we can safely prune those pairs where $Sig_{\mathtt{pf}}(s_1)$ and $Sig_{\mathtt{pf}}(s_2)$ do not intersect, and see the remaining pairs as candidate pairs.

[11]Prefix-filter works correctly for any token ordering but the ascending order of frequency generally achieves the best performance [6].

[12]See [9] for proof.

Global token ordering:
`planning` $<$ `us` $<$ `urban` $<$ `studies` $<$ `p` $<$ `and`
$\mathcal{R} = \{\langle$`us` $\Leftrightarrow$ `urban studies`$\rangle$, $\langle$`p` $\Leftrightarrow$ `planning`$\rangle\}$

$s_1 =$ `us and p`	$Sig_{\mathtt{pf}}(s_1) = \{$`us`$\}$
$s_2 =$ `urban studies and planning`	$Sig_{\mathtt{pf}}(s_2) = \{$`planning,urban`$\}$

(a) The first table contains a global token ordering and a rule dictionary. Strings s_1 and s_2 and their prefix-filter signature sets are shown in the second table.

s_1'	$Sig_{\mathtt{pf}}(s_1')$
1. `us and p`	$\{$`us`$\}$
2. `us and planning`	$\{$`planning`$\}$
3. `urban studies and p`	$\{$`urban,studies`$\}$
4. `urban studies and planning`	$\{$`planning,urban`$\}$

$Sig_{\mathrm{u}}(s_1) = \{$`planning,us,urban,studies`$\}$

(b) An illustration of calculating $Sig_{\mathrm{u}}(s_1)$.

Figure 3: An illustration of our filtering scheme. The user-inputted join threshold θ is 0.7.

Consider the example in Figure 3(a). A global token ordering and a rule dictionary containing two rules are given. The user-inputted threshold θ is 0.7. s_2 has 4 tokens, so $Sig_{\mathtt{pf}}(s_2)$ contains the $I_\theta(4) = \lfloor (1-\theta) \cdot 4 \rfloor + 1 = 2$ smallest tokens of s_2, i.e., `planning` and `urban`.

3.2 Filtering Scheme for pkduck

Because calculating pkduck similarities involves abbreviation rules in $\mathcal{R}$, conventional signature schemes are no longer sufficient. For example, in Figure 3(a), pkduck$(\mathcal{R}, s_1, s_2) = 1 \geq \theta$, but their prefix-filter signature sets do not intersect.

In this subsection, we extend the prefix-filter[13] signature approach to design a filtering scheme for pkduck.

We first observe that pkduck$(\mathcal{R}, s_1, s_2) \geq \theta$ if and only if

$$\max_{s_1' \in D(s_1)} Jaccard(s_1', s_2) \geq \theta \qquad (2)$$

or

$$\max_{s_2' \in D(s_2)} Jaccard(s_1, s_2') \geq \theta \qquad (3)$$

Next, we calculate for each string s a signature set $Sig_{\mathrm{u}}(s)$, which is the union of the prefix-filter signature sets of all its derived strings, i.e.,

$$Sig_{\mathrm{u}}(s) = \bigcup_{s' \in D(s)} Sig_{\mathtt{pf}}(s')$$

Figure 3(b) shows the prefix-filter signatures of all derived strings of s_1 and $Sig_{\mathrm{u}}(s_1)$.

The following theorem involving $Sig_{\mathrm{u}}(s)$ shows a necessary condition for two strings to be similar.

[13]Our filtering scheme can be fit in by any conventional signature scheme, but we focus on extending prefix-filter in this paper. Section 6 includes an explanation on this choice.

Theorem 1. *s_1 and s_2 satisfy Equation 2 only if $Sig_{\mathtt{u}}(s_1) \cap Sig_{\mathtt{pf}}(s_2) \neq \varnothing$ and vice versa for Equation 3.*

Due to space constraints, we put all proofs of theorems into our extended technical report [1].

Theorem 1 states that, we can select as candidate string pairs all $\{s_1, s_2\}$ where $Sig_{\mathtt{u}}(s_1)$ and $Sig_{\mathtt{pf}}(s_2)$ intersect, or $Sig_{\mathtt{u}}(s_2)$ and $Sig_{\mathtt{pf}}(s_1)$ intersect. The pairs not selected can be safely pruned because they do not satisfy Equation 2 or Equation 3 and thus will have a pkduck similarity less than θ. In Figure 3, $\{s_1, s_2\}$ is selected as a candidate string pair because $Sig_{\mathtt{u}}(s_1) \cap Sig_{\mathtt{pf}}(s_2) \neq \varnothing$.

Comparison with JaccT [4] and SExpand [15]. Both JaccT and SExpand calculate the similarity between two strings by searching for the most similar pair of derived strings. So the authors of JaccT and SExpand propose to select as candidate string pairs all $\{s_1, s_2\}$ where $Sig_{\mathtt{u}}(s_1)$ and $Sig_{\mathtt{u}}(s_2)$ intersect. We can see that our filtering framework selects significantly fewer candidate pairs than JaccT and SExpand because $Sig_{\mathtt{pf}}(s)$ is subsumed by $Sig_{\mathtt{u}}(s)$ and is expected to have much fewer tokens. Therefore, in addition to handling abbreviations having multiple full forms, the pkduck measure also leads to better performance.

3.3 PTIME Signature Generation

An important yet unanswered question is how to calculate the signature set $Sig_{\mathtt{u}}(s)$, given an input string s. JaccT and SExpand both calculate $Sig_{\mathtt{u}}(s)$ in a brute-force manner by enumerating all possible derived strings, which are $O(2^n)$. This is not scalable when n is large. In this subsection, we propose a PTIME algorithm to calculate $Sig_{\mathtt{u}}(s)$.

3.3.1 Basic Flow of the Algorithm

Figure 4 shows the basic flow of our algorithm.

The main idea is to loop over each possible signature t and test if t is in $Sig_{\mathtt{u}}(s)$. The motivation is that the set $\mathcal{T}$ of tokens from which $Sig_{\mathtt{u}}(s)$ can draw is small and polynomial. $\mathcal{T}$ only contains tokens in s and any rule in $A(s)$, so $|\mathcal{T}|$ is $O(k \cdot n)$ where k is the maximum number of tokens a string in S could have, i.e., $\max_{s \in S} |s|$.

Given a token t, *isInSigU* is the key function testing if t is in $Sig_{\mathtt{u}}(s)$. We implement this *isInSigU* function using the idea of categorizing derived strings of s based on the number of tokens they have. W_l is the set of derived strings of s containing t and having l tokens. X_l is the least number of smaller tokens (tokens preceding t in the global ordering) that one string in W_l could have (Section 3.3.2 will describe how to calculate X_l). We have the following theorem.

Theorem 2. *t is in $Sig_{pf}(s')$ of a string $s' \in W_l$ if and only if $X_l + 1 \leq I_\theta(l)$.* [14]

For example, let s be s_1 and t be `studies` in Figure 3. W_4 contains the third and the fourth derived strings of s_1, which both have four tokens and contain the token `studies`. The third derived string contains one smaller token `urban` while the fourth derived string contains two, so $X_4 = 1$. Then we have $X_4 + 1 = 2 \leq I_\theta(4) = 2$. Therefore, `studies` is in the prefix-filter signature set of one string in W_4 (the third derived string in Figure 3(b)).

To check if t is in $Sig_{\mathtt{u}}(s)$, we just need to check if there exists an l *s.t.* $X_l + 1 \leq I_\theta(l)$. Note that l values greater than

[14]See the extended technical report [1] for proof.

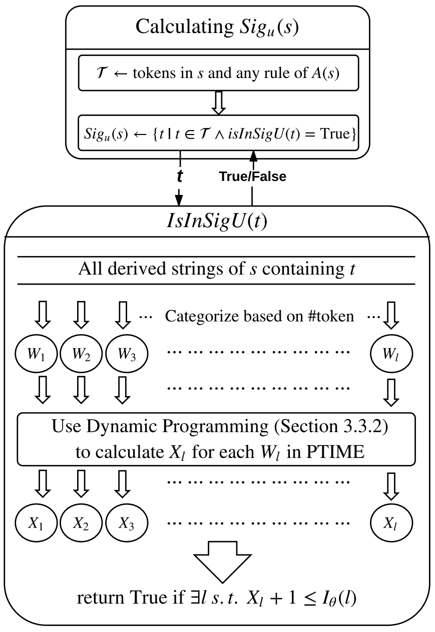

Calculating $Sig_u(s)$
$\mathcal{T} \leftarrow$ tokens in s and any rule of $A(s)$
$Sig_u(s) \leftarrow \{t \mid t \in \mathcal{T} \wedge isInSigU(t) = \text{True}\}$
t
True/False
$IsInSigU(t)$
All derived strings of s containing t
Categorize based on #token
W_1 W_2 W_3 W_l
Use Dynamic Programming (Section 3.3.2) to calculate X_l for each W_l in PTIME
X_1 X_2 X_3 X_l
return True if $\exists l$ $s.t.$ $X_l + 1 \leq I_\theta(l)$

Figure 4: The flow of calculating $Sig_u(s)$.

$\lfloor \frac{k}{\theta} \rfloor$ do not need to be considered because derived strings having more than this number of tokens will have a Jaccard similarity less than θ with any string in S.

3.3.2 Calculating X_l using Dynamic Programming

We propose a dynamic programming algorithm to calculate X_l. Formally, let $g(i, l, o)$ denote the least number of smaller tokens a string satisfying the following constraints could have: (1) it is a derived string of the first i tokens of s (i.e. $s(1, i)$), (2) it has l tokens and (3) it contains token t if $o = T$ and does not contain token t if $o = F$. It can be seen that $X_l = g(|s|, l, T)$.

We can recursively calculate $g(i, l, o)$ by considering the different cases of $s(i)$'s participation in rule applications. Specifically, $g(i, l, T)$ is the minimum over the following things:

1. $g(i-1, l-1, T) + small(s(i))$, where $small(s(i))$ evaluates to 1 if $s(i)$ is a smaller token and 0 otherwise. In this case, token $s(i)$ does not participate in any rule application.

2. $g(i-1, l-1, F) + small(s(i))$, if $s(i)$ is t. $s(i)$ still does not participate in any rule application in this case. Yet because we use $g(i-1, l-1, F)$ to calculate $g(i, l, T)$, $s(i)$ must be token t.

3. $\min_{r \in end(i)} g(i - aside(r), l - rside(r), T) + small(r)$, where $end(i)$ is the set of rules in $A(s)$ that end at $s(i)$, $aside(r)$ and $rside(r)$ are respectively the size of the applicable side and rewrite side of a rule r, and $small(r)$ is the number of smaller tokens in the rewrite side of r. In this case, we are trying to apply a rule ending at $s(i)$.

4. $\min_{r \in end(i) \cap A_t(s)} g(i - aside(r), l - rside(r), F) + small(r)$, where $A_t(s)$ is the set of rules in $A(s)$ whose rewrite

side contains t. We are still trying to apply rules ending at $s(i)$, but similar to the second case, the rewrite side of a rule must contain t.

The recursion base is $g(0,0,F)=0$. The calculation of $g(i,l,F)$ is similar and simpler.

EXAMPLE 3. *Suppose we want to check if* `studies` *is in* $Sig_u(s_1)$ *in Figure 3, where* s_1 = `us and p`. *Consider the calculation of* $g(1,2,T)$. *Token* $s_1(1)$ *must participate in a rule application, otherwise the resulting derived string will not have two tokens. There is only one applicable rule* ⟨`us` ⇔ `urban studies`⟩ *ending at* $s_1(1)$*, which has* `studies` *as well as one smaller token* `urban` *in the rewrite side. So* $g(1,2,T)=g(0,0,F)+1=1$. *Similarly, we can get* $g(2,3,T)=g(3,4,T)=1$. *Finally, because* $X_4+1=g(3,4,T)+1 \leq I_\theta(4)$*, we know* `studies` *is in* $Sig_u(s_1)$.

Complexity Analysis. There are $O(k\cdot n)$ tokens in $\mathcal{T}$ which we will loop over. The time complexity of calculating X_l values is $O(k\cdot n)$ because each applicable rule is considered once for each l. Therefore, the overall time complexity of generating $Sig_u(s)$ for a string is $O(k^2n^2)$.

Further Optimization. Consider the third and fourth cases of the recursive calculation of $g(i,l,T)$. We need to consider every applicable rule ending at $s(i)$. We observe that this process can be further optimized by categorizing rules ending at $s(i)$ based on $aside(r)$ and $rside(r)$. For the same combination of $aside(r)$ and $rside(r)$, we only care about the rule that has the fewest smaller tokens in the rewrite side. We can then precompute this information and reduce the overall signature generation complexity from $O(k^2n^2)$ to $O(k^3n)$. This optimization technique, we call *rule compression*, is especially effective given the dictionary learned in Section 4 because n is often much greater than k as we will show in our experiments.

3.4 Candidate Verification

The last step of the join algorithm is to verify candidate string pairs. Unfortunately, the following theorem states the hardness of calculating pkduck similarities.

THEOREM 3. *Calculating pkduck similarities is NP-Hard.* [15]

So we propose an efficient heuristic algorithm to calculate pkduck similarities for all candidate pairs. In the following, we only consider calculating $\max_{s'_1\in D(s_1)} Jaccard(s'_1,s_2)$. Calculating $\max_{s'_2\in D(s_2)} Jaccard(s_1,s'_2)$ is the same.

The basic idea is to keep applying the most 'useful' applicable rule until no applicable rule remains. The usefulness $U(r)$ of an applicable rule $r\in A(s_1)$ is defined as follows. Suppose there are c common tokens between the rewrites side of r and s_2, $U(r)$ is defined as $\frac{c}{rside(r)}$, which is the percentage of 'useful' tokens in the rewrite side.

Main steps of this algorithm are:

(1) Select an applicable rule $r\in A(s_1)$ with the largest $U(r)$ and apply it;

(2) If no r exists or $U(r)=0$, goto (6);

(3) Remove from $A(s_1)$ all rules whose applicable side overlaps with that of r;

(4) Remove from s_2 tokens that intersect with the rewrite side of r;

(5) Recalculate rule usefulnesses, goto (1);

(6) return $Jaccard(s'_1,s_2)$.

The time complexity of this algorithm is $O(k\cdot n)$ because the loop from step (1) to (5) repeats at most k times and each loop takes $O(n)$ time to select the most useful rule and update rule usefulnesses.

The algorithms for calculating JaccT and SExpand similarities have the time complexity of respectively $O(2^{2n})$ and $O(k\cdot n^2)$, which are not scalable when n is large. The lower time complexity of our algorithm results from the smaller search space of pkduck. pkduck only searches for a derived string of one string that is the most similar to the other string, so step (1) in the above algorithm only needs to consider applicable rules of one string. In contrast, SExpand has to consider applicable rules of both strings, resulting in a time complexity quadratic in n, while JaccT uses a brute-force approach iterating over all pairs of derived strings.

4. LEARNING ABBREVIATION RULES

In this section, we present an unsupervised approach to learn an abbreviation dictionary from input strings. Given a set of strings S, every pair of a token in S and a token sequence in S can form a potential abbreviation rule. However, the number of pairs is huge and most of the pairs do not contain real abbreviations.

To this end, in Section 4.1, we describe a Longest Common Subsequence (LCS) based approach to efficiently extract a small set of candidate abbreviation rules. In Section 4.2, we discuss how to use existing NLP techniques to further refine the LCS-based candidate rule set.

4.1 Generating LCS-based Candidate Rules

4.1.1 The LCS-based Assumption

We assume that the length of the LCS between an abbreviation and its full form is usually equal or very close to the length of the abbreviation. Consider the following abbreviation rules in Figure 1:

- ⟨`compl` ⇔ `complex`⟩,
- ⟨`harvd` ⇔ `harvard`⟩,
- ⟨`hst` ⇔ `health sciences technology`⟩.

Characters in blue form the LCS between the abbreviation and the full form in each rule. We can see that in each rule, the length of the LCS equals the length of the abbreviation (i.e. the abbreviation is a subsequence of its full form). There are also rules where the length of the LCS is very close but not exactly equal to the length of the abbreviation such as ⟨`bill` ⇔ `william`⟩.

More formally, we make the assumption that, in an abbreviation rule ⟨*abbr* ⇔ *full*⟩, the difference between the length of LCS(*abbr, full*) and the length of *abbr* is smaller than or equal to a small threshold δ (typically 0 or 1 in practice), i.e., $|abbr|-|LCS(abbr, full)|\leq\delta$.

This LCS-based assumption is expressive and can capture a variety of frequent abbreviation patterns, including prefix abbreviations (e.g. ⟨`camp` ⇔ `campus`⟩), single term abbreviations (e.g. ⟨`mgmt` ⇔ `management`⟩) and acronyms (e.g. ⟨`cs`

[15]See the extended technical report [1] for proof.

⇔ `computer science`⟩). Moreover, it can also capture some common misspellings (e.g. ⟨`manuver` ⇔ `maneuver`⟩) and erroneous concatenations (e.g. ⟨`newyork` ⇔ `new york`⟩) which frequently appear in manually entered data.

Therefore, we generate a set of LCS-based candidate rules by identifying all pairs of a token and a token sequence from strings in S that satisfy the LCS-based assumption. The formal definition of the LCS-based candidate rule generation problem is presented in Definition 1.

DEFINITION 1 (LCS-BASED RULE GENERATION). *Given a set S of strings and a non-negative integer δ, find all distinct rules $\langle abbr \Leftrightarrow full \rangle$ s.t. (1) $|abbr| - |LCS(abbr, full)| \leq \delta$ and (2) there exist two strings s_1 and s_2 in S s.t. abbr is a token of s_1 and full is a token subsequence of s_2.*

4.1.2 *Efficient Calculation*

The key challenge in solving the candidate rule generation problem is how to efficiently identify the candidate rules. We first describe a naive algorithm.

A Naive Algorithm. The most straightforward way to identify the candidate rules is to enumerate all possible pairs of a token (*abbr*) and a token sequence (*full*) in S and use the following standard dynamic programming procedure to compute $LCS(abbr, full)$ and check whether the LCS-based assumption is satisfied:

$$f(i,j) = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ f(i-1, j-1) + 1, & i, j > 0 \text{ and } s_1[i] = s_2[j] \\ \max(f(i, j-1), f(i-1, j)), & i, j > 0 \text{ and } s_1[i] \neq s_2[j] \end{cases}$$

where s_1 is *abbr*, s_2 is the concatenation of tokens in *full*, $s_x[i]$ denotes the i-th character of string s_x (counting from 1) and $f(i, j)$ denotes the LCS between the prefix of s_1 with length i and the prefix of s_2 with length j.

There are $O(|S| \cdot k)$ tokens and $O(|S| \cdot k^2)$ token sequences (recall that k is the maximum number of tokens that a string in S could have). So the search space is $O(|S|^2 \cdot k^3)$. Even if we see k as a relatively small constant, this search space is still quadratic in $|S|$, making this naive algorithm not scale when $|S|$ is large.

Despite this huge search space, the number of candidate rules is expected to be much smaller due to the LCS-based assumption. This motivates us to design the following efficient algorithm to retrieve LCS-based candidate rules without enumerating all pairs.

Efficient trie-based Algorithm. For simplicity, we first discuss the case of $\delta = 0$, i.e., identifying all pairs of a token t and a token sequence ts in S where t **is a subsequence of** ts. We will discuss later how to extend our algorithm to the case of $\delta > 0$.

Our algorithm is based on the trie structure [11], which is widely used in Information Retrieval to efficiently index a set of strings and search a query string in linear time in the string length. Figure 5 shows the trie constructed for four strings. The string formed by concatenating the characters on the path from the root to a node is called the prefix of that node. For example, the prefix of node 10 is `dap`. We say a trie node **matches** a string if the string equals the prefix of this node. In Figure 5, every leaf node (in orange) uniquely matches one original string[16].

[16]Indexed strings are often appended a special character to ensure no string is the prefix of another.

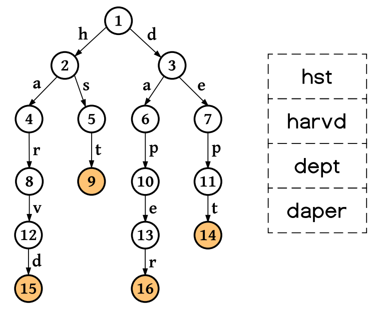


Figure 5: A trie constructed by four strings.

We first create a trie T to index all tokens in S. Then, the core problem we need to solve is for each token sequence ts in S, find all leaf nodes in T that matches some subsequence of ts.

Our algorithm recursively calculates this set of leaf nodes by considering the characters in ts one by one. Formally, we use $F(i)$ to denote the set of nodes in T that match some subsequence of the first i characters of ts (denoted as $ts[1..i]$). The recursion base is $F(0) = \{\text{root of } T\}$, meaning the root matches the empty subsequence. What we need are all leaf nodes in $F(|c|)$ where c is the concatenation of tokens in ts.

For $i > 0$, $F(i)$ is calculated based on $F(i-1)$ as follows. According to the definition of $F(i)$, $F(i-1)$ should be a subset of $F(i)$ because subsequences of $ts[1..i-1]$ are also subsequences of $ts[1..i]$. So we initially set $F(i)$ to $F(i-1)$. Let c_i denote the i-th character of ts. Recall that each node $x \in F(i-1)$ matches a subsequence of $ts[1..i-1]$. If x has an outgoing edge with a character label c_i to a child x', we can append c_i to the subsequence that x matches, and form a new subsequence of $ts[1..i]$. This new subsequence of $ts[1..i]$ is matched by the child x', which we add to $F(i)$.

Every leaf node $x \in F(|c|)$ matches a token in S which is also a subsequence of ts. Therefore, the prefix of x and ts can form a valid LCS-based candidate rule.

We use the following example to illustrate the recursive calculation of the F sets.

EXAMPLE 4. *Let T be the trie in Figure 5 and ts be `health st`. The F sets are in the following table.*

i	c_i	$F(i)$
0		{1}
1	`h`	{1, 2}
2	`e`	{1, 2}
3	`a`	{1, 2, 4}
4	`l`	{1, 2, 4}
5	`t`	{1, 2, 4}
6	`h`	{1, 2, 4}
7	`s`	{1, 2, 4, 5}
8	`t`	{1, 2, 4, 5, 9}

Let us consider the calculation of $F(8)$. $F(7)$ contains nodes 1, 2, 4 and 5 which respectively match the empty subsequence, subsequences `h`, `ha` and `hs`. The 8-th character is `t`. In $F(7)$, only node 5 has an outgoing edge with character label `t`. Thus, node 9 is added into $F(8)$.

Algorithm 1: LCS-based Candidate Rule Generation

```
Input: A set $S$ of tokenized strings.
Output: A set $R$ of LCS-based candidate rules.
1  $T \leftarrow$ an empty trie;
2  for each token $t$ in $S$ do
3      Insert $t$ into $T$;
4  $R \leftarrow \varnothing$;
5  $root \leftarrow$ the root node of $T$;
6  for each token sequence $ts \in S$ do
7      $c \leftarrow$ concatenation of tokens in $ts$;
8      $F(0) \leftarrow \{root\}$;
9      for $i \leftarrow 1$ to $|c|$ do
10         $F(i) \leftarrow F(i-1)$;
11         for each $x \in F(i-1)$ do
12             if node $x$ has an outgoing edge with character $c_i$ then
13                 $x' \leftarrow$ the corresponding child;
14                 Insert $x'$ into $F(i)$;
15     for each $x \in F(|c|)$ do
16         if node $x$ is a leaf node then
17             $abbr \leftarrow$ the prefix of $x$;
18             $full \leftarrow ts$;
19             Insert $\langle abbr \Leftrightarrow full \rangle$ into $R$;
20 return $R$;
```

After calculating all F sets, node 9 is the only leaf node in $F(8)$, which suggests one valid LCS-based candidate rule $\langle$`hst` $\Leftrightarrow$ `health st`$\rangle$.

Algorithm 1 presents the pseudo-code of our algorithm. Lines 9∼14 show the recursive calculation of the F sets. Lines 15∼19 generate an LCS-based candidate rule for each leaf node in $F(|c|)$.

Extension to $\delta > 0$. To support positive δ values, we can modify the above algorithm as follows. First, for any token t and token sequence ts, if $|t| - |LCS(t, ts)| \leq \delta$, $LCS(t, ts)$ must be one of the subsequences of t having at least $|t| - \delta$ characters. Because δ is small, we can just insert all these subsequences into the trie index. Second, in Line 17 of Algorithm 1, if the prefix of x is not a token in S, we should assign to *abbr* the original token in S that x corresponds to.

Performance Analysis. Constructing a trie for a set of strings runs linearly in the sum of the string length. So the cost of building a trie is very small.

For a given token sequence ts, compared to the naive approach, our algorithm does not iterate over all leaf nodes and only visits trie nodes that match a subsequence of ts. In the worst case, this number could be as large as the size of T. But we will empirically show in Section 5 that this number is very small on real-world datasets and that this algorithm is very efficient and scales well.

4.2 Refinement

Some of the LCS-based candidate rules are not likely to contain real abbreviations. Consider the rule $\langle$`te` $\Leftrightarrow$ `database`$\rangle$. Although it satisfies the LCS-based assumption, in practice it is unlikely that people will abbreviate `database` as `te`. Other examples include $\langle$`ia` $\Leftrightarrow$ `information`$\rangle$ and $\langle$`ay` $\Leftrightarrow$ `dictionary`$\rangle$. Including those kinds of rules in the dictionary not only incurs unnecessary overhead in the ASJ process but also hurts the accuracy of join results since they falsely equate a pair of a token and a token sequence.

Thus it is important to design a rule refiner[17] to filter out rules that are not likely to contain real abbreviations from the LCS-based candidate rule set.

We implement the rule refiner combining heuristic patterns from existing NLP works [2,3,19,21]. By analyzing large amounts of text, these works create handcrafted heuristic patterns to filter out unlikely abbreviations from a set of candidates extracted from text. An example pattern is that if a vowel letter in *full* preceded by a consonant letter matches a letter in *abbr*, the preceding consonant letter must also match a letter in *abbr*. For example, in $\langle$`te` $\Leftrightarrow$ `database`$\rangle$, the `e` in `database` matches the `e` in `te`, but the preceding `s` does not match any letter in `te`. Thus this rule will be filtered out by this pattern. Similarly, $\langle$`ia` $\Leftrightarrow$ `information`$\rangle$ and $\langle$`ay` $\Leftrightarrow$ `dictionary`$\rangle$ will also be filtered out by this pattern. Interested readers should refer to the papers for more patterns and details.

There are also some supervised learning approaches [8,18,28] that can be adopted to implement the rule refiner. However, we focus on building an automatic end-to-end workflow and leave the comparison of different implementations for future work.

5. EXPERIMENTAL RESULTS

In this section, we describe our experiments and results. The goals of our experiments are (1) evaluating the end-to-end accuracy and efficiency of our workflow and (2) comparing individual parts of our workflow with state-of-the-art alternatives. Section 5.1 describes the experimental settings. Section 5.2∼5.4 respectively evaluates our similarity measure, join algorithm and abbreviation dictionary.

5.1 Experimental Settings

Datasets. We use four real-world datasets:

- DISEASE contains 634 distinct disease names. The average number of tokens per string (denoted as a) is 2.1 and the maximum number of tokens a string could have (denoted as k) is 10.
- DEPT contains 1,151 distinct department names we collect from the MIT Data Warehouse by merging all department name columns in all tables. The a value and k value are respectively 3.4 and 10.
- COURSE contains 20,049 distinct course names we collect from the MIT Data Warehouse by merging all course name columns. The a value and k value are respectively 4.1 and 17.
- LOCATION contains 112,394 distinct location names (street names, city names, etc.) we collect from 7,515 tables crawled from Data.gov. The a value and k value are respectively 3.6 and 13.

Sample similar strings. Figure 1 shows sample similar strings in the DEPT dataset found by our approach. Figures 6, 7 and 8 show some sample similar strings in DISEASE,

[17]The characteristics of abbreviations are language-dependent [18,22], but we focus on English abbreviations in this paper.

Table 1: Comparing pkduck with SExpand and traditional non-weighted Jaccard (using LCS-based dictionary with $\delta = 0$). JaccT is not included because it did not terminate in one week. P, R and F respectively stand for Precision, Recall and F-measure.

	θ	Disease			Dept			Course			Location		
		P	R	F	P	R	F	P	R	F	P	R	F
Jaccard	0.7	0.00	0.00	0.00	0.64	0.18	0.28	0.45	0.08	0.14	0.53	0.21	0.30
	0.8	0.00	0.00	0.00	0.60	0.08	0.14	0.50	0.03	0.06	0.25	0.01	0.02
	0.9	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.01	0.00	0.00	0.00
SExpand	0.7	0.71	0.75	0.73	0.14	0.93	0.25	0.01	1.00	0.03	0.01	0.98	0.02
	0.8	0.83	0.73	0.78	0.30	0.90	0.45	0.04	0.99	0.07	0.01	0.97	0.03
	0.9	0.99	0.69	0.82	0.42	0.62	0.50	0.15	0.89	0.25	0.02	0.74	0.05
pkduck	0.7	0.88	0.74	0.81	0.75	0.79	**0.77**	0.66	0.77	**0.71**	0.46	0.55	**0.50**
	0.8	0.97	0.72	0.83	0.83	0.60	0.70	0.78	0.58	0.66	0.74	0.28	0.40
	0.9	0.99	0.72	**0.83**	0.96	0.37	0.54	0.95	0.39	0.56	0.80	0.24	0.37

1	`cmt disease`
	`charcot marie tooth disease`

2	`ic/pbs`
	`interstitial cystitis/painful bladder syndrome`

Figure 6: Sample similar strings in Disease.

1	`prac it mgmt`
	`practical information technology management`

2	`seminar in marine geology and geophysics at mit`
	`sem in mg g at mit`

3	`quant analysis emp methods`
	`quant analysis empirical meth`

Figure 7: Sample similar strings in Course.

Course and Location. These demonstrate that abbreviations are a common type of term variation in real-world data and that solving ASJ with abbreviations has pragmatic benefits.

Quality Metrics of Join Results. We have all pairs of strings that refer to the same disease (i.e. the ground truth) for the Disease dataset. For Dept, we manually construct the ground truth. For Course and Location, we randomly sample 5% strings and manually construct the ground truth. Then we test the Precision (P), Recall (R) and F-measure (F), where $F = \frac{2 \times P \times R}{P+R}$. All reported P, R and F numbers are rounded to the nearest hundredth.

Implementation & Environment. Non-weighted Jaccard is used as the underlying similarity function f. All algorithms are implemented in C++[18]. All experiments were run on a Linux server with 1.0GHz CPU and 256GB RAM.

5.2 Evaluating the Similarity Measure

In this experiment, we compare our pkduck similarity measure with traditional non-weighted Jaccard, JaccT [4] and SExpand [15] by evaluating the end-to-end accuracy of join results.

LCS-based dictionary with $\delta = 0$ is used. Table 2 shows some characteristics of this dictionary. Section 5.4 will include a comparison of different dictionaries.

We implement JaccT and SExpand using a modification of our signature generation algorithm to generate JaccT and SExpand signatures in polynomial time[19]. However, even with this optimization, JaccT still did not terminate in one week on any dataset because of the $O(2^{2^n})$ algorithm used to calculate the similarities for candidate string pairs, where n is can be in the thousands as shown in Table 2. Therefore, we conclude that it is not practical to use JaccT and do not include it in the comparison.

[18]Due to space limitation, interested readers could refer to our extended technical report [1] for a discussion on how to implement the ASJ in an RDBMS and an additional scalability experiment.

1	`martin luther king ave se`
	`mlk avenue se`

2	`historic columbia river hwy e`
	`hist col rvr hwy`

3	`broadway ste`
	`brdway suite`

Figure 8: Sample similar strings in Location.

Table 2: Characteristics of the LCS-based dictionary ($\delta = 0$). n is the number of applicable rules in a string. f is the sum of the frequency of abbreviations having more than 5 different full forms.

	Maximum n	Average n	f
Disease	51	2.60	0.02
Dept	188	11.90	0.04
Course	1,351	40.87	0.20
Location	1,716	38.87	0.26

Results are shown in Table 1. We observe the following:

- Our pkduck measure has the highest F-measure on all datasets, and greatly outperforms SExpand. For example, on Course, pkduck's highest F-measure is 0.71 ($\theta = 0.7$) while SExpand's highest F-measure is only 0.25 ($\theta = 0.9$).
- SExpand suffers from low precision. For example, on Location, SExpand's highest precision is only 0.02. This is because SExpand is susceptible to abbreviations having multiple full forms, which are frequent in input strings as shown in Table 2. In general, as analyzed in Example 1, any two strings both containing an abbreviation that has multiple full forms will get a high SExpand similarity. Note on Disease with

[19]See our extended technical report [1] for details on the modification.

$\theta = 0.9$, SExpand has high precision (0.99). This is because abbreviations having multiple full forms are not very frequent and the threshold is large.

- Jaccard has very low recall. The highest recall on all datasets is only 0.21. This is not surprising because matching strings often have very low Jaccard similarity as shown in Figures 1, 6, 7 and 8.

Limitations. Despite the clear benefits pkduck has, we note its two limitations. First, pkduck is often subject to the ineffectiveness of the underlying function f:

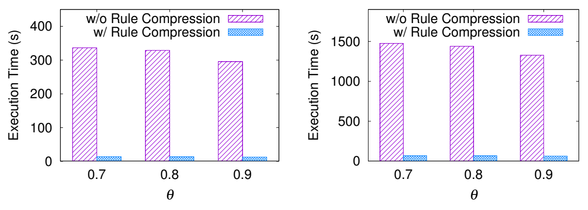
- pkduck has relatively low recall when θ is large. For example, on COURSE with $\theta = 0.9$, the recall of pkduck is only 0.39. This actually results from the use of Jaccard as the underlying similarity function. For two strings both containing very few tokens, a single dangling token will result in a very low Jaccard similarity. Consider two strings s_1 = `dept of cs` and s_2 = `department cs`. Even with the rule ⟨`dept` ⇔ `department`⟩, their pkduck similarity is only $\frac{2}{3}$ because of the dangling token `of`. On the contrary, SExpand is less sensitive to the underlying function because it can append different full forms of the same abbreviation to both strings to increase their similarity. For example, suppose `cs` has five different full forms as in Figure 2, the SExpand similarity between s_1 and s_2 is $\frac{12}{13}$ ($s'_1 \cap s'_2 = 12$ and $s'_1 \cup s'_2 = 13$). Unfortunately, as mentioned before, this comes at the cost of precision.
- The fact that some strings are considered similar by f even if they actually mean different things, may cause pkduck to incorrectly join some strings. One example of such false positives on COURSE is s_1 = `special sub in ee cs` and s_2 = `special sub in computer science`, with a learned rule ⟨`cs` ⇔ `computer science`⟩. Their pkduck similarity is $\frac{5}{6}$ because the Jaccard similarity between s'_1 = `special sub in ee computer science` and s_2 is $\frac{5}{6}$. However, they are two different courses.

Second, pkduck may join an abbreviation with an incorrect full form. For example, one false positive on DISEASE is s_1 = `ml` and s_2 = `metachromatic leukodystrophy` (s_1 refers to `mucolipidoses` not s_2). In this case, considering other attributes of a record will help (e.g. the symptoms of a disease). However, this is beyond the scope of this paper so we leave it for future work.

5.3 Evaluating the Join Algorithm

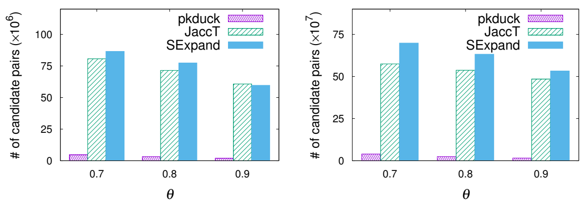
In this experiment, we evaluate the efficiency and scalability of our join algorithm and compare it with JaccT and SExpand. JaccT, SExpand and our join algorithm all follow the filter-verification framework which first generates prefix-filter signatures, then selects a set of candidate string pairs and finally verifies each candidate string pair. So we first compare our join algorithm with JaccT and SExpand in terms of signature generation efficiency, number of candidate string pairs and verification efficiency. Then, we evaluate how our join algorithm scales as the input size varies. The LCS-based dictionary with $\delta = 0$ is used.

Signature generation. Both JaccT and SExpand calculate the signature of a string by iterating over its derived strings, which is $O(2^n)$. Table 2 shows that in LCS-based



(a) COURSE (b) LOCATION

Figure 9: Performance of our PTIME signature generation algorithm. JaccT and SExpand did not finish generating signatures in one week.



(a) COURSE (b) LOCATION

Figure 10: Number of candidate string pairs, in comparison with JaccT and SExpand. The signatures of JaccT and SExpand are generated using a modification of our PTIME signature generation algorithm.

dictionaries n can be in the thousands, making this approach far from scalable. We implemented this brute-force way of generating signatures and it did not terminate in one week on any dataset.

In contrast, our PTIME signature generation algorithm is very efficient. The execution time is shown in Figure 9. Even without the rule compression optimization (Section 3.3), our signature generation algorithm took less than 1,500 seconds to finish on the largest dataset LOCATION.

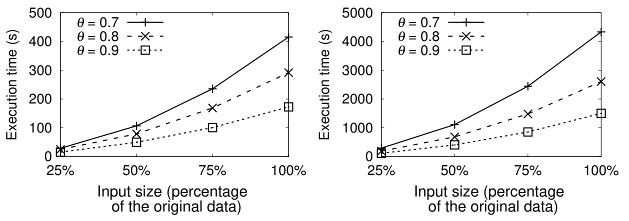
Figure 9 also shows that the rule compression technique significantly improves the efficiency. For example, when $\theta = 0.7$, rule compression makes the algorithm respectively 23× and 21× faster on COURSE and LOCATION. The reason is that the rule compression technique groups rules having the same number of tokens on the applicable side and the rewrite side, which lowers down the time complexity of our signature generation algorithm.

Number of candidate string pairs. Figure 10 shows the number of candidate string pairs selected by JaccT, SExpand and pkduck. It can be seen that pkduck selects much fewer candidate string pairs than JaccT and SExpand. For instance, on LOCATION with $\theta = 0.9$, pkduck selects 30× fewer candidate pairs than JaccT, and 33× fewer candidate pairs than SExpand. As analyzed in Section 3.2, this results from the much smaller search space of pkduck compared to those of JaccT and SExpand.

Verification efficiency. We report in Table 3 the average time of SExpand and pkduck to verify a candidate string pair ($\theta = 0.7$). JaccT is not included in the comparison because it did not terminate in one week on any dataset.

Table 3: Average verification time ($\theta = 0.7$) in comparison with SExpand.

	Course	Location
pkduck	8.12×10^{-5} s.	1.02×10^{-4} s.
SExpand	1.75×10^{-2} s.	1.69×10^{-1} s.



(a) Course (b) Location

Figure 11: Scalability of our join algorithm.

We can see that pkduck's verification algorithm is much faster than that of SExpand on Course and Location where n is large. For example, on Location, pkduck's average verification time is approximately three orders of magnitude shorter than that of SExpand. This is expected since the time complexity of pkduck's verification algorithm is $O(k \cdot n)$ whereas that of SExpand is $O(k \cdot n^2)$.

Scalability. We test how our entire join algorithm (including signature generation, candidate selection and candidate verification) scales as the input size varies on Course and Location. For each dataset, we generate three smaller datasets by randomly choosing a subset with respectively 25%, 50% and 75% of the original strings. The execution time on all small and original datasets is shown in Figure 11. As shown, our join algorithm scales well on both datasets.

5.4 Evaluating the Abbreviation Dictionary

Number of rules. Table 4 shows the number of LCS-based rules ($\delta = 0$ or 1) and number of rules after refinement, in comparison to the search space. We can see that the number of rules satisfying the LCS-based assumption is very small compared to the huge search space. For example, with $\delta = 0$ on the Course dataset, the search space is 1.9×10^{10} while the number of LCS-based candidate rules is only 3.3×10^5. Also, many LCS-based candidate rules are filtered out by the rule refiner. For example, among all 44,748 LCS-based candidate rules on Dept when $\delta = 1$, only 4,786 of them are preserved by the rule refiner.

Comparison of Dictionaries. We compare our LCS-based dictionaries with three types of dictionaries:

- ExampleS [5]: the state-of-the-art approach to learn synonym rules from examples of matching string pairs. We implement ExampleS using as examples all string pairs with Jaccard similarity not less than a threshold τ, as described in the paper.
- Handcrafted dictionaries: we manually construct abbreviation dictionaries for two small datasets Disease and Dept.
- A general-purpose dictionary: we collect from `abbreviations.com` 20,014 pairs of general-purpose abbreviations and their full forms.

Table 4: Number of LCS-based candidate rules (L) and number of rules after refinement (R), as compared to the search space (all pairs of a token and a token sequence).

	Search space	δ	L	R
Disease	3.5×10^6	0	4,302	1,362
		1	34,972	4,306
Dept	3.8×10^7	0	9,274	2,764
		1	44,748	4,786
Course	1.9×10^{10}	0	333,054	41,354
		1	2,068,916	67,292
Location	4.1×10^{11}	0	1,631,330	106,192
		1	17,374,792	286,424

To compare different dictionaries, we feed them to our join algorithm (using the pkduck similarity measure), and compare the accuracy of the join results with the user-inputted threshold $\theta = 0.7$. δ values 0 and 1 are used for LCS-based dictionaries. For those learned by ExampleS, we use τ values 0.4, 0.6 and 0.8.

Table 5 shows results. It is clear that LCS-based dictionaries greatly outperform dictionaries learned by ExampleS. For example, on Course, the highest F-measure of LCS-based dictionaries is 0.71 whereas that of ExampleS is only 0.14. We can also see ExampleS has very low recall. On Disease, the recall of ExampleS is 0. The maximum recall on all four datasets is only 0.27. The reasons are twofold. First, as can be seen in Figures 1, 6, 7 and 8, matching strings often have very low Jaccard similarity[20], so using traditional Jaccard-based ASJ misses many good examples. Second, ExampleS uses the frequency in the examples to infer how likely a rule is a real synonym rule (higher frequency indicates higher likelihood). Although high frequency is a good indicator (e.g. ⟨`dept` ⇔ `department`⟩ occurs many times in Dept), many real abbreviations occur very few times (e.g. ⟨`tat` ⇔ `training alignment team`⟩ only has one occurrence in Dept) and are thus missed by ExampleS.

Table 5 also shows that our LCS-based dictionaries have comparable accuracy as handcrafted dictionaries. Manual construction of the dictionary is an extremely cumbersome process which is prone to omissions of correct abbreviations, as indicated by the lower recall of handcrafted dictionaries. The low recall of the general-purpose dictionary indicates that abbreviations are highly domain-dependent and that our automatic approach is needed.

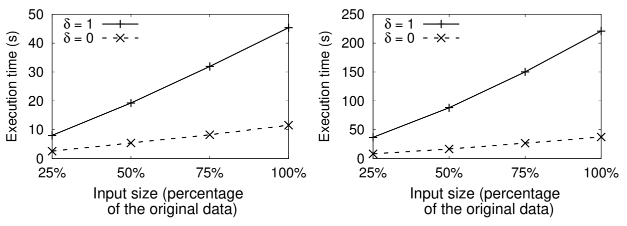
The comparison between two LCS-based dictionaries with $\delta = 0$ and $\delta = 1$ reveals that 0 is already a very good choice of δ in practice. The F-measure and precision of $\delta = 0$ is always no lower than that of $\delta = 1$. The recall of $\delta = 0$ is only slightly lower than that of $\delta = 1$ (by at most 2%). This shows that most real-world abbreviations can be captured by $\delta = 0$. Also, the LCS-based rule generation algorithm is easier to implement and runs faster with $\delta = 0$.

Scalability. We test how the LCS-based rule generation algorithm (Algorithm 1) scales as the input size varies on Course and Location.

[20]In fact, matching strings in Disease all have a Jaccard similarity less than 0.4, leading to ExampleS's 0 recall.

Table 5: Accuracy of join results ($\theta = 0.7$) when using LCS-based dictionaries, those learned by ExampleS [5], handcrafted dictionaries and a general-purpose abbreviation dictionary.

	Disease			Dept			Course			Location		
	P	R	F	P	R	F	P	R	F	P	R	F
ExampleS, $\tau = 0.4$	0.00	0.00	0.00	0.03	0.27	0.06	0.06	0.17	0.09	0.21	0.21	0.21
ExampleS, $\tau = 0.6$	0.00	0.00	0.00	0.14	0.23	0.17	0.09	0.17	0.12	0.27	0.21	0.23
ExampleS, $\tau = 0.8$	0.00	0.00	0.00	0.64	0.18	0.28	0.37	0.09	0.14	0.53	0.21	0.30
General-purpose	0.96	0.08	0.15	0.64	0.18	0.28	0.47	0.09	0.15	0.52	0.21	0.30
Handcrafted	0.92	0.69	0.79	0.80	0.74	**0.77**	—	—	—	—	—	—
LCS-based, $\delta = 0$	0.88	0.74	**0.81**	0.75	0.79	**0.77**	0.66	0.77	**0.71**	0.46	0.55	**0.50**
LCS-based, $\delta = 1$	0.88	0.74	0.81	0.70	0.80	0.75	0.61	0.78	0.68	0.25	0.57	0.35



(a) Course (b) Location

Figure 12: Scalability of the LCS-based rule generation algorithm.

Results are shown in Figure 12. We can see that Algorithm 1 has high efficiency and scales almost linearly as the input size grows. This high efficiency results from the use of the trie structure and the recursive calculation of the F array (Section 4.1.2).

6. RELATED WORK

There is a long literature on efficient computation of ASJs using traditional metrics, which can be divided into two categories: set-based (e.g. Jaccard [7,24] and Cosine [20,27]) and character-based (e.g. Edit [14,23,26] and Hamming Distance [6]). Many works [12,20,25,26] transform strings into sets of substrings with length q (i.e. q-grams), and then convert character-based metrics to set-based metrics. This idea can be adopted in our workflow when the underlying function f is one of those character-based metrics. Note, however, that traditional metrics are ineffective to deal with abbreviations, so these works cannot be adopted directly.

The filter-verification framework is widely adopted by aforementioned works for efficient computation. Under the framework, a filtering algorithm first filters out a large portion of dissimilar string pairs. A verification algorithm then verifies the remaining pairs by calculating their real similarities. Various signature-based filtering algorithms have been proposed, including prefix-filter [9,27], positional-filter [27], partition-based signature schemes [6,14] and approximate signature schemes [10]. Verifying set-based metrics is generally simple due to the straightforward calculation. Yet calculating character-based metrics such as edit distance is usually expensive. Algorithms have been designed to speed up the calculation of these metrics [14,26].

Our join algorithm for pkduck is also under this filter-verification framework. The filtering scheme extends prefix-filter [9] by calculating the signature union $Sig_u(s)$. Other types of signatures can also fit in this scheme, but it is generally hard to efficiently calculate $Sig_u(s)$ for two reasons. First, many signature algorithms [12,20,24,27] require building indexes on input strings to keep track of some positional information of tokens. However, to extend them to support pkduck, one has to build indexes for all derived strings, which is prohibitively expensive. Second, the signature space of many fast algorithms [6,14] is not polynomial, so it is not viable to modify our signature generation algorithm (which enumerates all possible signatures in the first step) to extend these algorithms.

Besides filter-verification algorithms, trie-join [23] is a fast algorithm using trie to directly compute join results. Both trie-join and our rule generation algorithm (Algorithm 1) employ the idea of identifying "valid trie nodes" for a set of query strings. However, because we focus on a different problem, Algorithm 1 differs inherently from trie-join in many key aspects. For example, the trie in trie-join is simply constructed by input strings. In contrast, the trie in Algorithm 1 is constructed by every subsequence of any token t that has at least $|t| - \delta$ characters. The definition of valid nodes and query strings are also different, due to the different problems being tackled.

Other highly related works [4,5,15] are discussed in detail in Section 1, so we do not repeat them in this section.

7. CONCLUSION

In this paper, we studied approximate string joins with abbreviations. We highlighted the limitations of existing approaches and proposed an end-to-end workflow to address them. We first designed a new similarity measure to quantify the similarity between two strings taking abbreviations into account. In contrast to existing measures, this measure is robust to abbreviations having multiple full forms. Note that this measure can also use other types of synonym rules. We then devised a PTIME join algorithm following the filter-verification framework, as opposed to existing algorithms whose time complexity is exponential. We also presented an unsupervised approach to learn a dictionary of abbreviation rules from input strings based on the LCS assumption. Experimental results on four real-world datasets showed that our approach is highly effective and efficient, and has clear advantages over state-of-the-art approaches.

8. ACKNOWLEDGMENT

This work was supported by the Qatar Computing Research Institute under Grant 6928356.

9. REFERENCES

[1] Extended technical report. web.mit.edu/wenbo/www/abbr.pdf.

[2] E. Adar. Sarad: A simple and robust abbreviation dictionary. *Bioinformatics*, 20(4):527–533, 2004.

[3] H. Ao and T. Takagi. ALICE: An algorithm to extract abbreviations from MEDLINE. *JAMIA*, 12(5):576–586, 2005.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[5] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[6] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[8] J. T. Chang, H. Schütze, and R. B. Altman. Creating an online dictionary of abbreviations from MEDLINE. *JAMIA*, 9(6):612–620, 2002.

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[10] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[11] R. De La Briandais. File searching using variable length keys. In *WJCC*, pages 295–298, 1959.

[12] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, et al. Approximate string joins in a database (almost) for free. In *VLDB*, volume 1, pages 491–500, 2001.

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

[14] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[15] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.

[16] M. Michelson and C. A. Knoblock. Mining the heterogeneous transformations between data sources to aid record linkage. In *ICAI*, pages 422–428, 2009.

[17] N. Okazaki and S. Ananiadou. A term recognition approach to acronym recognition. In *ACL*, 2006.

[18] N. Okazaki, M. Ishizuka, and J. Tsujii. A discriminative approach to Japanese abbreviation extraction. In *IJCNLP*, pages 889–894, 2008.

[19] Y. Park and R. J. Byrd. Hybrid text mining for finding abbreviations and their definitions. In *EMNLP*, pages 126–133, 2001.

[20] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.

[21] A. S. Schwartz and M. A. Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. In *PSB*, pages 451–462, 2003.

[22] X. Sun, N. Okazaki, J. Tsujii, and H. Wang. Learning abbreviations from Chinese and English terms by modeling non-local information. *ACM TALLIP*, 12(2):5:1–5:17, 2013.

[23] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1-2):1219–1230, 2010.

[24] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[25] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *TKDE*, 25(8):1916–1929, 2013.

[26] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[27] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, 2011.

[28] W. Zhang, Y. C. Sim, J. Su, and C. L. Tan. Entity linking with effective acronym expansion, instance selection, and topic modeling. In *IJCAI*, pages 1909–1914, 2011.