Dealing with Acronyms, Abbreviations, and Typos in Real-World Entity Matching

Joshua Wu
UC Berkeley
joshua.justin.wu@gmail.com

Dixin Tang
UT Austin†
dixin@utexas.edu

Nithin Chalapathi
UC Berkeley
nithinc@berkeley.edu

Tristan Chambers
UC Berkeley†
tristan.chambers@berkeley.edu

Julie Ciccolini
Techtivist†
julie@techtivist.com

Cheryl Phillips
Dept. of Communication,
Stanford University
cep3@stanford.edu

Lisa Pickoff-White
Investigative Reporting
Program, UC Berkeley†
pickoffwhite@berkeley.edu

Aditya Parameswaran
UC Berkeley
adityagp@berkeley.edu

ABSTRACT

String matching is at the core of data cleaning, record matching, and information retrieval. String matching relies on a similarity measure that evaluates the similarity of two strings, regarding the two as a match if their similarity is larger than a user-defined threshold. In our collaboration with journalists and public defenders, we found that real-world datasets, such as police rosters that journalists and public defenders work with, often contain acronyms, abbreviations, and typos, thanks to errors during manual entry, into, say, a spreadsheet or a form. Unfortunately, traditional similarity measures lead to low accuracy since they do not consider all three aspects together. Some recent work proposes leveraging synonym rules to improve matching, but either requires these rules to be provided upfront, or generated prior to matching, which leads to low accuracy in our setting and similar ones. To address these limitations, we propose Smash, a simple yet effective measure to assess the similarity of two strings with acronyms, abbreviations, and typos, all without relying on synonym rules. We design a dynamic programming algorithm to efficiently compute this measure, along with two optimizations that improve accuracy. We show that compared to the best baselines, including one based on ChatGPT with GPT-4, Smash improves the max and mean F-score by 23.5% and 110.8%, respectively. We implement Smash in OpenRefine, a graphical data cleaning tool, to facilitate its use by journalists, public defenders, and other non-programmers for data cleaning.

PVLDB Reference Format:
Joshua Wu, Dixin Tang, Nithin Chalapathi, Tristan Chambers, Julie Ciccolini, Cheryl Phillips, Lisa Pickoff-White, and Aditya Parameswaran. Dealing with Acronyms, Abbreviations, and Typos in Real-World Entity Matching. PVLDB, 17(12): 4104 - 4116, 2024.
doi:10.14778/3685800.3685830

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dx-tang/smash.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685830

1 INTRODUCTION

String matching is a process of identifying and matching similar strings underlying a variety of applications, such as data cleaning and integration, record linkage, and information retrieval [9, 14, 16, 29, 38]. For example, for data cleaning, string matching can match strings from different data sources that refer to the same entity, improving data quality and reducing errors. String matching typically evaluates the similarity of two strings using a similarity function of some sort, e.g., Jaccard similarity [36], or distance metric, e.g., Levenshtein distance [4]. Two strings are deemed to be a *match* if their similarity score is higher than (or, equivalently, their distance metric is smaller than) a given user-specified threshold.

While string matching has a rich history, *one unaddressed challenge is that real-world datasets include strings with various forms of* **acronyms** *and* **abbreviations** *to represent the same entities, as well as* **typos** *due to human mistakes in data entry.*

Application 1 (Police Roster Cleaning). *As part of a consortium titled CLEAN (Community Law Enforcement Accountability Network), we work with journalists and public defenders to clean, organize, and analyze police data across various states in the US, including* rosters*, information about police officers, typically organized as a CSV or spreadsheet. In one instance, we worked with public defenders from the National Association of Criminal Defense Lawyers (NACDL) to clean a dataset from a midwestern public defender office that includes police officer titles as a column (referred to as the Police Roster dataset henceforth). This dataset includes a number of acronyms (e.g., "school resource officer" as "sro"), abbreviations (e.g., "deputy marshall" as "dpty mrsl"), and typos (e.g., "sergeant" as "sargeant"), as well as combinations thereof, because these officer titles are manually entered by police department personnel. The public defender we worked with, who doesn't know programming, required two weeks to "clean" the police titles from over 700 to less than 100, and still wasn't entirely sure if the task was finished. Intuitively, this process involved manually comparing all pairs of over 700 titles, explaining why it took them such a long time. They, and other public defenders and data journalists, commonly use GUI-based data cleaning tools, such as OpenRefine [5], but none of the built-in similarity metrics in such tools sufficed for their purposes.*

†This work was done when Dixin Tang was affiliated with UC Berkeley, Tristan Chambers and Julie Ciccolini were working on the Full Disclosure Project at the National Association of Criminal Defense Lawyers (NACDL) and Lisa Pickoff-White was affiliated with KQED and the California Reporting Project.

Unfortunately, existing string similarity metrics, such as Levenshtein distance [4], affine gap distance [11], and Jaccard similarity [17, 36], fail to effectively perform string matching for such datasets with acronyms, abbreviations, and typos—we discuss traditional string similarity metrics in Section 2 in detail. Some other approaches [10, 25] tokenize the two input strings into two sets of words, build a bipartite graph between the two sets, with the similarity of two words computed using an existing metric (e.g., edit distance), and combine these similarities via bipartite matching (referred to as *Bipartite* henceforth). Since bipartite matching operates at the granularity of words, it cannot be directly applied to the scenarios where one string includes acronyms (e.g., "sro" for "school resource officer") or is a single abbreviated word (e.g., "apmngr" for "assistant park manager").

A recent line of research improves on the limitations of traditional measures by using domain-specific synonym rules for rewriting a short string to a long string (e.g., "sro" → "school resource officer") [9, 18, 24, 26, 28, 30, 33, 39]. To evaluate the similarity of two strings, synonym rules are used to rewrite the two strings followed by using (a variant of) traditional measure to compute similarity. Unfortunately, some of these papers rely on predefined synonym rules [9, 18, 24, 26, 33, 39], which may not exist for many datasets (such as ours). In addition, predefined synonym rules limit the scope of abbreviations they support. That is, they only support predefined ones (e.g., "deputy" → "dpty") but not arbitrary ones (e.g., "deputy" → "dpt"). Other papers instead propose automatically generating synonym rules [28, 30], but their performance is highly sensitive to the quality of the generated rules and they cannot consistently generate high-quality rules for different datasets; additionally, typos (as in our context) lead to issues in generating high-quality rules. Table 1 summarizes the limitations of existing similarity measures.

To address the limitations of existing approaches, we propose Smash[1], a simple yet effective metric that considers typos, acronyms, and abbreviations together, while not relying on synonym rules. The key idea of Smash is that for every word in the long string, some representation of it—the full word possibly with typos, its abbreviation, or the first letter—should appear as a substring in the short string in order, as visualized in Figure 1. One example is that the three letters of "sro" appear as the first letters of the three words of "school resource officer" in order, respectively, representing the acronym case. Therefore, we partition the short string into m substrings, where m equals the number of tokenized words in the long string. The Smash measure is defined as *the minimal sum of the distances between each word in the long string and its corresponding substring in the short string*. The distance between a word and a substring can be computed based on traditional measures, such as the affine gap [11] and subsequence [7], allowing the approach to flexibly adapt to user needs.

Unfortunately, efficiently computing Smash is challenging. Given the short string with n characters that is partitioned into m substrings, there is a high degree polynomial number of possible partitions (i.e., $\binom{n-1}{m-1}$ because we choose $m-1$ positions from the string minus the first character). Therefore, we develop a simple novel dynamic programming algorithm to compute Smash efficiently and include optimizations to further improve the accuracy of Smash (e.g., selectively skipping stop words).

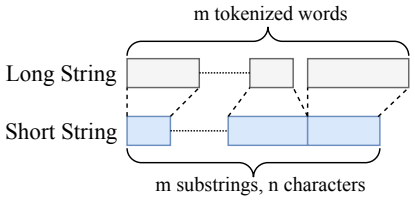

Figure 1: The intuition for capturing acronyms, abbreviations, and typos between two strings

We perform extensive experiments on real-world datasets by comparing Smash with all representative baselines that do not require pre-defined synonym rules, including Levenshtein, affine gap, two Jaccard similarity variants, Bipartite, and pkduck. Their features and differences from Smash are summarized in Table 1. Our results show that Smash significantly outperforms existing approaches. Specifically, compared to the best baselines, Smash improves the max and mean F-score by 23.5% and 110.8%, respectively. We additionally compare Smash with a state-of-the-art LLM-based approach, ChatGPT with GPT-4.0, and find that LLMs are unable to achieve the high recall necessary for string matching, nor are they able to adapt to unfamiliar datasets.

To help public defenders, data journalists, and other non-programmers with similar data cleaning requirements use Smash, we implement Smash in OpenRefine [5], an open-source data cleaning tool, where end-users can choose Smash as the similarity measure when matching strings across one or multiple datasets.

The contributions of this paper are summarized as follows:

- A novel yet simple similarity measure, Smash, that considers acronyms, abbreviations, and typos at the same time while not relying on brittle synonym rules (Section 3);
- A dynamic programming algorithm that efficiently computes Smash and two optimizations that improve the accuracy of Smash, while also being parametrizable with respect to various distance functions internally (Sections 4-5);
- An implementation in a popular GUI-based data cleaning tool, OpenRefine, to empower non-programmers to more efficiently perform string matching (Section 6); and
- A set of extensive experiments that compare Smash with six baseline approaches on four datasets that demonstrate the value of Smash and the corresponding dynamic programming approach in various real-world settings, including the one described in Application 1 on police roster data cleaning (Section 7).

2 RELATED WORK

We now discuss work related to Smash, including similarity measures for string matching, ground truth-aided string matching, and methods for reducing the execution time of string matching.

Similarity measures for string matching. Table 1 summarizes the limitations of existing similarity measures for string matching. Traditional similarity measures, such as Levenshtein distance [4] and affine gap distance [11], only consider scenarios with typos

[1]With apologies to the Hulk, Smash is named as such because it is able to "smash" together strings, taking into account typos, acronyms, and abbreviations.

Table 1: Summary of the Differences of Existing String Similarity Metrics and Smash

	Supported Features					Require Pre-Defined Synonym Rules	Adopt Synonym Rules Online
	Acronyms	Pre-Defined Abbreviations	Arbitrary Abbreviations	Typos	Mixed		
Levenshtein [4]	No	No	No	Yes	No	No	No
Affine Gap [11]	Yes	No	No	Yes	No	No	No
Jaccard-Word [36]	No	No	No	No	No	No	No
Jaccard-NG [17]	No	Yes	Yes	Yes	No	No	No
Bipartite [10, 25]	No	Yes	Yes	Yes	No	No	No
Smash (this paper)	Yes	Yes	Yes	Yes	Yes	No	No
pkduck [30], Match-DP [28]	Yes	Yes	Yes	Yes	Yes	No	Yes
Others [9, 18, 24, 26, 33, 39]	Yes	Yes	No	Yes	Yes	Yes	No

or acronyms. Levenshtein distance, also known as edit distance, measures the similarity of two strings by counting the minimal number of insertions, deletions, or substitutions required to edit one string to match the other. While this measure can identify typos, it does not take into account acronyms or abbreviations. On the other hand, affine gap [11] modifies edit distance by assigning a smaller penalty to a continuous sequence of insertions or deletions compared to the initial insertion or deletion. This property makes it better suited for capturing acronyms since the characters that follow the first letters of each word in the longer string are treated as "gaps" in the shortened string and are penalized at a discount. But affine gap does not address the case of abbreviations.

Another line of papers focuses on set similarity search, which focuses on tokenizing the two strings into two sets of words and then computing their similarity [10, 25, 26, 36]. For example, Jaccard similarity score [36] calculates the ratio between the number of common words and the total number of distinct words of the two strings (denoted as *Jaccard-Word*). Bipartite [10, 25] approaches, as discussed earlier, build a bipartite graph between the two sets to compute their similarity. But these approaches compute similarity at the granularity of words and do not consider character-level features. One method for addressing the limitation of set similarity search is creating n-grams for the two input strings and computing the Jaccard score over the two sets of n-grams [17] (denoted as *Jaccard-NG*). While Jaccard-NG considers abbreviations and typos, it does not consider acronyms. In addition, none of these methods consider the mixed cases of acronyms, abbreviations, and typos together.

There are many papers on leveraging synonym rules to improve traditional measures [9, 18, 24, 26, 28, 30, 33, 39]. As discussed before, most of these papers assume that the synonym rules are known upfront or provided by the user [9, 18, 24, 26, 33, 39], but these rules often do not exist, in domains such as the police data setting. Smash is different from these approaches since it does not rely on synonyms. Some other papers propose automatically generating synonym rules [28, 30], but our experiments in Section 7 show that the performance of these approaches is sensitive to the quality of the generated rules and they cannot consistently generate high-quality synonym rules for different datasets.

Consider pkduck [30], the state-of-the-art in this vein, as an example. pkduck generates candidate synonym rules based on the longest common sequence of each pair of strings, which produces many incorrect rules. Therefore, pkduck adopts manually developed refinement rules to discard any synonym rules that may be

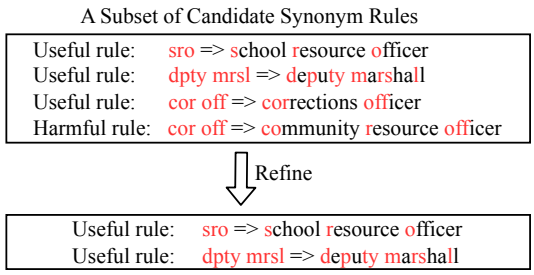

Figure 2: A subset of synonym rules generated by pkduck [30] for the Police Roster dataset

potentially harmful. Figure 2 shows a subset of synonym rules before and after the refinement for the Police Roster dataset. Initially, we have four rules. However, the rule ("cor off" → "community resource officer") is harmful because the ground truth shows that "cor off" should only map to "corrections officer". To improve the quality of the synonym rules, one refinement rule used by pkduck involves discarding the rewriting rules if the ratio between the number of consonants of the short and the long strings is smaller than a predefined threshold (0.6 by default) based on the assumption that an abbreviated short string should include a large fraction of consonants from the long string. This refinement rule, while discarding many harmful rules, will also discard useful rules, such as discarding "cor off" → "corrections officer" because the consonant ratio is $\frac{4}{11} = 0.37$ and smaller than the threshold 0.6. In fact, our experiments show that the refinement process can *degrade the performance of pkduck for some datasets.*

Ground truth-aided string matching. Some papers propose asking end-users to provide examples and automatically learn from these examples for more accurate string matching [12, 27]. A few papers adopt machine learning models for string matching [8, 14, 23]. One paper proposes a novel machine-learning model for matching the strings of healthcare data, trained on predefined ground truth [14]. Another paper fine-tunes pre-trained language models for string matching [23]. Finally, the paper [8] leverages existing similarity metrics to clean input data in order to improve the quality of transformer models. Smash is different from these papers because it does not require ground truth knowledge.

Reducing the execution time of string matching. There has been a number of papers on reducing the execution time of string matching [10, 13, 15, 19, 21, 22, 31, 32, 34, 35, 37, 38]. To avoid computing similarity for all pairs of strings, many papers exploit a

"filter-and-refine" framework [15, 21, 22, 31, 32, 37, 38]. In the "filter" step, they generate signatures for each string and use the signatures to generate candidate pairs of strings to evaluate. In the "refine" step, they compute the similarity for the candidate pairs to generate the final results. Some other papers implement string matching as a primitive operator in databases and optimize the performance of evaluating this operator [10, 13]. These papers are orthogonal to our goal of designing a novel metric to capture acronyms, abbreviations, and typos together, and our Smash distance can be used together with their approaches to improve the accuracy of string matching.

3 OUR STRING SIMILARITY MEASURE

In this section, we introduce the intuition and examples that motivate the design of Smash, and the formal definition of Smash.

Preliminaries. We define a string to be a sequence of characters. A *subsequence* of a string is a string that can be derived from the given one by deleting zero or more characters without changing the order of the remaining characters. We say a string is *tokenized* into an array of words if this string is divided into its component words in order based on predefined delimiters (e.g., a space character). The string matching process takes two strings, computes their similarity, and determines them to be a *match* if the similarity score is larger than a threshold. We call the string whose length is no smaller than the other string the *long string*, and the other string the *short string*. If two strings have the same lengths, we arbitrarily choose one string as the long one and the other string as the short one.

Intuition. Our key idea for capturing acronyms, abbreviations, and typos is that for the long and the short string representing the same entity, some form of representation of each tokenized word for the long string will appear as a substring of the short string in order, because we expect the long string as the canonical expanded representation for an entity while the short string is the potentially human-inputted, and therefore, error-prone version. We consider each scenario in turn. First, for the pure acronym scenario (e.g., "school resource officer" vs. "sro" in Figure 3), the first character of each word of the long string corresponds to each character in the short string in order. For the abbreviation scenario, each word of the long string will be a subsequence of the corresponding substring of the short string or vice versa, with the first character of the word and the substring being the same, which is based on our observation that the first character of a word commonly appears as the first character of its abbreviated version. One example is the Abbreviation-1 case in Figure 3, where the substrings "dpty " and "mrsl" of the short string are subsequences of the words "deputy" and "marshall", respectively. Another example is the Abbreviation-2 case in Figure 3, where the word "dpty" in the long string is a subsequence of "deputy " and the substring "mrsl" is a subsequence of "marshall" in the long string. For the typo scenario, a tokenized word in the long string can be modified due to typos and correspond to a substring in the short string, where the distance between the word and the corresponding substring can be measured using a traditional metric (e.g., Levenshtein distance). Finally, this intuition also covers the case when two strings have abbreviations, acronyms, and typos at the same time. For example, the Mixed-1 case in Figure 3 shows that the short string (i.e., "ims") is an abbreviation of the long string (i.e., "inspector") with a typo ("m" → "n"). The Mixed-2 case covers both

	Long String	Short String
Acronym	**s**chool **r**esouce **o**fficer	sro
Abbreviation-1	**deputy marshall**	dpty mrsl
Abbreviation-2	dpty **marshall**	**deputy** mrsl
Typo	inspector	imspector
Mixed-1	**in**spector	ims
Mixed-2	**a**ssistant **p**ark **manage**r	apmngr

Figure 3: Examples that motivate Smash

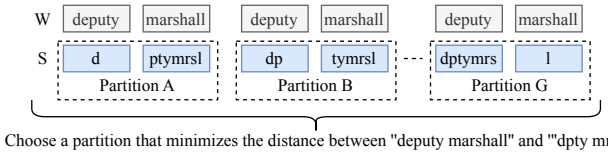

Figure 4: The example for computing $d_s(W, S)$

the acronym and abbreviation scenarios, where the short string "smashes" the long string into a single word. Smash is designed to cover all of the aforementioned cases.

Smash Definition. Motivated by this intuition, the distance between two strings is conceptually defined as *the minimal sum of distances between each word in the long string and its corresponding substring in the short string*. We now formalize the problem and formally define Smash.

Given two strings for which we need to compute the distance, we first tokenize the long string into an array of words based on predefined delimiters. We denote the array of words W. Consider the example Abbreviation-1 in Figure 3. For "deputy marshall", if the delimiter is the space character, then W is ["deputy", "marshall"]. In addition, we represent the short string as an array of characters, denoted S. Note that we remove the delimiters from the short string as a preprocessing step. For example, "dpty mrsl" is represented as an array of eight characters S = ["d", "p", "t", "y", "m", "r", "s", "l"], where spaces are removed. The length of W and S are m and n, respectively. We define a partition of the character array of the short string S as $P = \{[i_0, i_1], [i_1, i_2], \ldots, [i_{m-1}, i_m]\}$, where the number of partitions is the number of words m such that each partition of the character array $S[i_k : i_{k+1}]$ corresponds to the word $W[k]$. Note i_0 is 0, the first character of S, and i_m is n, the length of S, such that both the first and last partitions are non-empty. Continuing our example, if the partition of ["d", "p", "t", "y", "m", "r", "s", "l"] is $P = \{[0, 4], [4, 8]\}$, the substring $S[0, 4]$ is "dpty", which corresponds to $W[0]$ = "deputy" and the substring $S[4, 8]$ is "mrsl", which corresponds to $W[1]$ = "marshall". We use d_s to represent the Smash distance between W and S, defined as:

$$d_s(W, S) = \min_{P=\{[i_0,i_1],\ldots,[i_{m-1},i_m]\}} \sum_{k=0}^{m-1} d_w(W[k], S[i_k : i_{k+1}]) \quad (1)$$

Here, $d_w(W[k], S[i_k : i_{k+1}])$ computes the distance between a word and a substring (e.g., "dpty" v.s. "deputy" in the Abbreviation-1 example). So, computing d_s is equivalent to finding the partition P of the string S such that the sum of $d_w(W[k], S[i_k : i_{k+1}])$ is minimized, which is visualized in Figure 4 using the Abbreviation-1 example. We see that there are seven different ways to partition the character array of "dptymsrl", so to compute d_s, we will find the partition that minimizes the sum of $d_w(W[k], S[i_k : i_{k+1}])$.

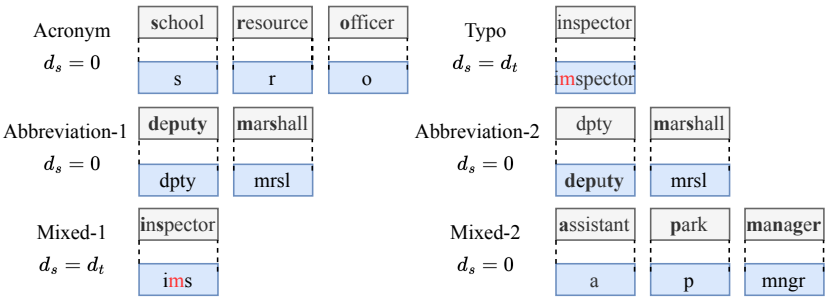

Figure 5: Applications of Smash to the motivating examples

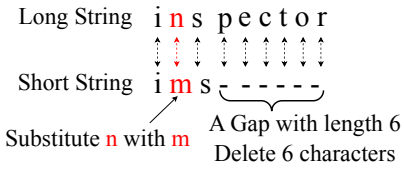

Figure 6: An example that shows the affine gap distance

d_w is computed using traditional string distance measures and is defined as:

$$d_w(W[k], S[i_k : i_{k+1}]) = \begin{cases} 0, & \text{if } W[k][0] = S[i_k] \\ & \text{and } W[k] \text{ is a subsequence} \\ & \text{of } S[i_k : i_{k+1}] \text{ or vice versa} \\ \infty, & \text{if } W[k][0] \neq S[i_k] \\ d_t(W[k], S[i_k : i_{k+1}]), & \text{otherwise} \end{cases} \quad (2)$$

We consider three cases. First, if the first characters of the word and the substring are the same and the word is a subsequence of the substring or vice versa, then we return 0, representing the case that the word and the substring are in the abbreviation or the acronym scenario.

The next two cases consider the typo scenario. If the typo happens in the first character we return ∞, representing that they do not match. Our observation on real datasets is that a typo is unlikely to happen in the first character and a mismatch in the first character typically represents a mismatch between the word and the substring.

Finally, we use function $d_t(W[k], S[i_k : i_{k+1}])$ to compute a distance in the presence of typos. By default, we use the affine gap distance [11], but we can swap in other distance metrics. This distance metric extends Levenshtein distance by assigning different weights to different operations (e.g., substitution vs. deletion) and for consecutive insertions or deletions (called a gap), assigning a smaller weight to insertions or deletions that are after the initial one. We choose affine gap over Levenshtein because it more accurately measures the case when an abbreviation has typos. Consider the Mixed-1 example in Figure 3, where the abbreviation of "inspector" is "ins" but misspelled as "ims". To measure the distance between the two strings, affine gap considers one substitution and six consecutive deletions as Levenshtein does, as shown in Figure 6. However, affine gap assigns a smaller weight to deleting the gap "pector" compared to Levenshtein, allowing it to more accurately capture an abbreviation with typos.

Applications of Smash to the Motivating Examples. We now show how Smash captures acronym, abbreviation, and typos in real examples from Figure 3. Figure 5 shows the optimal partition for each example. We see that for the Acronym example, the short string "sro" is partitioned to substrings "s", "r", and "o", which correspond to the words "school", "resource", "officer", respectively. Since the first characters of each pair of substring and word are the same and the substring is a subsequence of the corresponding word, the distance for each pair of substring and word is 0 based on the definition of d_w in Equation 2. Therefore, the Smash metric d_s is also 0. Similarly, for the Abbreviation-1, Abbreviation-2, and Mixed-2 examples, their distances are also 0. One thing to note is that the distance for Abbreviation-2 is 0 because for each pair of substring and word we check the subsequence condition in both directions as shown in Equation 2. Finally, for the Typo and Mixed-1 examples, we compute their distance using d_t, which adopts affine gap to compute their distances.

4 COMPUTING SMASH

In this section, we discuss how to efficiently compute the Smash distance between two strings. We demonstrate optimal substructure for computing Smash, present a dynamic programming algorithm based on the optimal substructure and an algorithm that constructs the string matches for Smash, and analyze their complexity.

4.1 Optimal Substructure

Here, we show optimal substructure, which implies that the optimal solution of a problem can be constructed from the optimal solutions of its subproblems. Recall that the Smash distance $d_s(W, S)$ determines the partition P of the string S such that the sum of the distance between each word and each substring is minimized. The intuition for optimal substructure of Smash is that *the minimal distance between the word array* $W[0 : m]$ *and the string* $S[0 : n]$ *should be the minimal distance between a smaller word array* $W[0 : m-1]$ *and a substring* $S[0 : p]$*, where* $p \leq n$ (i.e., the subproblem $d_s(W[0 : m-1], S[0 : p])$), *plus the distance between the last word* $W[m-1]$ *and the remaining substring* $S[p : n]$ (i.e., computed by $d_w(W[m-1], S[p : n])$ in Equation 2). Therefore, we enumerate the possible values of p to find the minimal sum of the two distances. Figure 7 shows an example for computing the minimal distance between ["assistant", "park", "manager"] and the string "apmngr" given that the subproblems are solved. Specifically, we enumerate $p \in [2, 6)$, where p splits the string into two substrings. The first substring corresponds the word array minus the last word (e.g., "ap" for ["assistant", "park"] when $p = 2$) and the second substring corresponds to the last word (e.g., "mngr" for ["manager"] when $p = 2$). We compute the minimal distance for

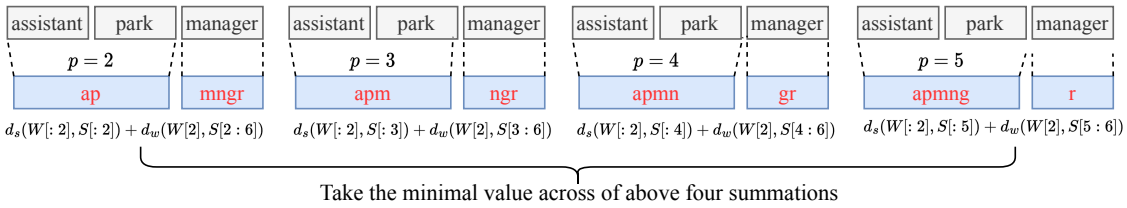

Figure 7: The example for demonstrating the optimal substructure

Algorithm 1: Computing the SMASH distance between a word array W and a string S

```
/* We use D[i][j] to store the minimal distance between
   W[0 : i+1] and S[0 : j+1] and E to construct the partition
   of S that yields the minimal distance                    */
D ← a m × n 2D array with initial values as ∞
E ← a m × n 2D array
/* The base case                                            */
for i = 0 to n − 1 do
    D[0][i] ← d_w(W[0], S[: i+1])
    E[0][i] ← 0
end
for i = 1 to m − 1 do
    for j = 1 to n − 1 do
        if j < i then
            continue // Skip if the length of the string is
              smaller than the number of words
        D[i][j] ← min_{p∈[i,j+1)} D[i−1][p] + d_w(W[i], S[p : j+1])
        E[i][j] ← argmin_{p∈[i,j+1)} D[i−1][p] + d_w(W[i], S[p : j+1])
    end
end
return D[m−1][n−1], E
```

each pair of substring and word array and return the minimal summation of the two distances. Note that index p starts at 2 (i.e., $m-1$) because we need to ensure the word array $W[0:m-1]$ has at least $m-1$ characters to match. Formally, we have:

$$d_s(W,S) = \begin{cases} d_w(W[0],S), & \text{if } m = 1 \\ \min\limits_{p\in[m-1,n)} (d_s(W[0:m-1],S[0:p]) + \\ \qquad d_w(W[m-1],S[p:n])), & \text{otherwise} \end{cases} \quad (3)$$

First, if we only have one word, we directly compute the distance between the word and the full string, i.e., $d_w(W[0],S)$. Otherwise, we compute $d_s(W,S)$ from its subproblems $d_s(W[0:m-1],S[0:p])$, where p varies from $m-1$ until n.

4.2 Dynamic Programming Algorithm

Intuition. Based on Equation 3, we develop a dynamic programming algorithm. We discuss the iterative version of this algorithm to avoid recursion. The high-level idea is to compute a 2D array D from bottom up, where $D[i][j]$ stores the minimal distance between the word array $W[0:i+1]$ and a string $S[0:j+1]$, such that $D[m-1][n-1]$ is the minimal distance between W and S. We store the auxiliary information in a 2D array E to construct the optimal partition of S that yields the minimal distance. Specifically, for the case where $W[0:i+1]$ and $S[0:j+1]$ have the optimal string matching with respect to the SMASH distance, $E[i][j]$ stores the position of the first letter of the last substring of $S[0:j+1]$ that matches the last word of $W[0:i+1]$, i.e., $W[i]$. For example, consider the case where $W[0:3]$=["assistant", "park", "manager"] and $S[0:6]$ ="apmngr" have the optimal string matching when the last word "manager" matches the substring "mngr". So $E[2][5]$ stores 2, which is the index of "m" in "apmngr" such that it can be used to locate the substring "mngr", which matches the last word "manager".

Algorithm. The algorithm is listed in Algorithm 1. We first initialize the 2D arrays, D and E. Then, we set the values of the first row of D to the distances between the first word in W and each prefix of S. The first row of E is set to all zeros since the first letter of the first word's matched substring is always the first letter of S.

Next, we fill in the remaining cells of D and E. For each $i \in [1, m-1]$ and $j \in [0, n-1]$, the algorithm computes the minimum distance between $W[0:i+1]$ and $S[0:j+1]$ by considering all possible partitions of $S[0:j+1]$ at the position $p \in [i, j+1)$ based on Equation 3. The value of p that yields the minimal distance is stored in $E[i][j]$. Figure 8 shows a running example for computing SMASH between ["assistant", "park", "manager"] and "apmngr". Specifically, it shows the steps of computing the rows $D[2]$ and $E[2]$ given their previous two rows are computed. For example, to compute $D[2][5]$, it enumerates the partition of "apmngr" and finds the one that yields the minimal distance based on Equation 3. Specifically, it finds that splitting "apmngr" into "ap" and "mngr" yields the minimal distance. Here, "ap" corresponds to "assistant park", their minimal distance is $D[1][1] = 0$ and the distance between "mngr" and "manager" is 0 based on the Equation 2. So $D[2][5] = 0$ and $E[2][5] = 2$, which is the index for the first letter of "mngr" in "apmngr".

Finally, the algorithm returns $D[m-1][n-1]$, which represents the minimal distance between the entire W and S, as well as E, which is used to construct the optimal partition of S.

Complexity. We use two **for** loops to fill in the cells of D and E. Computing the value for each cell of D and E requires executing the function d_w for $O(n)$ times. In total, we need to execute d_w for $O(m \times n^2)$ times. Recall that d_w computes the distance between a word and a substring and is defined in Equation 2. Its complexity is dominated by d_t, which computes the distance that accounts for typos. Assuming we choose affine gap, its complexity is $O(l_1 \times l_2)$,

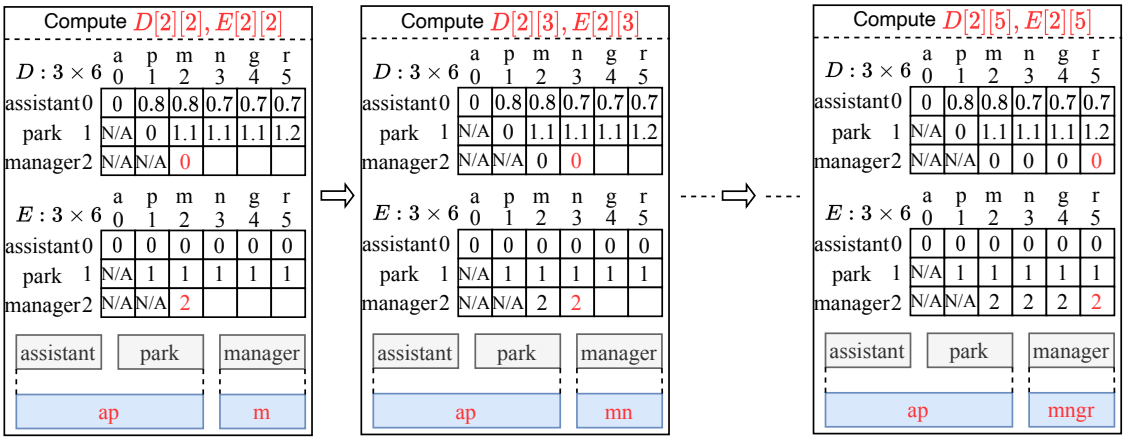

Figure 8: A running example that shows how our DP algorithm works

where l_1 and l_2 represent the number of characters of the word and the substring, respectively. We further assume the max number of characters of a word in the word array W is l, so the complexity for d_w is $O(l \times n)$. Finally, the complexity of Algorithm 1 is $O(m \times n^3 \times l)$. We believe this is an acceptable complexity for two reasons. First, as our experiments in Section 7.3 show, Smash has a similar execution time compared to pkduck, is slower than Levenshtein, and is faster than all other baselines while Smash performs string matching more accurately compared to all baselines in most cases. Second, Smash can be easily used together with blocking techniques to further reduce the execution time if needed.

Algorithm 2: Use E to compute the optimal partition for S that minimizes the distance between W and S

```
Initialize L as a list that stores n
p ← n
for i = m − 1 to 0 do
    p ← E[i][p − 1]
    Prepend p to L
end
return {[p[0], p[1]], ··· , [p[m − 1], p[m]]}
```

4.3 Constructing the Optimal Partition

Intuition. Recall that for the case where $W[0:i+1]$ and $S[0:j+1]$ have the minimal distance with respect to Smash, $E[i][j]$ stores the position of the first letter of the last substring of $S[0:j+1]$ that matches the last word of $W[0:i+1]$, i.e., $W[i]$. Therefore, we find the optimal partition for $S[0:n]$ by i) storing the position of the first letter of the substring that matches the last word of $W[0:m]$ and ii) repeatedly storing this position for the optimal solution for its subproblem, which is the minimal distance between the word array that removes the last word (i.e., $W[0:m-1]$) and the string that removes the last substring that matches $W[m-1]$ (i.e., $S[0:E[m-1][n-1]]$). For our running example in Figure 8, $E[2][5]$ stores 2, which points to "m" in "apmngr" and represents that "mngr" corresponds to the last word "manager". Recursively, for the subproblem "ap" and ["assistant", "park"], we take the value stored in $E[1][1]$, which is 1 and splits "ap" into "a" and "p". Therefore, the optimal partition is ["a", "p", "mngr"].

Algorithm. Algorithm 2 shows how we construct the optimal partition. We use a list L to store the positions for the optimal partition. Initially, L only includes n, which is the length of the short string S and also the end position for the optimal partition. Then, we repeatedly store the position of the first character of the substring of $S[0:p]$ that matches the last word of the word array $W[0:i+1]$ (i.e., $E[i][p-1]$). Finally, we use L to construct the optimal partition.

Complexity. We use one **for** loop to find the positions for the optimal partition of S, which takes $O(m)$ operations.

5 OPTIMIZATIONS AND APPLICABILITY

Optimizations. Our previous discussion assumes that some form of representation of one word of the long string should appear in the short string. But in a real dataset, this may not always be the case for two reasons. First, some words in the long string have little or no semantic value to distinguish one string from another and are often referred to as *stop words* (e.g., "at", "is", and "n"). Therefore, such words may be skipped in the short string. Second, we observe that when creating a short form of the long string, people tend to drop short words. For example, one short form of the long string "Motor carrier inspector 3" is "mci", where "3" is not included in the short string.

However, we cannot simply preprocess a dataset by removing these stop words or short words (e.g., a word that has 3 characters or less) because these words may still have semantic values for string matching. For example, the letter "n" in "state hwy n" is useful when matching the string "state highway north".

Therefore, we improve our dynamic programming algorithm by considering whether to skip a given word if this word is in the stop word list or is a short word and taking the minimal distance between the two cases. Specifically, Equation 3, which defines optimal

substructure, is modified as:

$$d_s(W,S) = \begin{cases} d_w(W[0],S), & \text{if } m = 1 \\ d_{new}(W,S), & \text{otherwise} \end{cases} \quad (4)$$

where $d_{new}(W,S)$ is defined as:

$$d_{new}(W,S) = \min \begin{cases} d_w(W[0:m-1],S) \\ \min\limits_{p \in [m-1,n)} (d_s(W[0:m-1],S[0:p]) + \\ \qquad d_w(W[m-1],S[p:n])) \end{cases} \quad (5)$$

Here, d_{new} considers whether skipping the last word (i.e., $W[m-1]$) and takes the minimal distance between the two cases. Therefore, Line 11 in Algorithm 1 is modified accordingly to consider the two cases when computing $D[i][j]$. Finally, we also modify Line 12 in Algorithm 1 for computing $E[i][j]$: if we choose to skip the word $W[i]$, $E[i][j]$ is set to $E[i-1][j]$. Note that adding these optimizations does not add additional complexity to our algorithm. This is because we only need to compute each cell of D and the complexity for computing $D[i][j]$ does not change given that the optimizations only add an operation that takes the minimum value of two cases, which takes $O(1)$.

Applicability. Smash is applied to the cases where a string representation is converted into another string in the form of acronyms, abbreviations, and typos, which are common when manually inputting text, as is true for our public defender and journalism collaborators. Smash can therefore be adopted to match two strings that are modified from the same string but have different abbreviations, acronyms, or typos. For example, Figure 3 shows that "deputy marshall" has three different forms of abbreviations: "dpty mrsl", "dpty marshall", and "deputy mrsl", all of which will be regarded as matches to the entity "deputy marshall" based on Smash. Smash can also deal with the cases when a word includes special characters or is misplaced if this word has the same first character as the original word (e.g., "deputy" vs. "de-uty"). These cases fall into the typo scenario, so their distance will be evaluated using the third case of Equation 2 (i.e., $d_t(W[k], S[i_k : i_{k+1}])$). In our implementation, d_t adopts affine gap. In addition, Smash can handle the case when typos happen in multiple places in a string, which will also be handled by d_t for each pair of word and substring. However, Smash cannot be applied to the cases where two strings are syntactically and structurally different but have the same semantic meanings, such as "The Big Apple" and "New York City", or those that rely on external domain knowledge. Our focus is more on syntactic cases, which are rather common in practice with manual data entry.

6 IMPLEMENTATION

We implement Smash as a function in Python, which takes two strings and returns the distance. Therefore, Smash can be adopted as a drop-in replacement of a metric in many data-cleaning scenarios. For example, it can be adopted to match strings for a full dataset in a pair-wise fashion or a partition of a dataset after blocking. It can also be adopted in various data cleaning algorithms, such as clustering algorithms based on string similarity.

To help public defenders, data journalists, and other non-programmers with similar data cleaning requirements use Smash, we integrate it in OpenRefine [5], a GUI-based open-source tool for

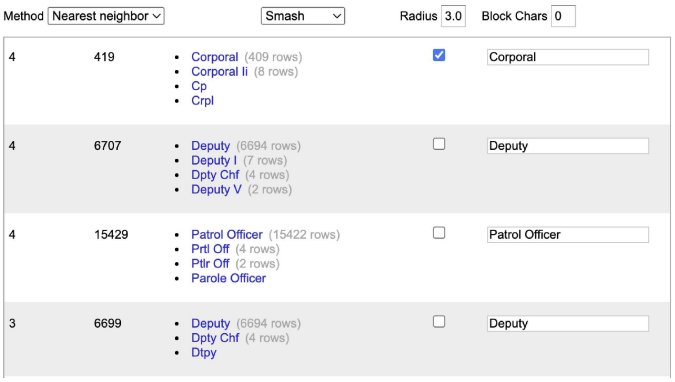

Figure 9: Integrating Smash in OpenRefine: a GUI-based data cleaning tool for use by non-programmers

data cleaning. OpenRefine allows users to select a metric for measuring the similarity between two strings and cluster the strings of a dataset based on that metric. Figure 9 shows an example of using Smash to clean the Police Roster dataset, discussed in Section 7.1. At the top, the user selects Nearest Neighbor Search along with Smash to cluster the data. The *Radius* in Figure 9 represents the distance threshold that determines whether two strings are regarded as a match. It is set as 3.0 in this example. *Block Chars* refers to the minimum number of contiguous characters two strings must share to be considered for a match in OpenRefine. For the example in Figure 9, it is set to 0 because the user does not want to rule out any possible matches before they are presented to Smash. The clustering algorithm will generate many clusters, where each one represents a set of strings that refer to the same entity. For example, Figure 9 shows four clusters, presented as four rows. The first three fields represent the number of distinct strings (e.g., 4 for the first row), the number of strings, and distinct strings for this cluster. The user can decide whether to accept this cluster using the check box and if so, they will replace the strings for that cluster with a string in the input text box (e.g., "Corporal" for the first row). This process is repeated until the user is satisfied with the results. For our Police Roster dataset, the number of user iterations was 5, which is a substantial reduction from the two weeks it took for our public defenders to do the same task.

7 EXPERIMENTS

Our experiments address the following research questions:

- How much does Smash improve precision, recall, and F-score for string matching, compared to existing approaches? (Section 7.2)
- What is the execution time for string matching when using Smash, as compared to existing approaches? (Section 7.3)
- How much do the optimizations of considering the short and skip words in our dynamic programming algorithm improve precision, recall, and F-score? (Section 7.4)

While we compare Smash with six baselines, as discussed next, we further dig deeper into one state-of-the-art synonym rules-based approach, pkduck. In addition, we compare Smash with a large language model (LLM)-based approach. Specifically, we further address the two questions:

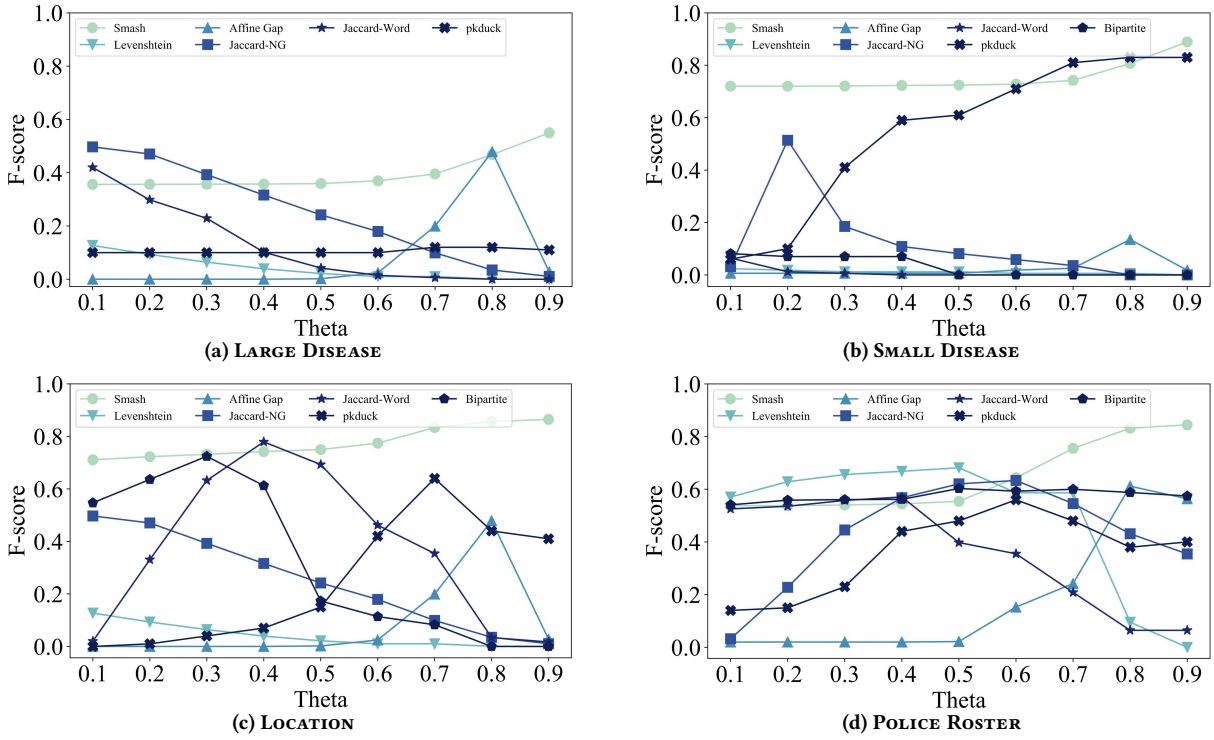

Figure 10: F-score under varied thresholds

- How do the refinement rules for pkduck impact its performance? (Section 7.5)
- How does ChatGPT with GPT-4.0 perform for string matching? (Section 7.6)

7.1 Datasets, Baselines, and Configurations

Datasets. Our experiments include four datasets. Police Roster contains 31,516 rows of police officer data. We test the "Title" column, which includes 154 distinct values. After manual inspection, we find 99 of these distinct values are standard titles, and 55 of them are modified forms of standard titles; each standard title has zero, one, or multiple modified forms.

Large Disease [3] contains 405,543 rows of medical data relating to disease. Each row stores information about a medical term, including a standard and a modified form of this medical term and additional metadata. This dataset includes many acronyms and abbreviations, but not misspellings. We evaluate a sample of this dataset since evaluating the full dataset is too time-consuming. The sampled dataset includes 30,000 rows (approx. 7.5%). We compare the pairs of standard and modified forms for the medical terms.

Small Disease contains a subset of Large Disease and includes 634 disease names. Like the Large Disease dataset, we evaluate the pairs of standard and modified disease names.

Location is a dataset that includes location names (e.g., street names, city names, etc.) [30]. This dataset includes a ground truth dataset that contains 116 pairs of standard and modified location names referring to the same location.

For all datasets, we conduct pair-wise comparisons without blocking for Smash and all baselines when evaluating their performance because our contribution is on the novel metric, Smash, and Smash can be used together with existing blocking techniques [20].

Baselines. We compare Smash with the following baselines: Levenshtein distance [4], affine gap distance [11], Jaccard-Word [36], Jaccard-NG [17], pkduck [30], and Bipartite, a set similarity approach using a bipartite graph [10, 25]. Recall that Jaccard-Word tokenizes two strings into two sets of words and computes the Jaccard score across them, while Jaccard-NG creates n-grams for the two input strings and computes the Jaccard score over the two sets of n-grams. We use 3-grams in the experiments. For Bipartite, we tokenize two strings into two sets of words and compute the similarity by building a bipartite graph on the two sets. Specifically, each edge of the bipartite graph is assigned a value, representing the similarity of a pair of words of the two sets, and is evaluated using Jaccard-NG. The goal of Bipartite is to find the bipartite graph matching with a minimal sum across edges. Finally, the similarity of Bipartite is the minimal sum divided by the number of edges, which is between 0 and 1. We use the SciPy library to find the bipartite graph matching that yields the minimal sum [1].

Configurations. We use a list of generic stop words (e.g., "at", "in", and "is") [6] and regard words with no more than 4 characters as short words. We convert a distance (i.e., for Smash, Levenshtein, and affine gap) into a similarity score such that we can compare them with similarity-based approaches (e.g., Jaccard-Word). Recall that a similarity score sits between 0 and 1, and a larger distance leads to a smaller similarity score. Our observation is that if the distance between two strings is larger than 10, they are unlikely to

Table 2: The maximum and mean F-scores

	Large Disease		Small Disease		Location		Police Roster	
	Max	Mean	Max	Mean	Max	Mean	Max	Mean
Smash	**0.55**	**0.40**	**0.89**	**0.75**	**0.86**	**0.78**	**0.84**	**0.64**
Bipartite	N/A	N/A	0.08	0.03	0.72	0.32	0.60	0.58
Levenshtein	0.13	0.04	0.02	0.01	0.13	0.04	0.68	0.50
Affine Gap	0.48	0.08	0.14	0.03	0.48	0.08	0.61	0.19
Jaccard-Word	0.42	0.12	0.06	0.01	0.78	0.37	0.57	0.36
Jaccard-NG	0.50	0.25	0.51	0.11	0.50	0.25	0.63	0.43
pkduck	0.12	0.11	0.83	0.55	0.64	0.24	0.56	0.36

Table 3: The precision, recall, and F-score for the four datasets

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Smash	0.27	0.74	**0.4**	0.35	0.7	0.47	0.47	0.66	**0.55**
Bipartite	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Levenshtein	1	0.01	0.01	1	0	0	1	0	0
Affine Gap	0.11	0.89	0.2	0.61	0.4	**0.48**	0.99	0.01	0.03
Jaccard-Word	1	0	0	1	0	0	1	0	0
Jaccard-NG	0.96	0.05	0.1	0.99	0.02	0.04	1	0.01	0.01
pkduck	0.12	0.15	0.13	0.16	0.12	0.14	0.19	0.1	0.13

(a) Large Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Smash	0.64	0.89	0.74	0.74	0.89	0.81	0.89	0.88	**0.89**
Bipartite	1	0	0	1	0	0	1	0	0
Levenshtein	1	0	0.01	1	0	0.01	1	0	0
Affine Gap	0.01	0.99	0.03	0.07	0.94	0.14	1	0.01	0.02
Jaccard-Word	1	0	0	1	0	0	1	0	0
Jaccard-NG	0.99	0.02	0.04	1	0	0	1	0	0
pkduck	0.88	0.74	**0.81**	0.97	0.72	**0.83**	0.99	0.72	0.83

(b) Small Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Smash	0.84	0.83	**0.83**	0.92	0.8	**0.86**	0.95	0.79	**0.86**
Bipartite	1	0.04	0.08	1	0	0	1	0	0
Levenshtein	1	0.38	0.55	1	0.06	0.11	1	0	0
Affine Gap	0.21	0.97	0.35	0.82	0.81	0.81	1	0.22	0.37
Jaccard-Word	0.99	0.22	0.35	0.99	0.22	0.35	1	0.01	0.02
Jaccard-NG	0.7	0.86	0.77	0.92	0.72	0.8	0.99	0.41	0.58
pkduck	0.76	0.55	0.64	0.94	0.28	0.44	0.97	0.26	0.41

(c) Location

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Smash	0.69	0.85	**0.76**	0.86	0.83	**0.84**	0.94	0.8	**0.86**
Bipartite	0.9	0.45	0.6	0.91	0.43	0.59	0.93	0.42	0.57
Levenshtein	0.99	0.44	0.61	1	0.07	0.13	1	0	0
Affine Gap	0.11	0.97	0.19	0.63	0.61	0.62	0.95	0.42	0.59
Jaccard-Word	0.98	0.12	0.21	0.98	0.03	0.06	0.98	0.03	0.06
Jaccard-NG	0.86	0.4	0.55	0.89	0.28	0.43	0.98	0.23	0.38
pkduck	0.83	0.33	0.48	0.83	0.25	0.38	1	0.25	0.4

(d) Police Roster

Table 4: Effectiveness of our optimizations that consider skipping stop words and short words

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
NoOpt	0.32	0.7	**0.44**	0.39	0.66	**0.49**	0.51	0.63	**0.56**
StopOpt	0.31	0.7	0.43	0.39	0.66	0.49	0.51	0.63	0.56
ShortOpt	0.27	0.74	0.4	0.35	0.7	0.47	0.47	0.66	0.55
BothOpt	0.27	0.74	0.4	0.35	0.7	0.47	0.47	0.66	0.55

(a) Large Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
NoOpt	0.66	0.86	**0.75**	0.75	0.86	0.8	0.9	0.85	0.88
StopOpt	0.66	0.86	**0.75**	0.75	0.86	0.8	0.9	0.85	0.88
ShortOpt	0.64	0.89	0.74	0.74	0.89	**0.81**	0.89	0.88	**0.89**
BothOpt	0.64	0.89	0.74	0.74	0.89	**0.81**	0.89	0.88	**0.89**

(b) Small Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
NoOpt	0.96	0.62	0.75	0.97	0.6	0.74	0.97	0.59	0.74
StopOpt	0.95	0.65	0.77	0.97	0.63	0.76	0.97	0.62	0.76
ShortOpt	0.84	0.83	**0.83**	0.92	0.8	**0.86**	0.95	0.79	**0.86**
BothOpt	0.84	0.83	**0.83**	0.92	0.8	**0.86**	0.95	0.79	**0.86**

(c) Location

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
NoOpt	0.74	0.71	0.73	0.89	0.58	0.7	0.95	0.51	0.66
StopOpt	0.74	0.75	0.74	0.88	0.61	0.72	0.95	0.54	0.69
ShortOpt	0.69	0.85	**0.76**	0.86	0.83	**0.84**	0.94	0.8	**0.86**
BothOpt	0.69	0.85	**0.76**	0.86	0.83	**0.84**	0.94	0.8	**0.86**

(d) Police Roster

be a match, so we set their similarity to 0 in this case. Otherwise, we normalize the distance d by dividing it by 10 and subtracting this value from one to give the similarity.

7.2 Precision, Recall, and F-Score

In this subsection, we compare precision, recall, and F-score (*PRF* scores for short) of Smash and the baselines on the four datasets. Recall that these datasets include standard and modified forms

Table 5: Performance impact of refinement rules for pkduck

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Refiner on	0.12	0.15	0.13	0.16	0.12	0.14	0.19	0.1	0.13
Refiner off	0.08	0.24	0.12	0.09	0.19	0.12	0.09	0.15	0.11
Smash	0.27	0.74	**0.4**	0.35	0.7	**0.47**	0.47	0.66	**0.55**

(a) Large Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Refiner on	0.88	0.74	**0.81**	0.97	0.72	**0.83**	0.99	0.72	0.83
Refiner off	0.71	0.88	0.78	0.77	0.86	0.81	0.79	0.85	0.82
Smash	0.64	0.89	0.74	0.74	0.89	0.81	0.92	0.8	**0.86**

(b) Small Disease

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Refiner on	0.76	0.55	0.64	0.94	0.28	0.44	0.97	0.26	0.41
Refiner off	0.44	0.63	0.52	0.72	0.34	0.46	0.80	0.28	0.42
Smash	0.84	0.83	**0.83**	0.92	0.8	**0.86**	0.95	0.79	**0.86**

(c) Location

	θ=0.7			θ=0.8			θ=0.9		
	P	R	F	P	R	F	P	R	F
Refiner on	0.83	0.33	0.48	0.83	0.25	0.38	1	0.25	0.4
Refiner off	0.73	0.49	0.59	0.72	0.43	0.54	0.79	0.43	0.55
Smash	0.69	0.85	**0.76**	0.86	0.83	**0.84**	0.94	0.8	**0.86**

(d) Police Roster

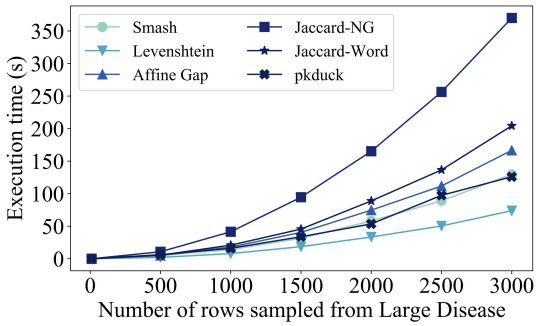

Figure 11: The execution time of string matching for a varied number of sampled rows from the Large Disease dataset

of strings. To compute the PRF scores, we compute the similarity scores between each standard form and each modified form (i.e., the cross-product of the two lists of strings). If the metric is higher than a threshold θ, we regard the two strings as a match. Otherwise, they do not match. Finally, we compare these results with the ground truth to compute the PRF scores. We first report F-scores using a wide range of thresholds (i.e., from 0.1 to 0.9) and then dig into the PRF scores for three thresholds, 0.7, 0.8, and 0.9, as in prior work [30].

First, we vary the thresholds from 0.1 to 0.9 with an interval of 0.1 and report the F-scores in Figure 10. Note that the results for the Large Disease dataset (i.e., Figure 10(a)) do not include Bipartite because it does not finish within 1 hour for this dataset. The reason is that building bipartite graphs is expensive and this dataset has many rows (i.e., 30,000). We see that Smash outperforms the baselines in most cases and has the highest F-score across all thresholds for each dataset. In addition, Smash has the highest mean F-scores over all thresholds compared to the baselines, as summarized in Table 2. Specifically, Smash improves the max and mean F-score compared to the best baselines by up to 23.5% (i.e., Levenshtein on Police Roster) and 110.8% (i.e., Jaccard-Word on Location), respectively. These results show that Smash can achieve the best performance across varied thresholds and is robust to the threshold parameter compared to the baseline approaches for different datasets.

Next, we report the PRF scores for thresholds 0.7, 0.8, and 0.9 in Table 3. We see that Smash has a higher F-score than baselines in most tests. For the Small Disease dataset, Smash has a smaller F-score than pkduck when θ is 0.7 or 0.8. However, Smash significantly outperforms pkduck in many other cases. Specifically, compared to pkduck, Smash improves the F-score by 58% on average and 323% in the best case (i.e., Large Disease, $\theta = 0.9$). Smash has a smaller F-score than affine gap when we test the Large Disease dataset and θ is 0.8. But Smash demonstrates a much higher F-score in other cases. We have also observed cases where the precision is 1 but the recall is 0 (e.g., Levenshtein, $\theta = 0.9$) because the metric does not find any matching pairs. If no matches are returned, the precision is trivially 1 because there are no false positives. There are also some cases when $P = 1$ and $R = 0$, but $F > 0$. This discrepancy is solely due to rounding imprecision when displaying the results. Overall, this experiment demonstrates that Smash significantly improves the F-score for string matching over many existing approaches under a wide range of datasets.

7.3 Execution Time of String Matching

Our next experiment evaluates the execution time of string matching using different similarity metrics. We sample a varied number of rows from the Large Disease, which has 30K rows and is our largest dataset, and report the time for string matching for the sampled data in Figure 11. We see that Smash has a similar execution time to pkduck although Smash has higher F-scores than pkduck in most test cases. While Smash has a higher execution time compared to the Levenshtein metric, it has much higher F-scores in most test cases. Finally, Smash has a smaller execution time than the other baselines. Note that the results for Bipartite are not reported because Bipartite only finishes within one hour when the number of sampled rows is no larger than 1,000.

7.4 Effectiveness of Considering Skipping Stop Words and Short Words

We now evaluate the optimizations that consider skipping stop words and short words. We test four variants of Smash: i) NoOpt: no optimizations, ii) StopOpt: only considering skipping stop words, iii) ShortOpt: only considering skipping short words, and iv) BothOpt: applying both optimizations.

Table 4 shows the results for each dataset. Each row represents the PRF scores of each variant of Smash for the three thresholds, 0.7, 0.8, and 0.9. Table 4 includes the results for all four datasets.

Table 6: Comparing Smash with ChatGPT (GPT-4.0)

	P	R	F
Smash ($\theta = 0.7$)	0.84	0.83	0.83
Smash ($\theta = 0.8$)	0.92	0.8	0.86
Smash ($\theta = 0.9$)	0.95	0.79	**0.86**
ChatGPT (Prompt 1)	1.0	0.27	0.43
ChatGPT (Prompt 2)	0.99	0.36	0.52
ChatGPT (Prompt 3)	0.96	0.39	0.56

We see that BothOpt significantly improves the F-score compared to NoOpt in all cases for the Location and Police Roster datasets. The improvement is up to 30% (i.e., Police Roster, $\theta = 0.9$). For the Large Disease and Small Disease datasets, there are four cases where BothOpt slightly degrades performance. The reason is that these two datasets contain many stop words and short words that have semantic meanings, and skipping those words may lead to false matches. For example, the Large Disease dataset has a standard form "BCG vaccine", where "BCG" are the initials of the people who invented this vaccine. If we apply the optimization of skipping the short words, we may skip "BCG" and then we will match the remaining string "vaccine" with "vax", which is an abbreviation for the general term "vaccine". However, "BCG vaccine" and "vax" refer to different entities in this dataset. For StopOpt, it improves the performance over NoOpt for the Location dataset. ShortOpt has higher F-scores compared to StopOpt for the Location and Police Roster datasets because most stop words are short (i.e., no larger than 4 characters) and are also considered by ShortOpt.

7.5 Performance Impact of Refinement Rules for pkduck

Recall that pkduck automatically generates synonym rules for performing string matching. It initially generates a set of candidate synonym rules based on the longest common sequence of each pair of strings and refines these rules by applying manually-developed refinement rules. We now evaluate the effectiveness of pkduck's refinement rules by testing two variants of pkduck that turn on and off the refinement, respectively.

Table 5 shows the PRF scores for the thresholds 0.7, 0.8, and 0.9 under the four datasets. We see that the refinement rules do not always improve F-scores. Specifically, the refinement rules improve the performance for the Large Disease and Small Disease datasets while they degrade the performance of pkduck for the other two datasets in most cases. The only exception is for the Location dataset when θ is 0.7. Smash, on the other hand, does not rely on synonym rules and has much higher F-scores than the two variants of pkduck in most cases.

7.6 Performance of ChatGPT

LLMs have demonstrated promise in understanding and generating text. Here, we test if a state-of-the-art LLM can provide performance comparable to Smash out of the box. Specifically, we asked the state-of-the-art LLM, ChatGPT Pro with GPT-4.0 [2] to perform string matching and evaluate the quality of the results. Our experiments are on the Police Roster.

We used three prompts. The first one is "Here are the police titles. They include acronyms, abbreviations, and typos. I want to find the pair of strings that represent the same entity. Please do the pair-wise comparison and give me the answer. Give me the pairs without using Python code using the format of "String A, String B": [Police Roster dataset]". The second one is similar, with the only difference being that we stress the importance of returning all pairs: "... **Ensure the completeness of all pairs and** give me the pairs...". Since we find the second prompt has a low recall (i.e., only returning 36 pairs), we tried the third and final prompt, explicitly asking ChatGPT to return 100 pairs: "... **Give me at least 100 pairs** without using Python code...". However, ChatGPT only returns 48 pairs, saying that it cannot find more pairs that represent the same police title.

The first prompt returns 26 pairs, whose precision is 1.0 as shown in Table 6. However, it has a low recall because it only returns a limited number of pairs. The second prompt, instead, returns 36 pairs, which has a higher recall compared to the first one, but a slightly lower precision. The third prompt returns 48 pairs but still has a low recall. The F-score for the three prompts is much smaller than the one Smash achieves. Specifically, the max F-score for ChatGPT is 0.56 while the max F-score for Smash is 0.86. Overall, we believe today's LLMs are not quite able to perform string matching tasks effectively, especially for datasets that include domain-specific acronyms, abbreviations, and typos (e.g., the Police Roster dataset). Despite our best attempts at prompt engineering, LLMs only return results they are truly confident about, thereby resulting in very low recall—something we were unable to fix even by explicitly requesting a certain number of results.

8 CONCLUSION

In this paper, we presented Smash, a string similarity measure that captures acronyms, abbreviations, and typos as is typical in real-world settings, without relying on brittle synonym rules. We identified an optimal substructure for Smash, developed a dynamic programming algorithm to efficiently compute Smash, and proposed two optimizations to further improve the accuracy of Smash. We implemented Smash in OpenRefine to help non-programmers, such as journalists and public defenders, more accurately perform string matching. Finally, we performed extensive experiments that demonstrate Smash significantly improves F-score over six existing approaches. Compared to the best baselines, Smash improves the max and mean F-score by 23.5% and 110.8%, respectively.

ACKNOWLEDGMENTS

We acknowledge support from grants DGE-2243822, IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, funds from the State of California, an NDSEG Fellowship, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: G-Research, Adobe, Microsoft, Google, and Sigma Computing.

REFERENCES

[1] Bipartite matching. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.maximum_bipartite_matching.html [Online; accessed 17-Novembor-2023].

[2] Chatgpt. https://chat.openai.com/ [Online; accessed 12-Mar-2024].

[3] Disease dataset. https://zenodo.org/record/4266963 [Online; accessed 27-February-2023].

[4] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance [Online; accessed 27-February-2023].

[5] Openrefine. https://github.com/OpenRefine/OpenRefine [Online; accessed 27-February-2023].

[6] Stopwords. https://www.ranks.nl/stopwords [Online; accessed 27-February-2023].

[7] Subsequence. https://en.wikipedia.org/wiki/Subsequence [Oneline; accessed 17-Novembor-2023].

[8] M. Almagro, E. J. Almazán, D. Ortego, and D. Jiménez. LEA: improving sentence similarity robustness to typos using lexical attention bias. In A. K. Singh, Y. Sun, L. Akoglu, D. Gunopulos, X. Yan, R. Kumar, F. Ozcan, and J. Ye, editors, *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 36–46. ACM, 2023.

[9] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 40–49. IEEE Computer Society, 2008.

[10] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 918–929. ACM, 2006.

[11] M. Bilenko and R. J. Mooney. Employing trainable string similarity metrics for information integration. In *IIWeb*, pages 67–72, 2003.

[12] P. S. G. C., A. Ardalan, A. Doan, and A. Akella. Smurf: Self-service string matching using random forests. *Proc. VLDB Endow.*, 12(3):278–291, 2018.

[13] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In L. Liu, A. Reuter, K. Whang, and J. Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5. IEEE Computer Society, 2006.

[14] J. Dai, M. Zhang, G. Chen, J. Fan, K. Y. Ngiam, and B. C. Ooi. Fine-grained concept linking using neural networks in healthcare. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 51–66. ACM, 2018.

[15] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *Proc. VLDB Endow.*, 10(10):1082–1093, 2017.

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

[17] O. Hassanzadeh, M. Sadoghi, and R. J. Miller. Accuracy of approximate string joins using grams. In V. Ganti and F. Naumann, editors, *Proceedings of the Fifth International Workshop on Quality in Databases, QDB 2007, at the VLDB 2007 conference, Vienna, Austria, September 23, 2007*, pages 11–18, 2007.

[18] Y. He, K. Ganjam, and X. Chu. SEMA-JOIN: joining semantically-related tables using big table corpora. *Proc. VLDB Endow.*, 8(12):1358–1369, 2015.

[19] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, 2014.

[20] B. Li, Y. Liu, A. Zhang, W. Wang, and S. Wan. A survey on blocking technology of entity resolution. *J. Comput. Sci. Technol.*, 35(4):769–793, 2020.

[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 257–266. IEEE Computer Society, 2008.

[22] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, 2011.

[23] Y. Li, J. Li, Y. Suhara, A. Doan, and W. Tan. Deep entity matching with pre-trained language models. *Proc. VLDB Endow.*, 14(1):50–60, 2020.

[24] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 373–384. ACM, 2013.

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

[26] P. Mundra, J. Zhang, F. Nargesian, and N. Augsten. Koios: Top-k semantic overlap set search. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 1531–1543. IEEE, 2023.

[27] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, 2012.

[28] G. Song, H. Lee, K. Shim, Y. Park, and W. Kim. String joins with synonyms. In Y. Nah, B. Cui, S. Lee, J. X. Yu, Y. Moon, and S. E. Whang, editors, *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24-27, 2020, Proceedings, Part III*, volume 12114 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2020.

[29] G. Song, K. Shim, and H. Lee. Substring similarity search with synonyms. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 2003–2008. IEEE, 2021.

[30] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. In *Proceedings of the VLDB Endowment Volume 11, Issue 1*, pages 53–65. ACM, 2017.

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

[32] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 85–96. ACM, 2012.

[33] J. Wang, C. Lin, M. Li, and C. Zaniolo. An efficient sliding window approach for approximate entity extraction with synonyms. In M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 109–120. OpenProceedings.org, 2019.

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

[35] Z. Wen, D. Deng, R. Zhang, and R. Kotagiri. 2ed: An efficient entity extraction algorithm using two-level edit-distance. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 998–1009. IEEE, 2019.

[36] Wikipedia contributors. Jaccard index, 2023. [Online; accessed 27-February-2023].

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

[38] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In J. Huai, R. Chen, H. Hon, Y. Liu, W. Ma, A. Tomkins, and X. Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140. ACM, 2008.

[39] P. Xu and J. Lu. Towards a unified framework for string similarity joins. *Proc. VLDB Endow.*, 12(11):1289–1302, 2019.