Similarity Joins for Uncertain Strings*

Manish Patil
Louisiana State University
USA
mpatil@csc.lsu.edu

Rahul Shah
Louisiana State University
USA
rahul@csc.lsu.edu

ABSTRACT

A string similarity join finds all similar string pairs between two input string collections. It is an essential operation in many applications, such as data integration and cleaning, and has been extensively studied for deterministic strings. Increasingly, many applications have to deal with imprecise strings or strings with fuzzy information in them. This work presents the first solution for answering similarity join queries over uncertain strings that implements possible-world semantics, using the edit distance as the measure of similarity. Given two collections of uncertain strings $\mathcal{R}$, $\mathcal{S}$, and input (k, τ), our task is to find string pairs (R, S) between collections such that $Pr(ed(R, S) \leq k) > \tau$ i.e., the probability of the edit distance between R and S being at most k is more than probability threshold τ. We can address the join problem by obtaining all strings in $\mathcal{S}$ that are similar to each string R in $\mathcal{R}$. However, existing solutions for answering such similarity search queries on uncertain string databases only support a deterministic string as input. Exploiting these solutions would require exponentially many possible worlds of R to be considered, which is not only ineffective but also prohibitively expensive. We propose various filtering techniques that give upper and (or) lower bound on $Pr(ed(R, S) \leq k)$ without instantiating possible worlds for either of the strings. We then incorporate these techniques into an indexing scheme and significantly reduce the filtering overhead. Further, we alleviate the verification cost of a string pair that survives pruning by using a trie structure which allows us to overlap the verification cost of exponentially many possible instances of the candidate string pair. Finally, we evaluate the effectiveness of the proposed approach by thorough practical experimentation.

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Systems—*Query processing*

*This work is supported in part by National Science Foundation (NSF) Grants CCF–1017623 and CCF–1218904.

SIGMOD/PODS'14, June 22 - 27 2014, Snowbird, UT, USA

Keywords

Uncertain strings; string joins; edit distance

1. INTRODUCTION

Strings form a fundamental data type in computer systems and string searching has been extensively studied since the inception of computer science. String similarity search takes a set of strings and a query string as input, and outputs all the strings in the set that are similar to the query string. A join extends the notion of similarity search further and require all similar string pairs between two input string sets to be reported. Both similarity search and similarity join are central to many applications such as data integration and cleaning. Edit distance is the most commonly used similarity measure for strings. The edit distance between two strings r and s, denoted by $ed(r, s)$, is the minimum number of single-character edit operations (insertion, deletion, and substitution) needed to transform r to s. Edit distance based string similarity search and join has been extensively studied in the literature for deterministic strings [7, 3, 2, 13, 18, 5]. However, due to the large number of applications where uncertainty or imprecision in values is either inherent or desirable, recent years have witnessed increasing attention devoted to managing uncertain data. Several probabilistic database management systems (PDBMS), which can represent and manage data with explicit probabilistic models of uncertainty, have been proposed to date [17, 16]. Imprecision in data introduces many challenges for similarity search and join in databases with probabilistic string attributes, which is the focus of this paper.

Uncertainty model: Analogous to the models of uncertain databases, two models - string-level and character-level - have been proposed recently by Jeffrey Jestes et al. [10] for uncertain strings. In the string-level uncertainty model all possible instances for the uncertain string are explicitly listed to form a probability distribution function (pdf). In contrast, the character-level model describes distributions over all characters in the alphabet for each uncertain position in the string. We focus on the character-level model as it is realistic and concise in representing the string uncertainty. Let $\Sigma = \{c_1, c_2, ..., c_\sigma\}$ be the alphabet. A character-level uncertain string is $S = S[1]S[2]...S[l]$, where $S[i]$ $(1 \leq i \leq l)$ is a random variable with discrete distribution over Σ i.e., $S[i]$ is a set of pairs $(c_j, p_i(c_j))$, where $c_j \in \Sigma$ and $p_i(c_j)$ is the probability of having symbol c_j at position i. Formally $S[i] = \{(c_j, p_i(c_j)) | c_j \neq c_m \text{ for } j \neq m, \text{ and } \sum_j p_i(c_j) = 1\}$. When the context of a string is unclear we represent $p_i(c_j)$

for string S by $Pr(S[i] = c_j)$. Throughout we use a lower case character to represent a deterministic string (s) against the uncertain string denoted by a upper case character (S). Let $|S|$ ($|s|$) be the length of string S (s). Then the possible worlds of S is a set of all possible instances s of S with probability $p(s)$, $\sum p(s) = 1$. S being a character-level uncertain string, $|S| = |s|$ for any of its possible instances.

Query semantics: In addition to capturing uncertainty in the data, one must define the semantics of queries over the data. In this regard, a powerful model of possible-world semantics has been the backbone of analyzing the correctness of database operations on uncertain data. For uncertain string attributes, Jestes et al. [10] made the first attempt to extend the notion of similarity. They used expected edit distance (*eed*) over all possible worlds of two uncertain strings. Given strings R and S, $eed(R, S) = \sum_{r_i, s_j} p(r_i)p(s_j)ed(r_i, s_j)$, where s_j (r_i) is an instance of S (R) with probability $p(s_j)$ ($p(r_i)$). Though *eed* seems like a natural extension of edit distance as a measure of similarity, it has been shown that it does not implement the possible-world semantics completely at the query level [6]. Consider a similarity search query on a collection of deterministic strings with input string r. Then, string s is an output only if $ed(r, s) \leq k$. For such a query R over an uncertain string collection, possible world semantics dictate that we apply the same predicate $ed(r, s) \leq k$ for each possible instance r of R, s of S and aggregate this over all worlds. Thus, a possible world with instances r, s can contribute in deciding whether S is similar to R only if s is within the desired edit distance of r. However, for the *eed* measure, all possible worlds (irrespective but weighted by edit distance) contribute towards the overall score that determines the similarity of S with R. To overcome this problem, in [6] the authors have proposed a (k, τ)-matching semantic scheme. Using this semantic, given a edit distance threshold k and probability threshold τ, R is similar to S if $Pr(ed(R, S) \leq k) > \tau$. We use this similarity definition in this paper for answering join queries.

Problem definition: Given two sets of uncertain strings $\mathcal{R}$ and $\mathcal{S}$, an edit-distance threshold k and a probability threshold τ, similarity join finds all similar string pairs $(R, S) \in \mathcal{R} \times \mathcal{S}$ such that $Pr(ed(R, S) \leq k) > \tau$. Without loss of generality, we focus on self join in this paper i.e. $\mathcal{R} = \mathcal{S}$.

Related work: Uncertain/Probabilistic strings have been the subject of study for the past several years. Efficient algorithms and data structures are known for the problem of string searching in uncertain text [8, 1, 9, 19]. In [6] authors have studied the approximate substring matching problem, where the goal is to report the positions of all substrings of uncertain text that are similar to the query string. Recently, the problem of similarity search on a collection of uncertain strings has been addressed in [4]. However, most of these works support only deterministic strings as query input. Utilizing these techniques for uncertain string as input would invariably need all its possible worlds to be enumerated, which may not be feasible to do taking into account the resultant exponential blowup in query cost. Though the problem of similarity join on uncertain strings has been studied in [10], it makes use of expected edit distance as a measure of similarity. We make an attempt to address some of the challenges involved in uncertain string processing by investigating similarity joins on them in this paper.

Our Contributions: In this paper, we present a comprehensive investigation on the problem of similarity joins for uncertain strings using (k, τ)-matching [6] as the similarity definition and make the following contributions:

- We propose a filtering scheme that integrates q-gram filtering with probabilistic pruning, and we present an indexing scheme to facilitate such filtering.
- We extend the frequency distance filtering introduced in [4] for an uncertain string pair and improve its performance while maintaining the same space requirement.
- We propose the use of a trie data structure to efficiently compute the exact similarity probability (as given by (k, τ)-matching) for a candidate pair (R, S) without explicitly comparing all possible string pairs.
- We conduct comprehensive experiments which demonstrate the effectiveness of all proposed techniques in answering similarity join queries.

2. PRELIMINARIES

In this section we briefly review filtering techniques for deterministic strings available in literature and extend them for uncertain strings later in the article. Let r, s be the two deterministic strings and k be the edit distance threshold.

2.1 q-gram Filtering

We partition s into $k+1$ disjoint segments $s^1, s^2, ..., s^{k+1}$. For simplicity let each segment be of length $q \geq 1$ i.e, $s^x = s[((x-1)q+1)..xq]$. Further, let $pos(s^x)$ represent the starting position of segment s^x in string s i.e, $pos(s^x) = (x-1)q+1$. Then using the pigeonhole principle, if r is similar to a string s, it should contain a substring that matches a segment in s. A naive method to achieve this is to obtain a set $q(r)$ enumerating all substrings of r of length q and for each substring check whether it matches s^x for $x = 1, 2, .., k+1$. However, in [14] authors have shown that we can obtain a set $q(r, x) \subseteq q(r)$ for each segment of s such that it is sufficient to test each substring $w \in q(r, x)$ for a match with s^x. Table 1 shows sets $q(r, x)$ populated for a sample string r. Using the proposed "position aware" substring selection, set $q(r, x)$ includes substrings of r with start positions in the range $[pos(s^x) - \lfloor (k-\Delta)/2 \rfloor, pos(s^x) + \lfloor (k+\Delta)/2 \rfloor]$ and with length q. The number of substrings in set $q(r, x)$ is thus bounded by $k+1$. In [14] authors prove that the substring selection satisfies "completeness" ensuring any similar pair (r, s) will be found as a candidate pair. We use a generalization of this filtering technique by partitioning s into $m > k$ partitions [14, 15] as summarized in the following lemma.

LEMMA 1. *Given a string r and s, with s partitioned into $m > k$ disjoint segments, if r is similar to s within an edit threshold k, r must contain substrings that match at-least $(m-k)$ segments of s.*

Once again by assuming each segment of s to be of length $q \geq 1$, we can compute the set $q(r, x)$ and attempt to match each $w \in q(r, x)$ with s^x as before to apply the above lemma.

2.2 Frequency Distance Filtering

Given a string s from the alphabet Σ, a frequency vector $f(s)$ is defined as $f(s) = [f(s)_1, f(s)_2, ..., f(s)_\sigma]$, where $f(s)_i$ is the count of the ith alphabet of Σ i.e, c_i. Let $f(r)$ and $f(s)$ be the frequency vectors of r and s respectively. Then the frequency distance of r and s is defined as $fd(r, s) = max\{pD, nD\}$,

$$pD = \sum_{f(r)_i > f(s)_i} f(r)_i - f(s)_i, nD = \sum_{f(r)_i < f(s)_i} f(s)_i - f(r)_i$$

Frequency distance provides a lower bound for the edit distance between r and s i.e., $fd(r,s) \leq ed(r,s)$ and can be computed efficiently [12]. Thus, we can safely decide that s is not similar to r if $fd(r,s) > k$.

3. q-GRAM FILTERING

In this section we adopt and extend the ideas introduced for deterministic strings earlier in Section 2.1 to uncertain strings. We begin with the simpler case where either of the two uncertain strings R and S is deterministic. Let R be that string with r being its only possible instance. We try to achieve an upper bound on the probability of r and S being similar i.e., $Pr(ed(r,S) \leq k)$. We then build upon this result for the case when both strings are uncertain and obtain an upper bound on the probability of R and S being similar i.e., $Pr(ed(R,S) \leq k)$.

Before proceeding, we introduce some notation and definitions. A string w of length l matches a substring in T starting at position i with probability $Pr(w = T[i..i+l-1]) = \prod_{ps=1}^{l} p_{i+ps-1}(w[ps])$. A string w matches T with probability $Pr(w = T) = \prod_{ps=1}^{l} p_{ps}(w[ps])$ if $|w| = |T| = l$; otherwise it is 0. We simply say w matches with T (or vice versa) if $Pr(w = T) > 0$. The probability of string W matching T is given by $Pr(W = T) = \prod_{ps=1}^{l} \sum_{c_j \in \Sigma} Pr(W[ps] = c_j) \times Pr(T[ps] = c_j)$. Once again, we say W matches T if $Pr(W = T) > 0$ for simplicity.

3.1 Bounding $Pr(ed(r,S) \leq k)$

The possible worlds Ω of S is the set of all possible instances of S. A possible world $pw_j \in \Omega$ is a pair $(s_j, p(s_j))$, where s_j is an instance of S with probability $p(s_j)$. Let $p(pw_j) = p(s_j)$ denote the probability of existence of a possible world pw_j. Note that s_j is a deterministic string and $\sum p(pw_j) = 1$. Then by definition, $Pr(ed(r,S) \leq k) = \sum_{ed(r,s_j) \leq k} p(pw_j)$. We first establish the necessary condition for r to be similar to S within an edit threshold k, i.e., $Pr(ed(r,S) \leq k) > 0$, and then try to provide an upper bound for the same.

Necessary condition for $Pr(ed(r,S) \leq k) > 0$***:***
We partition the string S into $m > k$ disjoint substrings. For simplicity, let q be the length of each partition. Note that each partition $S^1, S^2, ..., S^m$ is an uncertain string. Let r contain substrings matching $m' \leq m$ segments of S i.e., the number of segments of S with $Pr(w = S^x) > 0$ for any substring w of r is m'. Then it can be seen that for any $pw_j \in \Omega$, r contains substrings that match with at most m' segments from $s_j^1, s_j^2, ..., s_j^m$ that partition s_j. Based on this observation, the following lemma establishes the necessary condition for $Pr(ed(r,S) \leq k) > 0$.

LEMMA 2. *Given a string r and a string S partitioned into $m > k$ disjoint segments, for r to be similar to S i.e., $Pr(ed(r,S) \leq k) > 0$, r must contain substrings that match at-least $(m-k)$ segments of S.*

To apply the above lemma, we can obtain a set $q(r,x)$ using position aware selection as described earlier and use it to match against segment S^x. Table 1 shows the above lemma applied to a collection of uncertain strings. None of the segments of S_1 match any substring in r and hence they can not form a candidate pair. For S_2, even though the second segment matches some substring in r, we do not use it as we know by position aware substring selection that such an alignment can not lead to an instance of S that is similar to r. We can reject S_2 as well since it has only one matched segment. Strings S_3 and S_4 survive this pruning step and are taken forward.

Table 1: Application of q-gram Filtering

$m=3$, $q=2$, $k=1$, τ=0.25			
r	GGATCC		
$q(r,x)$	GG GA	GA AT TC	TC CC
S_1	A{(C,0.5),(G,0.5)}A{(C,0.5),(G,0.5)}AC		
	(AC,0.5) (AG,0.5)	(AC,0.5) (AG,0.5)	(AC,1)
S_2	AA{(G,0.9),(T,0.1)}G{(C,0.3),(G,0.2),(T,0.5)}C		
	(AA,1)	(GG,0.9) (TG,0.1)	(CC,0.3)✓ (GC,0.2) (TC,0.5)✓
S_3	G{(A,0.8),(G,0.2)}CT{(A,0.8),(C,0.1),(T,0.1)}C		
	(GA,0.8)✓ (GG,0.2)✓	(CT,1)	(AC,0.8) (CC,0.1)✓ (TC,0.1)✓
S_4	{(G,0.8),(T,0.2)}GA{(C,0.3),(G,0.2),(T,0.5)}CT		
	(GG,0.8)✓ (TG,0.2)	(AC,0.3) (AG,0.2) (AT,0.5)✓	(CT,1)

Computing upper bound for $Pr(ed(r,S) \leq k)$***:***
So far we were interested in knowing if there exists a substring $w \in q(r,x)$ that matches segment S^x. We now try to compute the probability that one or more substrings in $q(r,x)$ match S^x. Let E_x denote such an event with probability α_x. Then $\alpha_x = Pr(E_x) = \sum_{w \in q(r,x)} Pr(w = S^x)$. The correctness of α_x relies on the following observations:

- $q(r,x)$, being a set, contains all distinct substrings.
- Event of substring $w_i \in q(r,x)$ matching S^x is independent of substring $w_j \in q(r,x)$ matching S^x for $w_i \neq w_j$.

Next, our idea is to prune out the possible worlds of S which can not satisfy the edit-distance threshold k with r and obtain a set $\mathcal{C} \subseteq \Omega$ of candidate worlds. We can then use $Pr(\mathcal{C}) = \sum_{pw_j \in \mathcal{C}} p(pw_j)$ as the upper bound on $Pr(ed(r,S) \leq k)$. Consider a possible world pw_j in which s_j is the possible instance of S. s_j being the deterministic string, we can apply the process of q-gram filtering described in Section 2.1 to quickly assess if s_j can give edit distance within threshold k. If yes, pw_j is a candidate world and we include it in $\mathcal{C}$. This naive method requires all possible worlds of S to be instantiated and hence is too expensive to be used. Below we show how to achieve the desired upper bound i.e., $Pr(\mathcal{C})$ without explicitly listing set Ω or $\mathcal{C}$.

For ease of explanation, let $m = k+1$. We partition the possible worlds in Ω into sets $\Omega_0, \Omega_1, ..., \Omega_m$ such that:

- Ω_y includes any possible world pw_j where r contains substrings matching exactly y segments from $s_j^1, ..., s_j^m$ that partition s_j i.e., $y = |\{s_j^x | s_j^x \in q(r,x) \text{ for } x = 1, 2, .., m\}|$.
- $\Omega = \Omega_0 \cup \Omega_1 \cup \Omega_2 \cup ... \cup \Omega_m$
- $\Omega_y \cap \Omega_z = \emptyset$ for $y \neq z$

With this partitioning of Ω, we have following:

$$Pr(\mathcal{C}) = Pr(\Omega_1 \cup \Omega_2 \cup ... \cup \Omega_m) = Pr(\Omega \setminus \Omega_0)$$

$$= Pr(\Omega) - Pr(\Omega_0) = 1 - \prod_{x=1}^{m}(1-\alpha_x)$$

In the above equation, Ω_0 denotes the event that none of the segments of S match substrings of r. By slight abuse of notation, we say S^x matches r (using position aware substring selection) if $\alpha_x > 0$. Then, the following lemma summarizes our result on the upper bound.

LEMMA 3. *Let r and S be the given strings with edit threshold k. If S is partitioned into $m = k+1$ disjoint segments, $Pr(ed(r,S) \leq k)$ is upper bounded by $(1 - \prod_{x=1}^{m}(1-\alpha_x))$, where α_x gives the probability that segment S^x matches r.*

***Generalizing upper bound for** $m > k$:*
Finally, we turn our attention to compute $Pr(\mathcal{C})$ for the scenario where S is partitioned into $m > k$ segments. Once again considering the partitioning of Ω introduced above $Pr(\mathcal{C}) = Pr(\cup_{y=(m-k)}^{m}\Omega_y) = \sum_{y=m-k}^{m} Pr(\Omega_y)$. Then we observe that computing $Pr(\Omega_y)$ in this equation boils down to the following problem: There are m events E_x ($x = 1, 2, ..m$) and we are given $Pr(E_x) = \alpha_x$. What is the probability that exactly y events (among those m events) happen? Our solution is as follows. Let $Pr(i,j)$ denote the probability that, within the first i events, j of them happen. We then have the following recursive equation: $Pr(i,j) = Pr(E_i)Pr(i-1,j-1) + (1-Pr(E_i))Pr(i-1,j)$. By populating the $m \times m$ matrix using a dynamic programming algorithm based on the above recursion, we can lookup the last column to find out $Pr(\Omega_y)$ for $y = m-k, ..., m$. This recursion gives us an efficient ($O(m^2)$) way to compute $Pr(\mathcal{C})$. We note that it is possible to improve the running time to $O(m(m-k))$, but leave out the details for simplicity.

THEOREM 1. *Let r and S be the given strings with edit threshold k. Also assume S is partitioned into $m > k$ disjoint segments and α_x represents the probability that segment S^x matches r. Then $Pr(ed(r,S) \leq k)$ is upper bounded by the probability that at-least $(m-k)$ segments of S match r or in another words probability that r contains substrings matching at-least $(m-k)$ segments of S.*

Continuing the example in Table 1, we now try to apply the above theorem to strings S_3 and S_4. For S_3 we have $\alpha_1 = 1$, $\alpha_2 = 0$, and $\alpha_3 = 0.2$. Therefore the upper bound on S_3's similarity with r is $0.2 < \tau$ and S_3 can be rejected. Even though four out of six possible worlds of S_3 contribute to $\mathcal{C}$, the probability of each of them being small their collective contribution falls short of τ. Similarly the upper bound for S_4 can be computed as 0.4 and the pair (r, S_4) qualifies as a candidate pair. Thus Theorem 1 with an implicit requirement of Lemma 2 to be satisfied integrates q-gram filtering and probabilistic pruning.

Let string S be preprocessed such that each segment S^x is maintained as a list of pairs $(s_j^x, p(s_j^x))$, where s_j^x is an instance of S^x with probability $p(s_j^x)$. Also assume r is preprocessed and sets $q(r,x)$ are available to us for $x = 1, 2, .., m$ ($|q(r,x)| = k+1$, $\sum_{x=1}^{m}|q(r,x)| = (k+1)m$). Then the desired upper bound can be computed efficiently by applying the above theorem as it only adds the following computational overhead in comparison to its counterpart of deterministic strings: (1) computation cost for α_x of each segment is bounded by k and mk overall, (2) the cost of computing $Pr(\mathcal{C})$ using dynamic programming is bounded by $m(m-k)$.

3.2 Bounding $Pr(ed(R,S) \leq k)$

In this subsection, we follow the analysis from the earlier subsection taking into account the uncertainty introduced for string R. The possible worlds Ω of R and S is the set of all possible instances of $R \times S$. A possible world $pw_{i,j} \in \Omega$ is a pair $((r_i, s_j), p(r_i) * p(s_j))$, where s_j (r_i) is an instance of S (R) with probability $p(s_j)$ ($p(r_i)$). Also $p(r_i) * p(s_j)$ denote the probability of existence of a possible world $pw_{i,j}$ and $\sum p(pw_{i,j}) = 1$. Then by definition, $Pr(ed(R,S) \leq k) = \sum_{ed(r_i,s_j)\leq k} p(pw_{i,j})$.

***Necessary condition for** $Pr(ed(R,S) \leq k) > 0$:*
We begin by partitioning the string S into $m > k$ disjoint substrings as before and assume q to be the length of each partition. Then the following lemma establishes the necessary condition for R to be similar to S within edit threshold.

LEMMA 4. *Given a string R and a string S partitioned into $m > k$ disjoint segments, for R to be similar to S i.e., $Pr(ed(R,S) \leq k) > 0$, R must contain substrings that match at-least $(m-k)$ segments of S.*

The correctness of the above lemma can be verified by extending the earlier observation as follows: Let R contain substrings matching $m' \leq m$ segments of S i.e., the number of segments of S with $Pr(W = S^x) > 0$ for any (uncertain) substring W of R is m'. Then for any $pw_{i,j} \in \Omega$, r_i contains substrings that match with at most m' segments from $s_j^1, s_j^2, ..., s_j^m$ that partition s_j. Next, we obtain a set $q(R,x)$ for each segment S^x of S using the position aware substring selection. This allows us to only test substrings $W \in q(R,x)$ for a match against S^x. We highlight that the substring selection mechanism only relies on the length of two strings R and S, start position of a substring W of R and that of S^x. Therefore following same arguments in [14], we can prove that any similar pair (R,S) will be reported as a candidate.

***Computing** α_x:*
Let E_x denote an event that one or more substrings in set $q(R,x)$ match segment S^x and let α_x be its probability. Using a trivial extension of the earlier result in Section 3.1, we could perhaps compute $\alpha_x = Pr(E_x) = \sum_{W \in q(R,x)} Pr(W = S^x)$. However, we show that this leads to incorrect computation of α_x and requires a careful investigation. Let $R = A\{(A,0.8),(C,0.2)\}AATT$, $S = A\{(A,0.8),(C,0.2)\}AGCT$, $k = 1$ and $q = 3$. Then, we have $S^1 = A\{(A,0.8),(C,0.2)\}A$, $q(R,1) = \{A\{(A,0.8),(C,0.2)\}A,\ \{(A,0.8),(C,0.2)\}AA\ \}$. Using the above formula $Pr(E_1) = 0.64 + 0.04 + 0.64 = 1.32$, which is definitely incorrect. To understand the scenario better, let's replace each substring $W \in q(R,x)$ by a list of pairs $(w_j, p(w_j))$, where w_j is an instance of W with probability $p(w_j)$. Note that it is only a different way of representing set $q(R,x)$ and both representations are equivalent. $q(R,1) = \{(AAA, 0.8), (ACA,\ 0.2), (AAA, 0.8), (CAA, 0.2)\}$ and $Pr(E_1) = \sum_{w \in q(R,x)} p(w) \times Pr(w = S^x) = 1.32$ as before. However, this representation reveals that we have violated the second observation which requires matching of two substrings $w_i, w_j \in q(R,x)$ with S^x to be independent events. In the current example, both occurrences of a substring AAA in $q(R,1)$ belong to same possible world and effectively its probability contributes twice to $Pr(E_1)$.

We overcome this issue by obtaining an equivalent set $q(r,x)$ of $q(R,x)$ that satisfies the substring uniqueness requirement i.e., $w_i \neq w_j$ for all $w_i, w_j \in q(r,x)$ with $i \neq j$, and implicitly make the matching of two of its substrings with S^x independent events. To achieve this we pick up all distinct (deterministic) substrings $w \in q(R,x)$ (think of a representation of set $q(R,x)$ consisting of $(w_j, p(w_j))$ pairs) to be part of $q(r,x)$. To distinguish between these two sets, let $p_R(w_j)$ represent the probability associated with substring w_j in $q(R,x)$ and $p_r(w_j)$ be the same for $q(r,x)$. Then, we maintain the equivalence of sets by following the two step process described below for each $w \in q(r,x)$ and obtain the probability to be associated with it i.e., $p_r(w)$.

Step 1: Sort all occurrences of w in $q(R,x)$ by their start positions in R. Group together all occurrences that overlap with each other in R to obtain groups $g_1, g_2, ...$ Then no two occurrences across the groups overlap each other. Such a grouping is required only when there is a suffix-prefix match for w (i.e., some suffix of w represents same string as its prefix), otherwise all its overlapping occurrences represent different possible worlds of R and hence are in a single group by themselves. We assign the probability $p(g_i)$ to each group g_i as described below. Let ps_j represent the start position of occurrence w_j in R for $j = 1, 2, .., |g_i|$. The region of overlap between an occurrence w_j of w and its previous occurrences in R is given by range $[y,z] = [ps_j, ps_{j-1} + q - 1]$. We define $\beta_j = \beta_{j-1} + pr_R(w_j) - Pr(w_j[1..(z-y+1)] = R[y..z])$ with the initial condition $\beta_0 = 1$, $ps_0 = -1$. Then $p(g_i) = \beta_{|g_i|}$. In essence, we keep adding the probability of every occurrence while taking out the probability of its overlap.

Step 2: Assign $p_r(w) = 1 - \prod(1 - p(g_i))$.

The first step combines all overlapping occurrences into a single event and then we find out the probability that atleast one of these events takes place in second step. Now we can correctly compute the probability of event S^x matching substrings in $q(R,x)$ by using its equivalent set $q(r,x)$ as $\alpha_x = Pr(E_x) = \sum_{w \in q(r,x)} p_r(w) \times Pr(w = S^x)$. For the example under consideration, for a substring "AAA" we obtain a single group with its associated probability 0.8 using the process described above. Then $q(r,1) = \{(AAA, 0.8), (ACA, 0.2), (CAA, 0.2)\}$ and $Pr(E_1) = 0.68$ is correctly computed.

Computing upper bound for $Pr(ed(R,S) \leq k)$***:***
Finally, to obtain the upper bound on $Pr(ed(R,S) \leq k)$ we obtain set $\mathcal{C} \subseteq \Omega$ by pruning out those possible worlds which can not satisfy the edit-distance threshold k. Consider a possible world $pw_{i,j}$ in which s_j (r_i) is a possible instance of S (R). Both r_i and s_j being deterministic strings, we can quickly assess if r_i and s_j can be within edit distance k by applying the process of q-gram filtering described in Section 2.1. If affirmative, $pw_{i,j}$ is a candidate world and we include it in $\mathcal{C}$. However, our goal is to compute $Pr(\mathcal{C})$ without enumerating all possible worlds of $R \times S$. As before, we partition the possible worlds in Ω into sets $\Omega_0, \Omega_1, ..., \Omega_m$ such that $\Omega = \cup_{y=0}^{m} \Omega_y$ and $\Omega_y \cap \Omega_z = \emptyset$ for $y \neq z$. Moreover, Ω_y includes any possible world $pw_{i,j}$ where r_i contains substrings matching exactly y segments from $s_j^1, ..., s_j^m$ that partition s_j i.e, $y = |\{s_j^x | s_j^x \in q(r_i,x) \text{ for } x = 1, 2, .., m\}|$. Then $Pr(\mathcal{C}) = Pr(\cup_{y=(m-k)}^{m} \Omega_y) = \sum_{y=m-k}^{m} Pr(\Omega_y)$ and can be computed by following the same dynamic programming approach described earlier. Therefore the key difference in the current scenario (both R and S are uncertain) from the one in the previous subsection is the computation of α_x. After computing all α_x we can directly apply Lemma 2 and Theorem 1 and are rewritten as below. By slight abuse of notation as before, we say S^x matches R if $\alpha_x > 0$.

LEMMA 5. *Let R, S be the given strings with edit threshold k. If S is partitioned into $m = k+1$ disjoint segments, $Pr(ed(R,S) \leq k)$ is upper bounded by $(1 - \prod_{x=1}^{m}(1 - \alpha_x))$, where α_x gives the probability that segment S^x matches R.*

THEOREM 2. *Let R, S be the given strings with edit threshold k. Also assume S is partitioned into $m > k$ disjoint segments and α_x represents the probability that segment S^x matches R. Then $Pr(ed(R,S) \leq k)$ is upper bounded by the probability that at-least $(m-k)$ segments of S match R or, in another words the probability that R contains substrings matching at-least $(m-k)$ segments of S.*

It is evident that the cost of computing the upper bound in the above theorem is dominated by the set $q(r,x)$ computations. If this is assumed to be part of the preprocessing then the overhead involved is exactly the same as in the previous subsection. Let the fraction of uncertain characters in the strings be θ, and the average number of alternatives of an uncertain character be γ. For analysis of q-gram filtering, we assume uncertain character positions to be uniformly distributed from now onwards. Then $|q(r,x)| = (k+1)\gamma^{\theta \cdot q}$, and computing set $q(r,x)$ for each segment takes $q\gamma^{\theta \cdot q}$ times when string R is deterministic (previous subsection). Note that the multiplicative q appears only when substring w has a suffix-prefix match and its occurrences in set $q(R,x)$ overlap. Assuming typical values $\theta = 20\%$, $\gamma = 5$ and $q = 3$, it takes only two and half times longer to compute α_x when R is uncertain using $q(r,x)$.

4. INDEXING

Using Theorem 2 we observe that if a string R does not have substrings that match a sufficient number of segments of S, we can prune the pair (R,S). We use an indexing technique that facilitates the implementation of this feature to prune large numbers of dissimilar pairs. So far we assumed each string S is partitioned into m segments, each of which is of length q. In practice, we fix q as a system parameter and then divide S into as many disjoint segments as necessary i.e. $m = max(k+1, \lfloor |S|/q \rfloor)$. Without loss of generality let $m = \lfloor |S|/q \rfloor$. We use an even-partition scheme [14, 15] so that each segment has a length of q or $q+1$. Thus we partition S such that the last $|S| - \lfloor |S|/q \rfloor * q$ segments have length $q+1$ and length is q for the rest of them.

Let $\mathcal{S}_l$ denote the set of strings with length l and $\mathcal{S}_l^x$ denote the set of the x-th segments of strings in S_l. We build an inverted index for each $\mathcal{S}_l^x$ denoted by $\mathcal{L}_l^x$ as follows. Consider a string $S_i \in \mathcal{S}_l$. We instantiate all possibilities of its segment S_i^x and add them to $\mathcal{L}_l^x$ along with their probabilities. Thus $\mathcal{L}_l^x$ is a list of deterministic strings and for each string w, its inverted list $\mathcal{L}_l^x(w)$ is the set of uncertain strings whose x-th segment matches w tagged with probability of such a match. To be precise, $\mathcal{L}_l^x(w)$ is enumeration of pairs $(i, Pr(w = S_i^x))$ where i is the string-id. By design, each such inverted list $\mathcal{L}_l^x(w)$ is sorted by string-ids as described later. We emphasize that a string-id i appears at most once in any $\mathcal{L}_l^x(w)$ and in as many lists $\mathcal{L}_l^x(w)$ as the number of possible instances of S_i^x. We use these inverted indices to answer the similarity join query as follows.

We sort strings based on their lengths in ascending order and visit them in the same order. Consider the current

string $R = S_i$. We find strings similar to R among the visited strings only using the inverted indices. This implies we maintain indices only for visited strings to avoid enumerating a string pair twice. It is clear that we need to look for similar strings in $\mathcal{S}_l$ by querying its associated index only if $|R| - k \leq l \leq |R|$. To find strings similar to R, we first obtain candidate strings using the proposed indexing as described in next paragraph. We then subject these candidate pairs to frequency distance filtering (Section 5). Candidate pairs that survive both these steps are evaluated with CDF bounds (Section 6.1) with the final verification step (Section 6.2) outputting only the strings that are similar to R. After finding similar strings for $R = S_i$, we partition S_i into $m > k$ (as dictated by q) segments and insert the segments into appropriate inverted index. Then we move on to the next string $R = S_{i+1}$ and iteratively find all similar pairs.

Finally, given a string R, we show how to query the index associated with $\mathcal{S}_l$ to find candidate pairs (R, S) such that $S \in \mathcal{S}_l$ and $Pr(ed(R, S) \leq k) > \tau$. We preprocess R to obtain $q(r, x)$ that can be used to query each inverted index $\mathcal{L}_l^x$. For each $w \in q(r, x)$ we obtain an inverted list $\mathcal{L}_l^x(w)$. Since all lists are sorted by string-id, we can scan them in parallel to produce a merged (union) list of all string-ids i along with the α_x computed for each of them. We maintain a *top* pointer in each $\mathcal{L}_l^x(w)$ list that initially points to its first element. At each step, we find out the minimum string-id i among the elements currently at the top of each list, compute α_x for a pair (R, S_i) using the probabilities associated with string-id i in all $\mathcal{L}_l^x(w)$ lists (if present). After outputting the string-id and its α_x as a pair in the merged list, we increment the *top* pointers for those $\mathcal{L}_l^x(w)$ lists which have the *top* currently pointing to the element with string-id i. Let the merged list be $\mathcal{L}\alpha_x$. Once again all $\mathcal{L}\alpha_x$ lists for $x = 1, 2, .., m$ are sorted by string-ids. Therefore by employing *top* pointers and scanning lists $\mathcal{L}\alpha_x$ in parallel, we can count the number of segments in S_i that matched with their respective $q(r, x)$ by counting the number of $\mathcal{L}\alpha_x$ lists that contain string-id i. If the count is less than $m-k$ we can safely prune out candidate pair (R, S_i) using Lemma 5. Otherwise, we can compute the upper bound on $Pr(ed(R, S_i) \leq k)$ by supplying the α_x values already computed to the dynamic programming algorithm. If the upper bound does not meet our probability threshold requirement, we can discard string S_i as it can not be similar to R by Theorem 2, otherwise (R, S_i) is a candidate pair.

Given a string R, the proposed indexing scheme allows us to obtain all strings $S \in \mathcal{S}$ that are likely to be similar to R without explicitly comparing R to each and every string in $\mathcal{S}$ as has been done for related problems in the area of uncertain strings [10, 6, 4]. For a string r in a deterministic strings collection, we need to consider $m(k + 1)$ of its substrings while answering the join query using the procedure just described. In comparison, in the probabilistic setting we need to consider $m(k+1)\gamma^{\theta \cdot q}$ deterministic substrings of R. Moreover, a string-id can belong to at most $\gamma^{\theta \cdot q}$ inverted lists in $\mathcal{L}_l^x$ in probabilistic setting whereas inverted lists are disjoint for deterministic strings collection. Thus, the proposed indexing achieves competitive performance against its counterpart for answering a join query over deterministic strings. Further, indexing scheme uses disjoint q-grams of strings instead of overlapping ones as in [6, 4]. This allows us to use slightly larger q with same storage requirements.

5. FREQUENCY DISTANCE FILTERING

As noted in [4], frequency distance displays great variation with increase in the number of uncertain positions in a string and can be effective to prune out dissimilar string pairs. We first obtain a simple lower bound on $fd(R, S)$ and then show how to quickly compute the upper bound for the same. For each character $c_i \in \Sigma$, let $f(S)_i^c$, $f(S)_i^t$ denote the minimum and maximum possible number of its occurrences in S respectively. For brevity, we drop the function notations and denote these occurrences as fS_i^c and fS_i^t. Note that fS_i^c also represents the number of occurrences of c_i in S with probability 1 and fS_i^t represents the number of certain and uncertain positions of c_i. Thus $fS_i^u = fS_i^t - fS_i^c$ gives the uncertain positions of c_i in S. fR_i^c, fR_i^u and fR_i^t are defined similarly. We observe that, if $fR_i^t < fS_i^c$, any possible world pw of $R \times S$, will have a frequency distance at least $(fS_i^c - fR_i^t)$. By generalizing this observation, we can obtain a lower bound on $fd(R, S)$ as summarized below.

LEMMA 6. *Let R and S be two strings from the same alphabet Σ, the we have $fd(R, S) \geq max\{pD, nD\}$, where*

$$pD = \sum_{fS_i^t < fR_i^c} (fR_i^c - fS_i^t),\ nD = \sum_{fR_i^t < fS_i^c} (fS_i^c - fR_i^t)$$

Since the edit distance of a string pair is lower bounded by its frequency distance, we can prune out (R, S) if the minimum frequency distance obtained by above the lemma is more than the desired edit threshold k. To obtain the upper bound on the probability of $fd(R, S)$ being at most k, we use the technique introduced in [4] that relies on the expected value of all possible frequency distances. Using such an expectation for positive and negative frequency distance ($E[pD]$, $E[nD]$), One-Sided Chebyshev Inequality and following the same analysis in [4], we obtain following theorem.

THEOREM 3. *Let R and S be two strings from the same alphabet Σ. Then we have,*

$$Pr(ed(R, S) \leq k) \leq Pr(fd(R, S) \leq k) \leq \frac{B^2}{B^2 + (A - k)^2}$$

$$\textit{where, } A = \frac{||R| - |S||}{2} + \frac{E[pD] + E[nD]}{2}$$

$$B^2 = \frac{(|R| - |S|)^2}{2} + \frac{(||R| - |S||)(E[pD] + E[nD])}{2} + min(|R| \cdot E[nD], |S| \cdot E[pD]) - A^2$$

The main obstacle in using the above theorem is efficient computation of $E[pD] = \sum_{c_i} E(fR_i - fS_i)$, $E[nD] = \sum_{c_i} E(fS_i - fR_i)$. We focus on computing $E[nD]$ below as $E[pD]$ can be computed in a similar fashion. With frequency of c_i in S i.e. fS_i varying between fS_i^c and fS_i^t, let $Pr(fS_i = x)$ represents the probability that c_i appears exactly x times. Putting it an other way, $Pr(fS_i = x)$ represents the probability that c_i appears at exactly $(x - fS_i^c)$ uncertain positions from (fS_i^u) uncertain positions overall. This leads to a natural dynamic programming algorithm that can compute $Pr(fS_i = x)$ for all $x = fS_i^c, ..., fS_i^t$ by spending $O((fS_i^u)^2)$ time. Please refer to [4] more details. With the goal of efficiency in computing $E[nD]$, authors preprocess S and maintain these values in $O(fS_i^u)$ space. Without loss of generality, let $fR_i^c < fS_i^c \leq fR_i^t < fS_i^t$. Then by definition, $E[nD] = \sum_{c_i} E[nD_i]$ where,

$$E[nD_i] = \sum_{x=fR_i^c}^{fR_i^t} \sum_{\substack{y=max \\ (x+1, fS_i^c)}}^{fS_i^t} Pr(fR_i = x)Pr(fS_i = y)(y - x)$$

In the above equation, $Pr(fR_i = x)$ and $Pr(fS_i = y)$ can be computed in constant time using precomputed answers. Therefore, a naive way of computing $E[nD_i]$ will take $O(fS_i^u fR_i^u)$. Below we speed up this computation and achieve $min(fS_i^u, fR_i^u)$ time. We maintain the following probability distributions for each c_i of S. For $0 \leq x \leq fS_i^u$,

$$S1_i[x] = Pr(fS_i = fS_i^c + x)$$

$$S2_i[x] = \sum_{y=x}^{fS_i^u} Pr(fS_i = fS_i^c + y)$$

$$S3_i[x] = \sum_{y=x}^{fS_i^u} (y - x + 1)Pr(fS_i = fS_i^c + y)$$

$$S4_i[x] = \sum_{y=0}^{x} (x - y)Pr(fS_i = fS_i^c + y)$$

$S1_i$ is simply a probability distribution of c_i appearing at uncertain positions in range $[0, fS_i^u]$ (precomputed using dynamic programming). $S2_i$ maintains the probability that c_i appears at at-least x uncertain positions i.e. $S2_i[x] = Pr(fS_i \geq fS_i^c + x)$. $S3_i$ maintains the same summation with elements in the summation series scaled by $1, 2, ...$ Finally $S4_i$ takes the summation series for $Pr(fS_i \leq fS_i^c + x)$, scales it by $0, 1, ...$ in reverse direction and maintains the output at index x. The intuition behind maintaining the scaled summations is that, given a particular frequency z of fR_i, the expectation of its frequency distance with $fS_i \in [fS_i^c, fS_i^t]$ resembles the summation series for $S3_i[x]$ or $S4_i[x]$. All the above distributions can be computed in $O(fS_i^u)$ time and occupy the same $O(fS_i^u)$ storage. Similar probability distributions are also maintained for R. We achieve the speed up without hurting preprocessing time and at no additional storage cost. $E[nD_i]$ can now be computed as follows:

$$E[nD_i] = \sum_{x=fR_i^c}^{fS_i^c - 1} \sum_{y=fS_i^c}^{fS_i^t} (...) + \sum_{x=fS_i^c}^{fR_i^t} \sum_{y=x+1}^{fS_i^t} (...)$$
$$= nD_i^1 + nD_i^2$$

$$nD_i^1 = \sum_{x=fR_i^c}^{fS_i^c - 1} Pr(fR_i = x)(\sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y)(y - x))$$
$$= \sum_{x=fR_i^c}^{fS_i^c - 1} Pr(fR_i = x)(fS_i^c - x - 1) \sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y)$$
$$+ \sum_{x=fR_i^c}^{fS_i^c - 1} Pr(fR_i = x) \sum_{y=fS_i^c}^{fS_i^t} Pr(fS_i = y)(y - fS_i^c + 1)$$
$$= R4_i[fS_i^c - fR_i^c - 1] \times S2_i[0]$$
$$+ (R2_i[0] - R2_i[fS_i^c - fR_i^c]) \times S3_i[0]$$

$$nD_i^2 = \sum_{x=fS_i^c}^{fR_i^t} Pr(fR_i = x) \sum_{y=x+1}^{fS_i^t} Pr(fS_i = y)(y - x)$$
$$= \sum_{x=fS_i^c}^{fS_i^t} R1_i[x - fR_i^c] \times S2_i[x - fS_i^c + 1]$$

If the fraction of uncertain characters in the strings is θ, frequency filtering summarized in Theorem 3 can be applied in $O(\sigma\theta(|R| + |S|))$. Typical alphabet size being constant, the efficiency of applying frequency filtering depends on the degree of uncertainty and string lengths. Therefore, with increase in length of input strings, improvement from $|R| \times |S|$ to $|R| + |S|$ provides substantial reduction in the filtering time. While answering the similarity join query, we preprocess $R = S_i \in \mathcal{S}_l$ to compute the arrays for each character in alphabet Σ and maintain them as a part of our index. All candidate pairs passing the q-gram filtering are then subjected to frequency distance filtering for further refinement before moving onto next string $R = S_{i+1} \in \mathcal{S}_l$.

6. VERIFICATION

The goal of verification is to conclude whether strings in the candidate pair (R, S) that has survived the above filters, are indeed similar i.e., $Pr(ed(R, S) \leq k) > \tau$. A straightforward solution is to instantiate each possible world of $R \times S$ and add up the probabilities of possible worlds where possible instances of R, S are within edit threshold k. Before resorting to such expensive verification, we make a last attempt to prune out a candidate pair, by extending the CDF bounds in [6]. If unsuccessful, we use the trie-based verification that exploits common prefixes shared by instances of an uncertain string.

6.1 Bound based on CDF

We briefly review the process in [6] and highlight the changes needed to compute the mentioned bounds correctly when both input strings are uncertain. We populate the matrix $|R| \times |S|$ using dynamic programming. In each cell $D = (x, y)$, we compute (at most) $k + 1$ pairs of values i.e., $\{(L[j], U[j]) | 0 \leq j \leq k\}$, where $L[j]$ and $U[j]$ are the lower and upper bounds of $Pr(ed(R[1..x], S[1..y]) \leq j)$ respectively. Then by checking the bounds in the cell $(|R|, |S|)$, we can accept or reject the candidate string pair (R, S), if possible. To fill in the DP table, consider a basic step of computing bounds of a cell $D = (x, y)$ from its neighboring cells - upper left: $D1 = (x-1, y-1)$, upper: $D2 = (x, y-1)$, and left: $D3 = (x-1, y)$. As noted in [6], when the $R[x]$ matches $S[y]$ (with probability $p_1 = \sum_{c_i} Pr(R[x] = c_i)Pr(S[y] = c_i)$), it is always optimal to take the distribution from the diagonal upper left neighbor. When $R[x]$ does not match $S[y]$ with probability $p_2 = 1 - p_1$, we use the relaxations suggested in [6]. Let $(argmin D_i)$ returns index i $(1 \leq i \leq 3)$ such that $L_{D_i}[0]$ is greatest; a tie is broken by selecting the greatest $L_{D_i}[1]$ and so on.

THEOREM 4. *At each cell $D = (x, y)$ of the DP table, $L[j] \leq Pr(ed(R[1..x], S[1..y]) \leq j) \leq U[j]$, where*

$$L[j] = max(p_1 L_{D_1}[j],\ p_2 L_{(argmin\ D_i)}[j-1])$$

$$U[j] = min(1,\ p_1 U_{D_1}[j] + p_2 U_{D_1}[j-1] + \sum_{i=2}^{3} U_{D_i}[j-1])$$

PROOF. We follow the analysis in [6] as follows. Consider a possible world $pw_{i,j}$ in which $r_i[x] = s_j[y]$. Let the distance values at cells D and D_i $(1 \leq i \leq 3)$ be v and v_i, respectively. Then we have $v = v_1$. This is because $v_2, v_3 \geq v_1 - 1$; thus, $v = min(v_1, v_2 + 1, v_3 + 1) = v_1$. Next, consider a possible world $pw_{i,j}$ in which $r_i[x] \neq s_j[y]$. Then, $v = min(v_i) + 1$. By using $(argmin\ D_i)$, we pick one fixed

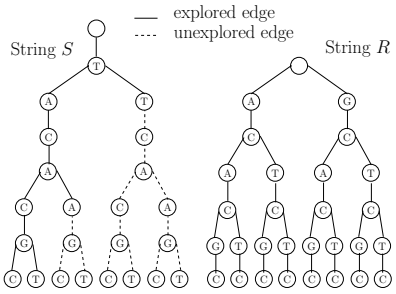


Figure 1: Trie-based Verification Example

neighbor cell (i.e., the one that has a small distance value with the highest probability) instead of accounting for all possible worlds in which $r_i[x] \neq s_j[y]$; and hence the true v value could be smaller than this one in some possible worlds. However, we observe that out of all possible worlds with distance v in the cell D, worlds with edit distance v in D_1 are not disjoint with worlds with distance $v-1$ for D_2. The same argument applies for worlds with $v-1$ as distance in D_3 as well. Therefore, we choose the maximum out of the two scenarios as our lower bound. For obtaining the upper bound, the case where $r_i[x]$ matches $s_j[y]$ remains the same. Possible world $pw_{i,j}$ with distance $v-1$ for D_2, can be extended by reading an addition character of R and we get distance v in cell D for all of them. Similarly, moving from distance $v-1$ in D_3 to distance v in D can be thought to be the case of inserting a character of S. Hence, we do not need to scale down the probability $U_{D_2}[v-1]$ as well $U_{D_3}[v-1]$ to obtain the upper bound for cell D. □

We note that the bounds summarized in the above theorem are different than the ones presented in [6], as they cannot be used directly for the current scenario[1]. Finally, the simple DP algorithm can be improved by computing $(L[j], U[j])$ only for those cells $D = (x, y)$ for which $|x-y| \leq k$, since $L[k] = U[k] = 0$ otherwise. Thus, we can apply the CDF bounds based filtering for a candidate pair (R, S) in $O(min(|R|, |S|)(k+1)max(k, \gamma))$ where γ is average number of alternatives of an uncertain character.

6.2 Trie-based Verification

Prefix-pruning has been a popular technique to expedite verification of a deterministic string pair (r, s) for edit threshold k. A naive approach for this verification would be to compute the dynamic programming matrix (DP) of size $|r| \times |s|$ such that cell (x, y) gives the edit distance between $r[1...x]$ and $s[1...y]$. Prefix-pruning observes that if all cells in row x i.e., $(x, *)$ do not meet threshold k, then the following rows can not have cells with edit distance k or less i.e., $DP[i > x, *] > k$. Even using such an early termination condition, verifying all-pairs (all possible of instance of $R \times S$) for a candidate pair (R, S) can be expensive. With the goal of avoiding naive all-pairs comparison, we propose trie-based verification. Let $\mathcal{T}_S$ be the trie of all possible instances of S and $\mathcal{T}_R$ be the same for string R. Let node u in $\mathcal{T}_S$ represents a string u (obtained by concatenating the edge labels from root to node u), then all possible instances of S with u as a prefix are leaves in the subtree rooted at u. We say a node $u \in \mathcal{T}_S$ is similar to node $v \in \mathcal{T}_R$ if $ed(u, v) \leq k$. Using prefix-pruning then we have following observation [5]:

- Given $u \in \mathcal{T}_S$, $v \in \mathcal{T}_R$: if u is not similar to any ancestor of v, and v is not similar to any ancestor of u, any possible instance s of S with prefix u can not be similar to a possible instance r of R with v as its prefix.

Using the technique in [11, 5], we can compute a set of similar nodes in $\mathcal{T}_R$ for each node $u \in \mathcal{T}_S$. Then, if $u = s_j$ is a leaf node, each node $v = r_i \in \mathcal{T}_R$ in its similar set that is also a leaf node, gives us a possible world $pw_{i,j}$ whose probability contributes to $Pr(ed(R, S) \leq k)$. However techniques in [5] implicitly assume both trie structures are available. Here we propose on-demand construction of trie which avoids all possible instances of S to be enumerated. Note that we still need to build the trie $\mathcal{T}_R$ completely. However its construction cost can be amortized as we build $\mathcal{T}_R$ once and use it for all candidate pairs $(R, *)$. As noted in [11], nodes in $\mathcal{T}_R$ that are similar to node $u \in \mathcal{T}_S$ can be computed efficiently only using such a similarity set already computed for its parent. This allows us to perform a (logical) depth first search on $\mathcal{T}_S$ and materialize the children of $u \in \mathcal{T}_S$ only if its similarity set is not empty. Figure 1 illustrates of this approach and reveals that on-demand trie construction can reduce the verification cost by avoiding instantiation and consequently comparison with a large fraction of possible worlds of S. In the figure, only the nodes linked with solid lines are explored and instantiated by the verification algorithm. Moreover, we do not display the similar node sets and the probabilities associated with trie nodes for simplicity.

7. EXPERIMENTS

We have implemented the proposed indexing scheme and filtering techniques in C++. The experiments are performed on a 64 bit machine with an Intel Core i5 CPU 3.33GHz processor and 8GB RAM running Ubuntu. We consider the following algorithms for comparisons which use only a subset of the filtering mechanisms. Algorithm QFCT makes use of all the filtering schemes listed in this article whereas QCT, QFT, FCT bypass frequency-distance filtering, filtering based on CDF bounds and q-gram filtering respectively.

Datasets: We use two synthetic datasets obtained from their real counterparts employing the technique used in [10, 4]. The first data source is the author names in dblp ($|\Sigma| = 27$). For each string s in the dblp dataset we first obtain a set $A(s)$ of strings that are within edit distance 4 to s. Then a character-level probabilistic string S for string s is generated such that, for a position i, the pdf of $S[i]$ is based on the normalized frequencies of the letters in the i-th position of all the strings in $A(s)$. The fraction of uncertain positions in a character-level probabilistic string i.e., θ is varied between 0.1 to 0.4 to generate strings with different degree of uncertainty. The string length distributions in this dataset follow approximately a normal distribution in the range of $[10, 35]$. For the second dataset we use a concatenated protein sequence of mouse and human ($|\Sigma| = 22$), and break it arbitrarily into shorter strings. Then uncertain strings are obtained by following the same procedure as that for the dblp data source. However, for this dataset we use slightly larger string lengths with less uncertainty i.e. string lengths roughly follow uniform distribution in the range $[20, 45]$ and θ ranges between 0.05 to 0.2. In both datasets, the average number of choices (γ) that each probabilistic character $S[i]$ may have is set to 5. The default values used for the dblp dataset are: the number of strings in collection $|\mathcal{S}| = 100$K,

[1]a) Lower bound violation: $r = ACC$, $S = A\{(C, 0.7), (G, 0.1), (T, 0.1)\}$ with $k = 1$. b) Upper bound violation: $r = DISC$, $S = DI\{(C, 0.4), (S, 0.5), (R = 0.1)\}$ with $k = 1$.

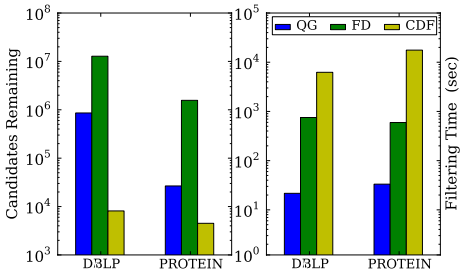


Figure 2: Effectiveness vs. Efficiency

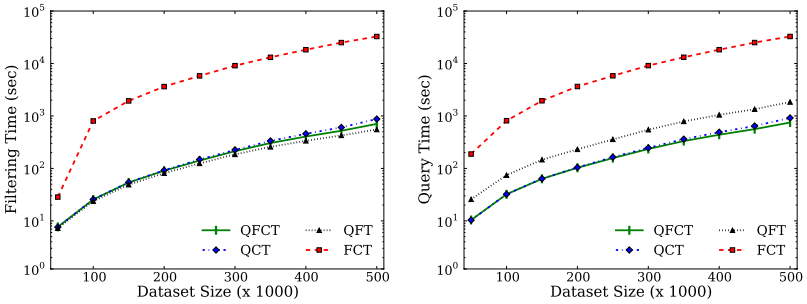


Figure 3: Effect of Dataset Size $|\mathcal{S}|$

average string length ≈ 19, $\theta = 0.2$, $k = 2$, $\tau = 0.1$, and $q = 3$. Similarly for protein dataset we use default setting with $|\mathcal{S}| = 100$K, average string length $= 32$, $\theta = 0.1$, $k = 4$, $\tau = 0.01$, and $q = 3$.

7.1 Effectiveness vs. Efficiency of Pruning

In this set of experiments, we compare the pruning ability of the filtering techniques and the overhead of applying them on both the datasets with $\theta = 0.2$, $k = 2$ and $\tau = 0.1$. Figure 2 shows the number of candidates remaining after applying each filtering scheme and reveals that CDF bounds provide the tightest filtering among the three. Effectiveness of the CDF follows from the fact that it uses upper as well as lower bounds to prune the strings. The upper bound obtained by q-gram filtering tends to be looser than the CDF as it depends on the partitioning based on q, whereas frequency distance based upper bound is sensitive to the length difference between two strings. However, the effectiveness of CDF comes at the cost of time. On the other hand, q-gram filtering is extremely fast and can still prune out a significant number of candidate pairs taking advantage of the indexing scheme. For the protein dataset, q-gram is close to CDF bounds in terms of effectiveness and is an order of magnitude faster than computing CDF bounds. Frequency distance filtering being dependent only on alphabet size and uncertain positions in the strings (against CDF's dependance on string length) can help to improve query performance by reducing the number of candidate pairs passed on to CDF for evaluation. Therefore, in the following experiments, algorithm variants use these filtering techniques in the increasing order of their overhead as suggested by their acronyms.

Figure 2 also reveals that applying q-gram filtering and the CDF bounds filtering takes longer for the protein dataset than for dblp data. Due to larger string length and fixed q, q-gram filtering needs to partition protein strings into a larger number of segments (i.e., m). Thus, there are more α_x probabilities to be computed and it takes longer to compute the desired upper bound in Theorem 2. Similarly, computing CDF bounds needs to populate a dynamic programming matrix whose size depends on the string lengths. However, frequency distance filtering benefits from smaller alphabet set and lower degree of uncertainty in protein sequences and shows better performance for the protein data.

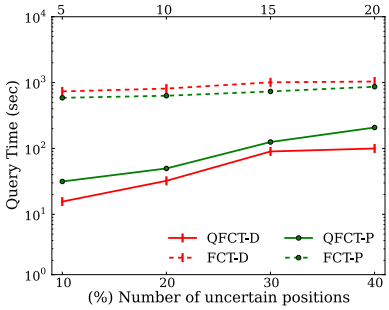
7.2 Effects of Data Size $|\mathcal{S}|$

Figure 3 shows the scalability of various algorithms on the dblp dataset, where we vary $|\mathcal{S}|$ from 50K to 500K. With computationally inexpensive q-gram filtering as the first step, algorithms QFCT, QFT and QCT achieve efficient filtering even for the larger datasets. For the exceptional case of the algorithm FCT, the filtering overhead increases almost quadratically with increase in the input size as both filtering techniques (frequency distance and CDF bounds) need to explicitly compare the query string R with all possible strings $S \in \mathcal{S}$ ($|S| \geq |R| - k$). Also, the filtering time required for QFT and QCT closely follows that for QFCT. This confirms the ability of q-gram filtering to significantly reduce the filtering overhead, and highlights the advantages offered by the proposed indexing scheme incorporating it.

Figure 3 also shows the time required for answering the join query for these algorithms. FCT, lacking efficient filtering (though effective), takes the longest to output its answers. However, the query time for QFT, despite using efficient q-gram filtering, shows a rapid increase. In contrast to this, the good scalable behavior of QFCT and QCT emphasizes the need for using tight filtering conditions based on the lower and upper bounds of CDF. In the absence of these, exponentially more number of candidates need trie-based verification which results in quickly deteriorating query performance. Thus, a combination of q-gram filtering with CDF bounds in QFCT achieves the best of both worlds, allowing us to restrict the increase in both filtering time as well as the number of trie-based verifications. Though the number of outputs increased quadratically with data size, the increase in the number of false positives in the verification step of QFCT (i.e., the scenario where a candidate pair was not an output after verification) was found to be linear to the output size. An order of magnitude performance gain of QFCT over others seen in Figure 3 will be further extended for larger input collections. With algorithm QFT requiring a higher number of expensive verifications and QCT showing similar trends as that of QFCT, we use only the remaining two algorithms i.e., QFCT and FCT for the experiments to follow. We also append a character 'D' or 'P' to the algorithms acronym to distinguish between its query times on the dblp and protein datasets.

7.3 Effects of θ

An increase in the number of uncertain positions in the string has a detrimental effect on both algorithms QFCT and FCT as shown in Figure 4. This is due to the direct impact of θ on every step of the algorithm in answering join queries. Starting with the q-gram filtering, more uncertain positions for query string R imply more time required for populating the sets $q(r, x)$ as a preprocessing step, as well as for adding the string R to inverted indices after answering the query. Also the larger size of set $q(r, x)$ due to the increase in θ increases look up time in inverted indices and consequently



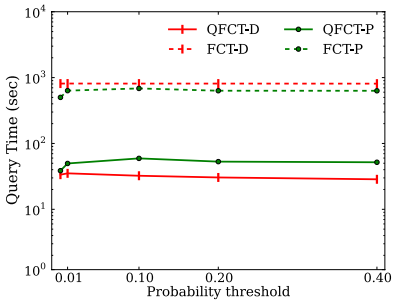


Figure 4: Effect of θ

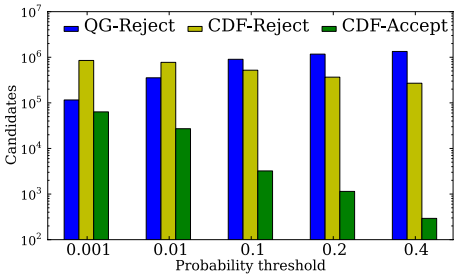


Figure 5: Effect of τ

increase the time required for computing α_x. Though size of a set $q(r, x)$ can increase exponentially with θ, its impact is limited due to the small fixed value of q. There is another subtle impact of θ on q-gram filtering. With more uncertain positions in query string R, more strings in the collection can be matched with substrings of R. We found this increase to be linear with $\approx 1.5\%$ of all join pairs evaluated by q-gram filtering for $\theta = 0.1$ to only $\approx 4\%$ evaluated for $\theta = 0.4$ on the dblp dataset. Thus, proposed q-gram filtering serves the purpose of efficient pruning even with the increased uncertainty.

The impact of θ on the computation of frequency distance and CDF bounds is more obvious. Computing the expected frequency distance of a character directly depends on the number of positions in input strings (R, S) where it appears probabilistically. Due to the probability computation of two positions matching in R and S ($R[x] = S[y]$), it takes longer to populate a dynamic programming matrix for CDF. Thus, the increase in filtering time of query algorithms is almost linear to θ. Finally, in the trie-based verification, more possible words need to be evaluated, increasing verification cost exponentially. In conclusion, the verification step is the worst affected among all due to large θ and is the primary contributor in increased time for answering join queries. We note that in most of the scenarios, algorithm QFCT takes longer to answer join queries for the protein data than for the dblp data because of the higher overhead of q-gram and CDF filtering, which we pointed out in Section 7.1. On the other hand, algorithm FCT performs better for the protein data by virtue of faster frequency filtering as seen earlier. This comparative behavior of QFCT and FCT is also evident in Figure 4.

7.4 Effects of τ

Figure 5 shows the results on the dblp and protein dataset for different values of τ from 0.001 to 0.4. Though the query times remain insensitive to τ for a large range, a gradual increase or decrease in probability threshold has a two fold effect on query algorithms. We analyze the scenario by looking at the number of candidate pairs pruned by CDF bounds either by accepting based on lower bound or rejecting based on upper bound. As τ increases, upper bound filter becomes more and more selective as it can reject more number of candidate pairs. On the contrary, filtering based on lower bound looses its effectiveness with increased τ as it can not accept as many strings as it can for smaller values of τ. Thus the relative increase and decrease in number of candidate pairs pruned by CDF upper and lower bound respectively determines the overall effect of varying τ. When upper bound filter can not compensate for the loss in effectiveness of lower bound, more candidate pairs require trie-based verification resulting in higher query time. Such a scenario is evident in Figure 5 for protein data for τ ranging from 0.001 to 0.1.

τ has an interesting effect on q-gram filtering. Figure 5 shows the number of candidates pairs rejected by q-gram filtering in QFCT. It also shows the count of accepted candidates using CDF lower bound and rejected by CDF upper bound in QFCT. Note that q-gram filtering only uses the upper bound and Figure 5 shows the reduced effectiveness of CDF lower bound filtering. As τ increases, probabilistic pruning (Theorem 2) becomes more effective and prunes out a significant number of candidate pairs that satisfy the necessary condition for two strings to be similar as described in Lemma 5 (shown in Figure 5). In effect, q-gram filtering reduces the overhead of applying CDF bounds and to some extent compensates for the increased verification cost, if any. This effect can be seen by gradual decrease in the number of candidates rejected by CDF even though an increased number of candidates are pruned using the upper bound overall. Finally, for large τ, the q-gram filtering advantage coupled with reduced output size due to more selective τ results in improved query time.

7.5 Effects of k

Figure 6 shows the time required for answering a join query on the dblp dataset when k changes from 1 to 4 and for the protein dataset with $k = 2, 4, 6, 8$. With increased k we can expect more string pairs to satisfy an edit threshold and hence an increase in query time. As we loosen the edit threshold requirement, the effectiveness of q-gram filtering begins to deteriorate since the requirement for Lemma 5 can be met with string S having less number of its partitions being matched with substrings in R. Therefore, even with probabilistic pruning, many false candidates pairs are passed on to frequency distance and CDF filtering routines. Also, the number of candidates removed by the upper bound of frequency distance and CDF decreases with an increase in k. Though lower bound filtering in CDF can accept more candidates with an increase in k, this benefit is easily offset by loose upper bounds resulting in net increase in verification cost. With increased k, the time required for algorithm QFCT approaches that of FCT but still manages to save up to 35% of FCT's query cost.

7.6 Effects of q

In this set of experiments we try to investigate the effect of q-gram length on the efficiency and effectiveness of q-gram filtering using input collections with 100K strings. As pointed out earlier, q-gram filtering incurs more filtering overhead for higher string lengths with fixed q. We can hope

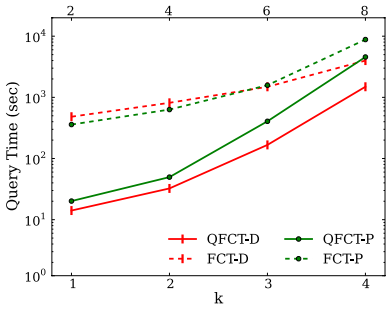
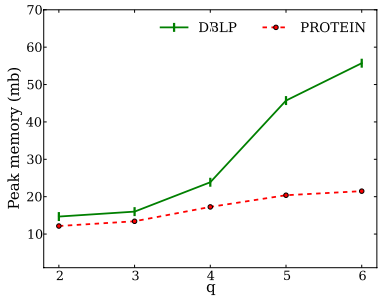


Figure 6: Effect of k



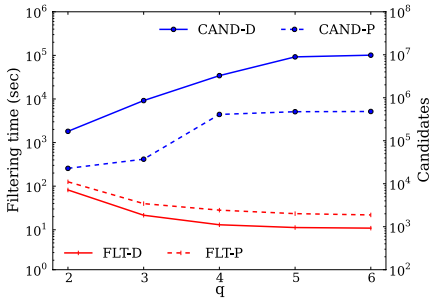


Figure 7: Effect of q

to reduce this overhead by increasing q, however such an increase has side-effects on the space-time tradeoff of q-gram filtering. Even though we will have fewer partitions for each string due to increased q, each segment now has more possible instances to be added to the inverted indices increasing the storage requirement as shown in Figure 7. The rate of increase is faster for the dblp dataset because of higher θ i.e., more uncertain positions and larger alphabet set. We note that we use peak memory usage as a measure that accounts for the indices maintained at any point during query answering based on the length of a string currently under consideration. Further, this also implies that query preprocessing that populates sets $q(r, x)$ needs more time offsetting the benefits of higher q to some extent. Figure 7 shows the improvement in the filtering time for q varying from 2 to 6. With size of $q(r, x)$ increasing exponentially with q, the improvement in filtering time achieved due to fewer segments also decreases exponentially.

For deterministic strings, increasing q makes it difficult for a segment of string s to match with substrings of query string r and implies potential improvement in pruning ability of q-gram filtering. For uncertain strings though, due to higher q, a segment may contain a larger number of uncertain positions. Hence there are more number of possible instances with increased chances for a segment to find a match in substrings of a query string. As a result, the effectiveness of q-gram filtering diminishes gradually for higher q as seen in Figure 7. We note that, though filtering time improves with q, time required for answering a join query shows uni-valley behavior as less effective filtering causes increased query time for higher q even with less filtering overhead. We found $q = 3$ or $q = 4$ offers the best combination of fast effective pruning with acceptable storage requirement. With peak memory usage of inverted indices less than the input data size itself for both $q = 3$ and 4, the space required for storing all indices as required for answering similarity search queries was found to be only ≈ 1.5 and ≈ 2 times the data size respectively.

7.7 Evaluating Trie-based Verification

We now analyze the performance benefits offered by the trie-based verification over a naive way of doing the same. Figure 8 shows the verification time required for answering join queries on the dblp and protein datasets with varying degree of uncertainty i.e., parameter θ. With an increase in the number of uncertain positions in the string, the number of possible worlds increases exponentially. This results in increased verification cost for both trie-based and naive verification. In naive verification, we need to enumerate possible worlds for each string in the dataset and also enumerate possible words for each of the candidate strings that may form a similar pair with it. In effect, we may enumerate all possible worlds for each string more than once. Additionally, given a candidate pair (R, S) it needs to compare every possible instance of R with that of S. In contrast, the trie-based verification enumerates all possible words for each string S only once and when it is selected as a candidate for some other string R in database, it enumerates and compares only those possible worlds which are highly likely to be similar to some instance of R. Thus the performance gains of trie-based verification increase with increasing θ as seen in Figure 8. We note that the cost of verification using trie-based approach also increases exponentially due to the requirement of having a complete trie in place for query string R. Moreover, trie-based verification can be more expensive than the naive method in scenarios where the majority of instances of $R \times S$ satisfy the edit threshold due to the overhead of building a trie and computing a set of similar nodes for each node in the trie. Though we obtained performance gains using trie-based verification on the protein data as well, they were less significant than for the dblp data due to higher string lengths, lower degree of uncertainty (θ) and smaller alphabet [illegible]

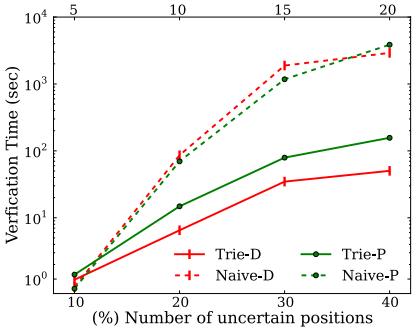


Figure 8: Trie-based Verification

7.8 Effects of String Length

In this final set of experiments, we test algorithms QFCT and FCT by varying the length of the probabilistic strings. For studying this effect, we use the 100K versions of the dblp and protein datasets, and append each probabilistic string to itself for 0,1,2 or 3 times. To ensure that the verification step does not get excessively expensive, we limit the number of probabilistic characters in a probabilistic string to be at most 8. Clearly, the costs of both algorithms increase with longer strings as seen in Figure 9. In terms of filtering time, computation of q-gram filtering and CDF bounds takes longer as string lengths increase, as described earlier in Section 7.1. However, frequency distance filtering being de-

pendent only on the number of uncertain character positions remains unaffected. This allows algorithm FCT to close the performance gap with QFCT for higher string lengths by virtue of efficient frequency distance filtering. Additionally, verification cost begins to dominate the query time with the increase in string lengths. We note that even trie-based verification needs to instantiate all possible worlds for each probabilistic string once while answering a join query. With each possible world enumeration taking more time, higher string length adversely affects the verification step. For fixed k, τ, and uncertain character positions, the number of output pairs decreases with increase in string length. Despite this, the query time increases because of the aforementioned reasons. We emphasize that the proposed filtering techniques maintain their effectiveness with varying lengths as the fraction of the candidate pairs that undergo verification and are accepted as output remains almost constant.

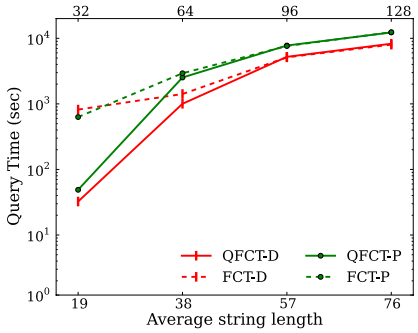


Figure 9: Effects of string length

7.9 Comparison with EED

In this subsection, we qualitatively compare the join query algorithm in [10] against algorithms presented in this work:

1. We partition each string in the collection based on q whereas q-gram filtering in [10] makes use of overlapping q-grams. This allows us to significantly reduce the space required for storing all q-grams ($\approx 5 \times datasize$ as reported in [10] against our index of twice the data size).
2. q-gram filtering presented in [10] requires each probabilistic string pair to be evaluated during query execution tasks like computation of frequency distance, CDF bounds computation. Algorithm QFCT employes indexing that incorporates q-gram filtering before applying expensive filters. Therefore, we can expect QFCT to offer benefits over the query algorithm in [10] similar to its advantages over algorithm FCT seen in Figure 3.
3. Computing the exact *eed* between two probabilistic strings requires all possible worlds for two strings to be instantiated in the same way as a naive verification method discussed in Section 7.7. On the other hand trie-based verification allows us to determine the similarity of a string pair efficiently (refer to Figure 8).

8. CONCLUSIONS

In this paper, we study the largely unexplored problem of answering similarity join queries on uncertain strings. We propose a novel q-gram filtering technique that integrates probabilistic pruning and extends frequency distance and CDF based filtering techniques. In future work, we plan to investigate tighter filtering conditions and improvements to the trie-based verification algorithm.

9. REFERENCES

[1] A. Amir, E. Chencinski, C. S. Iliopoulos, T. Kopelowitz, and H. Zhang. Property matching and weighted matching. In *CPM*, pages 188–199, 2006.

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

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

[4] D. Dai, J. Xie, H. Zhang, and J. Dong. Efficient range queries over uncertain strings. In *SSDBM*, pages 75–95, 2012.

[5] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[6] T. Ge and Z. Li. Approximate substring matching over uncertain strings. *PVLDB*, 4(11):772–782, 2011.

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

[8] C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inf.*, 71:259–277, February 2006.

[9] C. S. Iliopoulos, K. Perdikuri, E. Theodoridis, A. K. Tsakalidis, and K. Tsichlas. Algorithms for extracting motifs from biological weighted sequences. *J. Discrete Algorithms*, 5(2):229–242, 2007.

[10] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD Conference*, pages 327–338, 2010.

[11] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[12] T. Kahveci and A. K. Singh. Efficient index structures for string databases. In *VLDB*, pages 351–360, 2001.

[13] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

[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] M. Patil, X. Cai, S. V. Thankachan, R. Shah, S.-J. Park, and D. Foltz. Approximate string matching by position restricted alignment. In *EDBT/ICDT Workshops*, pages 384–391, 2013.

[16] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *SIGMOD*, pages 1239–1242, 2008.

[17] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[18] 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.

[19] H. Zhang, Q. Guo, and C. S. Iliopoulos. An algorithmic framework for motif discovery problems in weighted sequences. In *CIAC*, pages 335–346, 2010.