Efficient Exact Edit Similarity Query Processing with the Asymmetric Signature Scheme

Jianbin Qin[†] Wei Wang[†] Yifei Lu[†] Chuan Xiao[†] Xuemin Lin[†‡]

[†]School of Computer Science and Engineering, University of New South Wales
{jqin, weiw, yifeil, chuanx, lxue}@cse.unsw.edu.au

[‡] Software College, East Normal China University

ABSTRACT

Given a query string Q, an *edit similarity search* finds all strings in a database whose edit distance with Q is no more than a given threshold τ. Most existing method answering edit similarity queries rely on a signature scheme to generate candidates given the query string. We observe that the number of signatures generated by existing methods is far greater than the lower bound, and this results in high query time and index space complexities.

In this paper, we show that the minimum signature size lower bound is $\tau+1$. We then propose asymmetric signature schemes that achieve this lower bound. We develop efficient query processing algorithms based on the new scheme. Several dynamic programming-based candidate pruning methods are also developed to further speed up the performance. We have conducted a comprehensive experimental study involving *nine* state-of-the-art algorithms. The experiment results clearly demonstrate the efficiency of our methods.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Textual Databases*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern Matching*

General Terms

Algorithms, Performance

Keywords

Approximate Pattern Matching, Similarity Search, Similarity Join, Edit Distance, q-gram

1. INTRODUCTION

Given a query string Q, an *edit similarity search* finds all strings in a database whose edit distance with Q is less than a given threshold τ. Edit similarity searches have many applications, such as data integration and record linkage, bioinformatics, pattern recognition, and multimedia information retrieval. For example,

- In bioinformatics, edit similarity search can be employed to find similar protein sequences, and tandem repeats, which are useful to predicting diseases or designing new drugs [19, 27].
- Batch edit similarity searches, or *edit similarity joins*, can be used to find near duplicate records in a customer database [2], or near duplicate documents in a document repository [13].

As a result, there has been much interest in efficient algorithms to answer edit similarity search or join queries. This is an challenging problem, as edit distance computation is costly and a naïve algorithm that performs edit distance calculation for each string in the database is prohibitively expensive for large databases.

To address the performance challenge, most existing approaches adopt the *filter-and-verification* paradigm based on a signature scheme. A *candidate set* is generated for the query string by finding database strings that share at least a certain amount of common signatures with the query. Query results can be obtained by verifying the edit distance between each candidate and the query.

The numbers of signatures a method generates for data and query strings have a substantial impact on the query performance and index size. We give the numbers of signatures for data strings and the query string of several existing approaches in Table 1. Among them, Ed-Join has the smallest signature size with respect to τ. It is natural to wonder *if this is the minimum signature size, and if not, how we can further reduce the signature size.*

This paper presents our findings when trying to answer these two questions. First, we propose a framework of signature schemes and the associated query processing method for edit similarity queries. The framework encompasses all major signature-based algorithms for edit similarity queries. We prove that the lower bound on the minimum signature size for any algorithm in this framework is $\tau + 1$, where τ is the edit distance threshold. Next, we propose a novel signature scheme and corresponding query processing methods for edit similarity queries. Our proposal has three distinct features: (a) its minimum signature size is exactly $\tau + 1$, hence reaching the lower bound; (b) it is an asymmetric signature scheme — by asymmetric, we mean it uses different methods to generate signatures for data and query strings; (c) being asymmetric, we can instantiate two different edit similarity query processing algorithms out of it. Our two methods not only have interesting theoretic properties, but

SIGMOD'11, June 12–16, 2011, Athens, Greece.

are also highly efficient in practice. We also develop several candidate pruning techniques that further reduce the number of candidates needing verification. Finally, we perform a comprehensive experimental study comparing our two algorithms with nine state-of-the-art algorithms. Our algorithms demonstrate superior performance in most settings.

Our contributions can be summarized as:

- We are the first to introduce a general framework to capture the commonalities of many existing algorithms that are based on various kinds of signatures. We also show the lower bound of $\tau + 1$ for any algorithm belonging to this framework.
- We propose an asymmetric signature scheme that achieves the lower bound of the number of signatures on the data string or the query string.
- We design two efficient edit similarity query algorithms, IndexChunk and IndexGram, together with several novel candidate pruning algorithms.
- Although many algorithms have been proposed in the past decades on edit similarity queries, to the best of our knowledge, there is no systematic study of their performances. Hence, we conduct a comprehensive experimental study with seven state-of-the-art algorithms for edit similarity queries. Our proposed algorithms have been shown to outperform existing ones in terms of speed, index size, and robustness. The study also provides a clear picture of the relative performance and space-time tradeoffs of different algorithms.

The rest of the paper is organized as follows: Section 2 gives the problem definition and introduces related work. We describe the general framework that summarizes many signature-based edit similarity query algorithms in Section 3. We present an asymmetric signature scheme and show how to use it for edit similarity searches in Section 4. We propose several novel candidate pruning methods in Section 5. Experimental results are presented and analyzed in Section 6. Section 7 concludes the paper.

Note that we focus on solving the edit similarity queries *exactly* in this paper, thus excluding approximate or heuristic methods (e.g., Shingling [5], LSH [14], or BLAST [1]).

2. PROBLEM DEFINITION AND RELATED WORK

2.1 Problem Definition

Let Σ be a finite alphabet of symbols; each symbol is also called a *character*. A string S is an ordered array of symbols drawn from Σ. All subscripts start from 1. The length of string S is denoted as $|S|$. Each string S is also assigned an identifier $S.id$.

$ed(S, T)$ denotes the edit distance between strings S and T, which measures the minimum number of edit operations (insertion, deletion, and substitution) to transform S to T (and vice versa). It can be computed in $O(|S||T|)$ time and $O(\min(|S|, |T|))$ space using the standard dynamic programming [31].

Given a set of strings $\mathcal{S}$ and a query string Q, an *edit similarity selection* with threshold τ returns all strings $S \in \mathcal{S}$ such that $ed(S, Q) \leq \tau$ [15]. Many selection queries running in a batch mode result in the *edit similarity join* problem [7]. In this paper, we call edit similarity selections and joins collectively as *edit similarity queries*.

2.2 Prior Work

In the interest of space, we briefly survey prior work that is directly related to edit similarity query. We refer readers to the survey [23] and the recent tutorial [18] for a more complete coverage.

Similarity searches and joins have been studied for different representations of objects and similarity/distance functions. In spatial databases where objects are points in d-dimensional space, a similarity search using the Euclidean distance is just a *range search* and can be efficiently supported by R-trees [17] in low dimensional space and various specialized index data structures in high dimensional space [37]. Euclidean distance spatial joins in high dimensional space have been studied in [20].

Similarity searches in a metric space is hard and generally requires a metric index, such as M-tree [10]. Metric similarity joins based on the triangle inequality pruning and metric indexes have been proposed [11, 12].

Recently much work has been devoted to similarity searches and joins for sets and strings, including constraints defined using the overlap, Jaccard, cosine, and edit distance metrics [15, 26, 7, 3]. Most recently, even more complex similarity metrics are studied, such as the Bregman Divergence [39] and Earth Mover's Distance [35].

When edit similarity queries are considered, existing methods can be classified into three categories:

- *Gram-based.* Traditionally, fixed length q-grams are widely used for edit similarity search or join queries, because the count filtering is very effective in pruning candidates [15]. Together with prefix-filtering [7], the count filtering can also be implemented efficiently. Filters based on mismatching q-grams are proposed to further speed up the query processing [34]. Variable-length grams are also proposed [22, 36], which can be easily integrated into other algorithms and help to achieve better performance. Several list-merging methods were proposed by [21] to improve merge efficiency by skipping elements when probing inverted lists.
- *Tree-based.* A trie-based approach for edit similarity search has been proposed in [8]. It builds a trie for the dataset and support edit similarity search by incrementally probing the trie. [32] introduces a trie-based method to support edit similarity joins efficiently via sub-trie pruning techniques. [38] proposes a B^+-tree index structure B^{ed}-tree to support edit similarity queries through transforming strings into implicit digits and inserting them into a standard B^+-tree.
- *Enumeration-based.* Neighborhood generation-based methods enumerate all possible strings that are within τ edit distance from data strings. While naïve enumeration method only works in theory, recent proposals using deletion neighbourhood [29] and partitioning [33] can work well with small edit distance thresholds. PartEnum [2] performs enumeration based on partitions of the alphabet Σ and the strings.

Our proposed methods generally belongs to the gram-based approach. However, unlike all existing methods, our scheme uses different methods to extract (different) signatures from data strings and the query string. Another differences is in the number of signatures generated for query processing purpose. Our methods attain the lower bound on the minimum signature size. Nonetheless, we compared

our proposed methods with representative methods from all three categories in our experiment (Section 6).

Similarity searches or joins are usually much more costly than equality searches or joins. Even the latest exact similarity computation algorithms might be insufficient for huge amount of data or in applications with stringent time requirement. Therefore, another rich body of related work is to answer similarity queries approximately. The most influential work is those based on LSH [14, 4, 6]. There are also approximate methods based on heuristics [9, 30] or hashing [28].

We note that several works [24, 25] have used a similar idea to the IndexGram algorithm, namely, the query string is divided into multiple substrings and each substring is used to probe an index. The major differences are (1) we fixed the length of substring to q while they fix the number of substrings to $\tau + 1$, (2) thanks to prefix-filtering, our method only needs to process rare substrings, (3) we have better filtering algorithms to further remove the candidates. We have shown that IndexGram substantially outperforms these methods in the experiment (Section 6.5).

3. A SIGNATURE-BASED FRAMEWORK FOR EDIT SIMILARITY QUERIES

In this section, we develop a general framework for exact edit similarity queries. It encompasses a large number of existing methods for the problem. We also develop a lower bound for all schemes belonging to this framework and show there is a substantial gap between existing methods and the lower bound. This is exactly the motivation for our asymmetric signature scheme proposed later in Section 4.

In the rest of the paper, we consider edit similarity searches. In the interest of space, we defer the extension of our technique to edit similarity joins to the full version of this paper. Nonetheless, the join version of our methods are used in our experimental study (Section 6).

3.1 A Framework Based on Content Signatures

A general idea that underlies many existing solutions to edit similarity searches is that *if two strings are similar by having a small edit distance between them, then part of them (called* signatures *in this paper), must be identical.*

In this paper, we confine ourself to signatures that are part of the string content, hence named *content signature*[1]. A typical example is the q-gram, which is a substring of length q. More formally, consider a string S, a content signature is one of its non-empty *subsequences*. Different signature scheme admits different set of signatures by imposing certain restrictions. All possible signatures admitted by a signature scheme is called its *signature space*.

This above signature-based idea for edit similarity searches naturally suggests the following query processing method: given a query string Q, we can extract a signature from Q, and then find data strings that also generate the same signature (typically via an inverted index). It is obvious that this will immediately give us a *candidate set* with possible false positive results. Nonetheless, we can perform a pairwise verification between each candidate and the query to remove false positives and obtain the query answer.

However, the above idea is flawed for *exact* edit similarity searches, if only *one* signature is generated for the query or the data string. This is because given a pair of strings Q and S and their signatures, the two strings might differ exactly by one edit operation that destroys the signature. In order to guarantee that all query results are returned, we only consider signature schemes such that

- it extracts λ_τ signatures for a data string;
- it extracts Λ_τ signatures for a query string;
- the scheme has a tight lower bound, LB_τ, of common signatures for any two strings within an edit distance threshold.

Hence, a signature scheme suitable for exact edit similarity searches can be characterized as $\Gamma(\lambda_\tau, \Lambda_\tau, LB_\tau)$.

The above is the framework we propose. It encompasses many of the existing methods for edit similarity queries, including q-grams [15], VGRAMs [22], and signatures generated by enumeration [2, 33].

EXAMPLE 1. *A q-grams is a fixed length* substring *extracted from a given string. The q-gram-based signature scheme imposes the restriction that the signature length must be q. Let the alphabet be Σ, the signature space of q-grams is Σ^q. It was shown that if two strings are within edit distance τ, the intersection size of their q-grams sets must exceed a certain lower bound [16]. Hence, the method can be characterized as $\Gamma(|S|, |Q|, \max(|S|, |Q|) - q\tau)$.*

3.2 Minimum Signature Size

We define the *minimum signature size* of a signature scheme Γ as $\min(\lambda_\tau, \Lambda_\tau)$. Since the number of signatures is closely related to (1) the size of the index we need to build, and (2) the query performance of the method, we would like to find a signature scheme that minimizes this number. We first introduce the prefix filtering as a powerful reduction tool.

Prefix Filtering. Given a set U and a global ordering $\mathcal{O}$ for all elements in the universe, the θ-prefix of the set U, denoted as θ-prefix(U), is the first θ elements of U when all elements in U is sorted according to $\mathcal{O}$.

THEOREM 1 (PREFIX FILTERING, LEMMA 1 IN [7]). *Consider two sets U and V sorted according to a global order $\mathcal{O}$. If $|U \cap V| \geq \theta$ ($\theta < \min(|U|, |V|)$), then $(|U| - \theta + 1)$-prefix(U) $\cap$ $(|V| - \theta + 1)$-prefix(V) $\neq \emptyset$.*

Together with prefix filtering, the following lemma gives us a means to reducing the number of signatures if the lower bound required in the scheme is larger than 1. Therefore, in order to reduce the lower bound on the minimum signature size of any signature scheme in our framework, we only need to consider those schemes with lower bound of 1.

LEMMA 1. *Given a signature scheme $\Gamma(\lambda_\tau, \Lambda_\tau, LB_\tau)$ for exact edit similarity searches, there exists a signature scheme $\Gamma'(\lambda_\tau - LB_\tau + 1, \Lambda_\tau - LB_\tau + 1, 1)$ such that for any query Q, all candidates produced by Γ is a subset of candidates produced by Γ'.*

PROOF (SKETCH). We can explicitly construct the new signature scheme Γ' as follows:

- define an arbitrary total order for all signatures in the signature space of Γ.
- given the λ_τ signatures generated by Γ for a data string S, we only keep its $\lambda_\tau - LB_\tau + 1$-prefix as the signature of Γ' for S.

[1]When there is no ambiguity, we will simply refer content signatures as signatures.

Table 1: Worst Case Signature Sizes of Existing Edit Similarity Search/Join Methods

Method	$\lambda(\tau)$ Signatures for Data (S)	$\Lambda(\tau)$ Signatures for Query (Q)	Lower Bound
q-gram [15, 21]	$\lvert S\rvert$ q-grams [(a)]	$\lvert Q\rvert$ q-grams	$\max(\lvert S\rvert, \lvert Q\rvert) - q\tau$
Ed-Join [34]	$q\tau + 1$ q-grams	$q\tau + 1$ q-grams	1
VGRAM [22] [(b)]	$\lvert S\rvert + q_{\min} - 1$ VGRAMs	$\lvert Q\rvert + q_{\min} - 1$ VGRAMs	$\max(\lvert VG(S)\rvert - NAG(S,\tau), \lvert VG(Q)\rvert - NAG(Q,\tau))$ or by dynamic programming
NGPP [33]	$O(\tau^2 \cdot l_p)$ variants	$O(\tau^2 \cdot l_p)$ variants	1
PartEnum [2]	$O((q\tau)^{2.39})$ signatures	$O((q\tau)^{2.39})$ signatures	1

[(a)] Strings are padded with special characters at the end. [(b)] $q_{\min}$ is the minimum VGRAM length; $VG(X)$ is the number of VGRAMs generated for X; $NAG(X,\tau)$ is a pre-calculated number.

- given the Λ_τ signatures generated by Γ for a query string Q, we only keep its $\Lambda_\tau - LB_\tau + 1$-prefix as the signature of Γ' for Q.

According to Theorem 1, all candidates generated by Γ with lower bound LB are contained in the candidates generated by Γ' with lower bound of 1. □

Finally, for all signature schemes admitted by our framework, we have the following lower bound on its minimum signature size.

THEOREM 2 (LOWER BOUND OF MIN. SIGNATURE SIZE). *The minimum signature size of any scheme in our framework is at least $\tau + 1$, provided that the size of the signature space is at least $2\tau + 1$.*

PROOF (SKETCH). We prove the lower bound by contradiction. Assume there exists a signature scheme that extracts at most τ signatures from a string S (i.e., $\lambda_\tau \leq \tau$). Denote the signatures as $sigs(S)$. Consider an adversary that constructs a string T in the following manner: it considers each signature and uses one edit operation to change it to another signature which is not in $sigs(S)$. This is possible because the possible number of distinct signatures is more than 2τ. Then the edit distance between S and the resulting string T is at most τ. However, since all the signatures of S are "destroyed", S would not be retrieved if T is used as the query string. Therefore, this signature scheme cannot answer exact edit similarity queries. By symmetry, we can prove that there is no signature scheme that extract less than $\tau + 1$ signatures for the query string too. □

We summarize the minimum signature size for existing signature schemes in Table 1. As we can see from the table, the signature sizes of existing schemes are far from the lower bound $\tau + 1$. It is natural to ask *whether there exists a content signature scheme for exact edit similarity queries that has minimum signature size of $\tau + 1$*. In the next section, we show that this can be achieved by a novel asymmetric signature scheme.

4. q-chars: AN ASYMMETRIC SIGNATURE SCHEME

In this section, we propose an asymmetric signature scheme for edit similarity searches with threshold τ. By incorporating prefix filtering, we arrive at two new signature schemes that generate (and index) only $\tau + 1$ signatures for data strings or the query string, respectively.

4.1 q-chars-based Signature Scheme

We propose an asymmetric scheme for similarity searches and joins with an edit distance constraint. The idea is to extract q-grams from one string as signatures and extract q-chunks from another string as signatures. q-chunks are just substrings of length q that starts at $1 + i \cdot q$ positions in the string. In other words, all q-chunks of a string S, or *q-chunk set* (denoted as $c_q(S)$), form a disjoint yet complete partitioning of S. To make sure the last q-chunk has exactly q characters, we append $q - (\lvert S\rvert \mod q)$ special character \$ to the end of S.

The q-gram set of a string S is its all length q substrings. In order to make sure every character in S has a corresponding q-gram, we pad $q - 1$ special characters \$ to the end of S. The collection of q-grams generated for S is called its *q-gram set*, and is denoted as $g_q(S)$.

We call both q-gram and q-chunk signatures q-chars if there is no need to distinguish between them. Note that if two signatures are literally identical, we still treat them as two different signatures, as they come from different positions in the string [7].

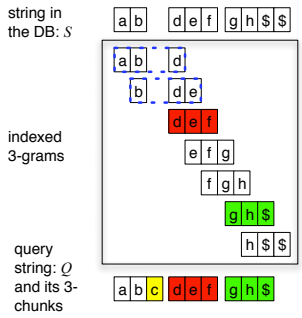


Figure 1: The q-chars Signature Scheme Example ($q = 3$)

EXAMPLE 2. *Consider the example in Figure 1. The data string S differs from the query string Q by deleting the character* c. *Note that we deliberately added spaces between characters in the strings and S's first two 3-grams, for the ease of illustration only.*

Now consider the three 3-chunks of Q. Note that these three 3-chunks can be deemed as a sample of 3-grams of Q. If there is no edit operation from Q to S, each of them will have a match in S's 3-gram set. Since in fact there is a deletion within the range of the first 3-chunk, its corresponding 3-gram in S will be destroyed. However, since there is no edit operation within the ranges of the rest of the 3-chunks,

their corresponding 3-grams are still preserved (albeit their offset in S might change).

Since Q has three q-chunks, it is obvious that any string S within edit distance of 1 from Q will preserve 3-1=2 q-chunks of Q. This is exactly the lower bound of one of our q-chars-based signature schemes (or more specifically, the basic IndexGram method).

The following theorem formally gives the lower bound for the q-chars-based signature scheme.

THEOREM 3 (LOWER BOUND OF COMMON SIGNATURES). *Let S and Q be two strings such that $ed(S,Q) \leq \tau$. Then both of the following inequalities hold:*

$$|g_q(S) \cap c_q(Q)| \geq \lceil |Q|/q \rceil - \tau \quad \textit{(for basic IndexGram)}$$
$$|c_q(S) \cap g_q(Q)| \geq \lceil |S|/q \rceil - \tau \quad \textit{(for basic IndexChunk)}$$

PROOF. We prove the first inequality and the second holds by symmetry. We say two signatures match if they are literally identical.

Let $k = \lceil |Q|/q \rceil$, where k is a constant and is the number of q-chunks for Q. Consider applying the edit operations from Q to S step by step. Before applying any edit operation, all k q-chunks have matching q-grams. Based on the position of each subsequent edit operation, we assign it to one of the q-chunks. For substitution, it is the q-chunk the modified character belongs to. For insertion, it is the q-chunk that the character preceding the inserted character belongs to. For deletion, it is the q-chunk that the deleted character belongs to. Hence, by the pigeon hole principle, with at most τ edit operations, there are at least $k - \tau$ q-chunks that have matching q-grams from S. □

We can further strengthen Theorem 3 by attaching the position information to each signature (i.e., q-grams or q-chunks). The position of a signature is the position of its first character in the string. We define two positional signatures, u and v, to be matching (with respect to τ), *if and only if* $u.sig = v.sig$ and $|u.pos - v.pos| \leq \tau$. Lemma 2 extends the lower bounds to positional signatures.

LEMMA 2. *Theorem 3 still holds when all signatures are positional signatures and the equality test between two signatures is replaced with matching test between two positional signatures.*

4.2 IndexChunk and IndexGram

There are two ways to apply the asymmetric q-chars signature scheme to edit similarity searches. Let a string in the dataset be S and the query string be Q. One way is to extract and index q-grams for all S in the database, and use q-chunks of Q as Q's signatures (as shown in Example 2). We call this method *basic IndexGram*. Another way is to extract and index q-chunks for strings in the database and use q-grams for the query string. This method is called *basic IndexChunk*.

Theorem 3 essentially gives us the count filter for q-chars-based signature scheme. In the same spirit as Lemma 1, by incorporating the prefix filtering, we can obtain a new signature scheme that generates fewer signatures.

For basic IndexGram, the lower bound of common signatures is $\lceil |Q|/q \rceil - \tau$. As the number of q-grams generated for S is $|S|$, the prefixes for the data strings should be its first $|S| - (\lceil |Q|/q \rceil - \tau) + 1$ q-grams; since $|Q| \geq |S| - \tau$ (due to the *length filtering*), the prefixes are the first $|S| - (\lceil (|S| - \tau)/q \rceil - \tau) + 1$ q-grams. The number of q-chunks generated for Q is $\lceil |Q|/q \rceil$, and the prefix for the query string is its first $\lceil |Q|/q \rceil - (\lceil |Q|/q \rceil - \tau) + 1 = \tau + 1$ q-chunks. We call this method IndexGram.

Similarly, we can derive that the prefix lengths for data strings in IndexChunk is $\tau + 1$, while the prefix length for the query string is $|Q| - (\lceil (|Q| - \tau)/q \rceil - \tau) + 1$.

Both IndexGram and IndexChunk have minimum signature size as $\tau + 1$. Hence both schemes are optimal according to Theorem 2. We list the detailed signature sizes for our algorithms in Table 2.

Algorithm 1: Preprocess+Index ($\mathcal{S}$, τ, $\mathcal{O}$)

```
   Data: S is the set of strings
         to be indexed. O is a global ordering of signatures.
1  for each string S ∈ S do
2      sigs ← the signature set of string S;
3      prefix_sigs ←
       the first λ_τ signatures from sigs ordered by O;
4      for each signature sig ∈ prefix_sigs do
5          I[sig] ← I[sig] ∪ (S.id, S.pos);
```

4.3 Query Preprocessing Algorithm

Preprocessing. In the preprocessing phase, we convert each data string into its corresponding signature set. Since we employ prefix filtering in both of our methods, an appropriate subset of signatures are further indexed using the inverted file. This process is illustrated in Algorithm 1.

The inverted index maps a q-chars signature into a list of strings such that the q-chars signature is among their prefix signatures. Each entry in the posting list consists of (id, pos), where id is the string ID, and pos is the starting position of the signature in the string id.

Algorithm 2: EditSimilarityQuery (R, τ)

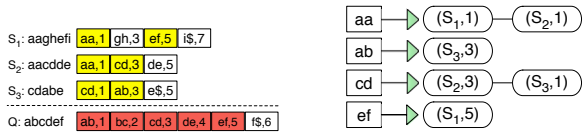
```
   Data: Q is the query string; I is an inverted index.
1  sigs ← signatures of Q;
2  prefix_sigs ← the first Λ_τ signatures of sigs;
3  candidates ← ∅;
4  for each signature sig ∈ prefix_sigs do
5      for each S ∈ I[sig] do
6          if S.id ∉ candidates
           and |S.len − |Q|| ≤ τ and |S.pos − sig.pos| ≤ τ then
7              candidates ← candidates ∪ {S.id};
8  for each candidate string S ∈ candidates do
9      if 2ndPhaseFilter(S, Q, τ, LB(S, Q)) then
10         if Verify(Q, S, τ) then
11             output S;
```

Answering Queries. We illustrate the edit similarity search algorithm in Algorithm 2. The algorithm has two phases.

- In the first candidate generation phase (Lines 1–7), it generates signatures for Q and use the appropriate prefix signatures to probe the index and generate candidates. For each candidate S returned from the inverted index probing, we apply length filtering and position filtering in Line 6.
- The second phase is in Lines 8–11. We apply a *second phase filtering* to each candidate to further reduce the number of candidates that have to be verified by the costly

Table 2: Worst Case Signature Sizes of q-chars-based Methods

Method	$\lambda(\tau)$ Signatures for Data (S)	$\Lambda(\tau)$ Signatures for Query (Q)	Lower Bound
Basic IndexChunk	$\lceil \lvert S\rvert/q\rceil$ q-chunks	$\lvert Q\rvert$ q-grams	$\lceil \lvert S\rvert/q\rceil - \tau$
IndexChunk	$\tau+1$ q-chunks	$\lvert Q\rvert - (\lceil(\lvert Q\rvert-\tau)/q\rceil - \tau)+1$ q-grams	1
Basic IndexGram	$\lvert S\rvert$ q-grams	$\lceil \lvert Q\rvert/q\rceil$ q-chunks	$\lceil \lvert Q\rvert/q\rceil - \tau$
IndexGram	$\lvert S\rvert - (\lceil(\lvert S\rvert-\tau)/q\rceil - \tau)+1$ q-grams	$\tau+1$ q-chunks	1



(a) Data Strings (with 2-chunks) and the Query (with 2-grams)

(b) Inverted Index for 2-chunks ($\tau = 1$)

Figure 2: IndexChunk Example

edit distance calculation (Line 10). We defer the discussion of the detail to Section 5. For now, we can think of a basic count filtering is applied here. If a candidate string passes the second phase filtering, its edit distance with Q is calculated and compared with the threshold in Line 10.

EXAMPLE 3. *Consider running the IndexChunk method on data and query strings in Figure 2. We consider $q = 2$ and $\tau = 1$. Figure 2(a) shows the 2-chunks and 2-grams.*

The lower bounds calculated for each S_i according to Theorem 3 are 3, 2, 2, respectively. If we naïvely intersect Q's 2-gram set with S_i's 2-chunk set, we obtain the intersection sizes as 1, 2, 2, respectively. However, if we use positional 2-grams and 2-chunks, the intersection sizes will be 1, 2, 0.

Now consider using prefix filtering in the IndexChunk method. We will just use the dictionary order as the global order $\mathcal{O}$, e.g., ab $\prec$ cd. *Since the prefix length for all strings are just $\tau + 1 = 2$, we only need to index the prefix 2-chunks (cells with yellow background). The inverted index built for the prefix is shown in Figure 2(b).*

Given query's 2-grams, we only uses its prefix signatures, which is the first 5 signatures (marked as red cells) according to $\mathcal{O}$. Probing these prefix signatures against the inverted index will give us candidate $\{S_2\}$ for cd *and $\{S_1\}$ for* ef. *Note that although S_3 is in* ab*'s posting list, since the two signatures' positions are more than 1 position away, S_3 is not added to the candidate set. The same holds for the S_3 entry in* cd*'s posting list.*

5. ADVANCED FILTERING

In this section, we consider several alternative ways to implement the second-phase filtering. We first introduce the naïve count filtering method and illustrate its tendency to over-estimate the matches. We then propose a dynamic programming-based algorithm that computes the maximum number of *true matches* and use it for more effective count filtering. We also design another dynamic programming-based algorithm that performs filtering directly by estimating the lower bound of the edit distance for a candidate pair.

5.1 Naïve Count Filtering

Line 9 of Algorithm 2 calls the function 2ndPhaseFilter to calculate the number of signatures shared by the data string and the query string. The naïve way to implement this function is given in Algorithm 3. It first loads the signatures of

Algorithm 3: NaïveCountFilter (Q, S, τ)

```
1  Load the signatures of Q and S ;     /* both are sorted */
2  g_sigs ← the q-gram signatures;
3  c_sigs ← the q-chunk signatures;
4  mismatch ← 0; M ← ∅;
5  LB ← the corresponding lower bound;
6  for each q-chunk signature chunk ∈ c_sigs do
7      match ← BinarySearch(g_sigs, chunk);
8      if match = nil then
9          mismatch ← mismatch + 1;
10         if mismatch > |c_sigs| − LB then
11             return (false, ∅)
12     else
13         while match ≠ nil and
           match = chunk and |chunk.pos − match.pos| ≤ τ do
14             M ← M ∪ (chunk, match);
15             match ← next(match)
               ;   /* move to the next q-gram signature */
16 f ← |M| ≥ LB;
17 return (f, M)
```

Q and S and counts the number of common signatures. In both q-chars-based methods, the signatures are a large set of q-grams and a small set of q-chunks. Given the difference in their sizes, we always iterate over the q-chunks, probing the longer q-grams to find a match. The criteria to decide a match are (1) the signatures have the same string content, and (2) their positions are within τ from each other (Line 13). Since the signatures are both sorted first by their global order and then their positions in the string, we can use binary search (Line 7). It is possible that the same q-gram appear multiple times in a string, hence we need to collect all such matches (Lines 13–15). Overall, the algorithm has a time complexity of $O(\frac{|Q|}{q} \cdot \log |Q|)$.

An optimization we injected into the algorithm is to keep track of the number of q-chunks that are not matched so far (in the variable *mismatch*). If the mismatch number is larger than total number of q-chunks less the lower bound, we can immediately prune the candidate pair (Lines 9–11).

5.2 Finding True Matches

Algorithm 3 returns M — a list of matches. As will be explained shortly, not all of them could be valid matches, hence we call them *candidate matches*.

We can model M as a bipartite graph $(U \cup V, E)$ as follows: for each match between a q-chunk c and a q-gram g, we create two nodes U_c and V_g and an edge between them.

EXAMPLE 4. *Consider the IndexGram method with $q = 2$, and the data string S and the query string Q in Figure 3. There are 5 candidate matches, as marked by 5 edges between the respective 2-grams (green rectangles) and 2-chunks (green eclipse).*

Algorithm 3 simply compare the size of candidate matches

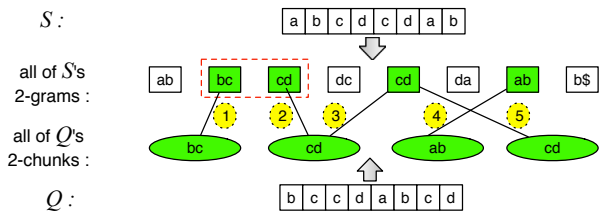


Figure 3: Illustrating Candidate Matches for Example 5 ($\tau = 2$)

with the lower bound to determine if the current candidate pair needs to be further verified or not (See Algorithm 2). This may admit false positives because two candidate matches might "conflict" with each other and only one of them is a *true match*. We use the following example to illustrate three types of conflicts.

EXAMPLE 5. *Consider the same example in Figure 3. Algorithm 3 will return 5 candidate matches as marked by yellow nodes (denoted as $M[i]$). We consider the following three types of conflicts*

- ***Multiple Matching (MM)**: edges $M[2]$ and $M[3]$ both stem from the 2nd chunk of Q.*
- ***Overlapping Matching (OM)**: edges $M[1]$ and $M[2]$ indicates that the first two chunks of Q are mapped to two overlapping bi-grams of S.*
- ***Cross Matching (CM)**: edges $M[4]$ and $M[5]$ cross each other.*

It can be shown, in any of the above three cases, that at most one of the (two) candidate matches is a true match. Without imposing the three constraints above, S and Q will be recognized as a valid candidate pair for any τ as all four q-chunks of Q have matches. However, we can compute the maximum number of matches respecting the three constraints as three (edges $M[1], M[3], M[4]$).

The following theorem further improves Lemma 2 by imposing the constraint to rule out any instance of MM, OM, or CM.

THEOREM 4. *Lemma 2 still holds with the* constraint *that no two of the matches of positional signatures belong to MM, OM, or CM.*

By removing the least number of the candidate matches, a set of candidate matches that does not observe the constraint can be made to be conflict-free and become *true matches*. We can return the number of true matches to perform count filtering.

Now the algorithmic problem is how to remove the least number of edges in the bipartite graph such that the resultant graph does not violate the constraint. This is essentially a specially constrained version of graph matching. Note that approximate solutions stemmed from unconstrained maximum graph matching cannot be used to prune candidate pairs.

Hence, we design the following dynamic programming-based algorithm to calculate the maximum matching number while observing the constraint. Let $opt[i]$ records the maximum number of true matches *if* the i-th edge in the candidate match list M is a true match and no more true matches after i. In order to calculate $opt[i]$, we need to find the last edge ($M[j]$) before the current one ($M[i]$), that is a true match. Once such an $M[j]$ is found, $opt[i]$ should be $opt[j]+1$. A straight-forward formulation would consider all preceding matches, i.e., $1 \le j < i$. However, since we have the lower bound (LB) on the number of q-chunks (hence the number of edges) that must be matched, we only need to consider l preceding edges, where $l = |M| - LB + 1$. This is because if the last true match edge is even before $M[i-l]$, there will be at most $LB-1$ edges that are matched, hence the candidate pair cannot satisfy the lower bound and should be discarded.

We also need to consider if the current edge and the last true match edge violate any of the three constraints. We say the two edges are compatible if they do not. We use a binary decision function $\delta(e_i, e_j)$ to encode this test, where $\delta(e_i, e_j) = 1$ iff e_i and e_j are compatible, and 0 otherwise. Given an edge e, denote its two vertices as $e.gram$ and $e.chunk$, respectively. Two edges e_i and e_j ($i < j$) are compatible if $e_i.chunk \neq e_j.chunk$ and $e_j.gram > e_i.gram + q$. The first test rules out MM and the second test rules out OM and CM (since $e_i.chunk \le e_j.chunk$).

Therefore, the final recursive formula[2] is:

$$\begin{cases} opt[k] = \max_{i=1}^{|M|-LB+1}\{\delta(M[k], M[k-i]) \cdot opt[k-i]\} + 1 \\ opt[0] = 0 \end{cases}$$

The recursive formula can be easily transformed into an efficient dynamic programming algorithm by filling the $opt[i]$ values with i ranging from 1 to $|M|$. The overall maximum number of true matches can be found from the last l elements of opt. If this number is less than LB, we could safely rule out this candidate pair.

Algorithm 4: DPTrueMatches (Q, S, τ)

```
Data: M' is all the candidate matches
      appended with a virtual omni-compatible edge
1 opt[0] ← 0;
2 for k = 1 to |M| do
3 |   max = −∞;
4 |   for i = 1 to min(k, |M| − LB + 1) do
5 |   |   if δ(M[k], M[k − i]) and opt[k − i] > max then
6 |   |   |   max ← opt[k − i] + 1;
7 |   opt[k] ← max;
8 return max_{i=LB}^{|M|}(opt[i])
```

EXAMPLE 6. *Consider the example in Figure 3, the first five values of the opt array when $k = 5$ is:* `[0, 1, 1, 2, 3, ]`. *To calculate opt[5], we need to consider its $|M| - LB + 1 = 4$ preceding edges. Among them, only $M[2]$ and $M[1]$ are compatible with $M[5]$. Hence $opt[5] = \max(opt[2], opt[1]) + 1 = 2$. Therefore, the opt array becomes* `[0, 1, 1, 2, 3, 2]`. *The final results is $\max_{i=LB}^{|M|}(opt[i]) = 3$.*

The algorithm has a time complexity of $O(|M|(|M| - LB))$. In most practical cases, since $|M| - LB$ is a small constant, the algorithm exhibits near linear time complexity.

5.3 Error Estimation-based Filtering

Another way to prune a candidate pair is to estimate a lower bound of the edit errors.

Assume that we have obtained a set of valid matches. This immediately gives us an alignment of the two strings. We

[2]We let $M[0]$ be compatible with all match edges.

develop an efficient method to estimate the minimum edit errors for this alignment. Our idea is that if we can enumerate all possible alignments (and their edit error lower bounds) involving at least one true match[3], and find the minimum value of these lower bounds, then it must be a lower bound of the edit distance (which must use one of the alignments we have explored). If this lower bound is larger than τ, the candidate pair can be discarded.

Observing that we only need to compute the minimum value of the lower bounds of all possible alignments, we propose a dynamic programming-based algorithm to efficiently calculate this value, thus saving us from a brute-force enumeration.

Estimating Edit Errors. First, we look at how to estimate an error lower bound for a particular alignment.

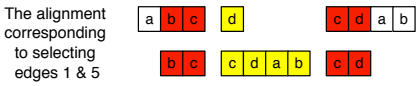


Figure 4: Illustrating the Error Estimation Method

EXAMPLE 7. *Assume we select edges 1 and 5 from Figure 3 as the true matches. This corresponds to an alignment shown in Figure 4. Consider the portions of strings between the two mapped edges (*`bc` *and* `cd`*). On one hand, the edit error must be at least the difference of these two substrings (in this example, 4 - 1 = 3). On the other hand, we know the second and the third q-chunks of Q are* not *matched, hence entailing edit distance of at least two. Therefore, the minimum edit error is finally estimated as* $\max(3, 2) = 3$.

Hence, we define the function $ed\_est(e_i, e_j)$ $(i < j)$ that estimates the lower bound of edit distance of two substrings obtained respectively by slicing edges e_i and e_j on the data and query strings as: [4] $ed\_est(e_i, e_j) = \max(\alpha, \beta)$ where $\alpha = \frac{e_j.chunk.pos - e_i.chunk.pos}{q} - 1$ and $\beta = |(e_j.chunk.pos - e_i.chunk.pos) - (e_j.gram.pos - e_i.gram.pos)|$.

Consider an alignment with k true matching edges, in the general case, it divides both strings into $k+1$ partitions. It can be shown that the sum of edit error estimations in each partition is also a lower bound of the edit distance between two strings.

Computing the Minimum Value of the Lower Bounds. Given an edge $M[i]$ as the current edge as a true match, it aligns q-chunks and q-grams to the left of itself. We denote the minimum value of lower bounds for such an partial alignement as $opt[i]$. We can obtain the following recursive formula:

$$opt[k] = \min_{i=1}^{|M|-LB+1} \{opt[k-i] + ed\_est(k-i, i)\} \quad (1)$$

EXAMPLE 8. *Consider the example in Figure 3. The opt array when* $k = 5$ *is:* `[0, 1, 1, 2, 2, ]`*.To calculate* $opt[5]$*, we need to consider its* $|M| - LB + 1 = 4$ *preceding edges. Among them, only* $M[2]$ *and* $M[1]$ *are compatible with* $M[5]$*. Hence* $opt[5] = \min(opt[2] + ed\_est(M[2], M[5]), opt[1] + ed\_est(M[1], M[5])) = 2$*. Therefore the opt array becomes* `[0, 1, 1, 2, 2, 2]`*.The final results is:* $\min_{i=LB}^{|M|}(opt[i] + ed\_est(M[i], \mathbf{nil})) = 3$.

[3]Otherwise, since the number of q-chunks is at least $\tau + 1$, the alignment has at least $\tau+1$ edit errors and hence can be discarded.

[4]Special cares need to be taken for the first and the last edges. $ed\_est(0, e_i)$ estimates errors from the start of two strings to the edge e_i, $ed\_est(e_i, \mathbf{nil})$ estimates errors from the edge e_i to the end of two strings.

Algorithm 5: DPErrEsti (Q, S, τ)

```
Data: M is all the candidate matches
      appended with a virtual omni-compatible edge
1 opt[0] ← 0;
2 for k = 1 to |M| do
3     min = ∞;
4     for i = 1 to min(k, |M| − LB + 1) do
5         if δ(M[k], M[k − i])
          and opt[k − i] + ed_est(k − i, i) < min then
6             min ← opt[k − i] + ed_est(k − i, i);
7     opt[k] = min;
8 return min_{i=LB}^{|M|}(opt[i] + ed_est(M[i], nil))
```

We design a dynamic programming-based algorithm to calculate the lower bound of the edit distance for a candidate pair (Algorithm 5). It is very similar to Algorithm 4 with the main difference that we calculate the minimum value of lower bound estimates (according to Equation (1)) and store it in the variable *min*. The minimum estimated edit error is equal to the match from $M[LB]$ to $M[|M|]$ whose *opt* value plus the error estimation to the end of strings are the smallest. If this error is over τ, we could prune this candidate pair.

The algorithm has a $O(|M|(|M| - LB))$ time complexity.

6. EXPERIMENTS

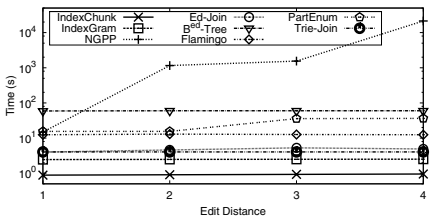
In this section, we report some of the most interesting findings in our comprehensive experimental study. We compared the performance of our two algorithms with seven other state-of-the-art methods (using publicly available implementation or implementation from original authors) for edit similarity queries.

6.1 Experiments Setup

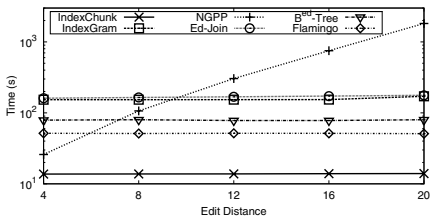
The following algorithms are used in the experiment.

- **IndexChunk** and **IndexGram** are our proposed algorithms that extract q-chunks and q-grams as signatures for the data strings, respectively.
- **Flamingo** [21] is a full-fledged open-source library for approximate string searches with several different similarity or distance functions[5]. We used the DivideSkipMerger [21] in its v3.0 release.
- **PartEnum** [2] is an edit similarity search and join method based on two-level partitioning and enumeration. We used the implementation in the Flamingo project and enhanced it to support both similarity searches and joins.
- **Ed-Join** is a q-grams-based edit similarity join algorithm using two mismatch filters [34]. We modified the source to support edit similarity searches.
- **B^{ed}-tree** [38] is a recent index structure for edit similarity searches and joins based on B$^+$-trees. It proposed three different transformations for efficient pruning of candidates during its query processing. We obtained the implementation from the authors.
- **Trie-Join** [32] is a recent trie-based edit similarity join method. We obtained the implementation from the authors.

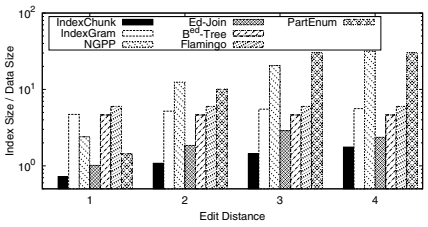
[5]`http://flamingo.ics.uci.edu/`



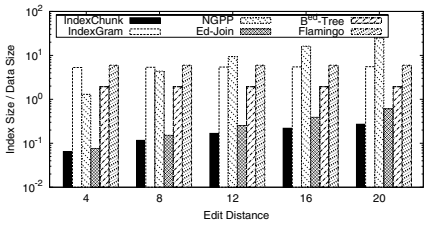
(a) IMDB, Preprocessing Time



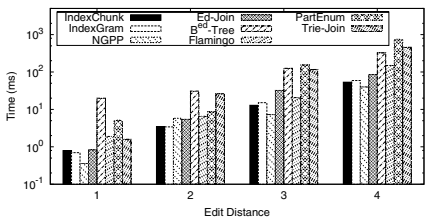
(b) UNIREF, Preprocessing Time



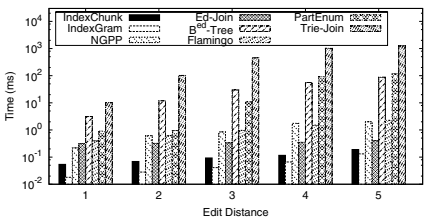
(c) IMDB, Relative Index Size



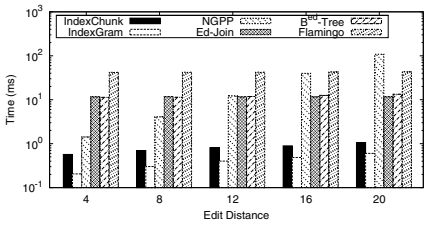
(d) UNIREF, Relative Index Size



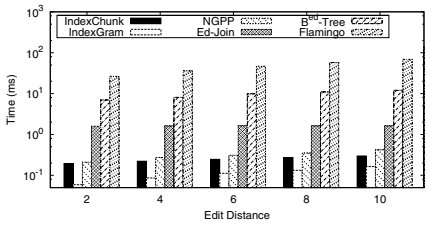
(e) IMDB, AVG Query Time



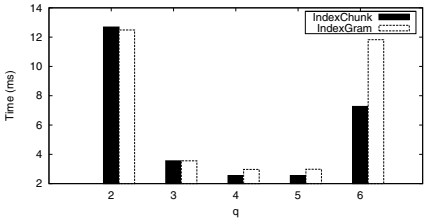
(f) DBLP, AVG Query Time



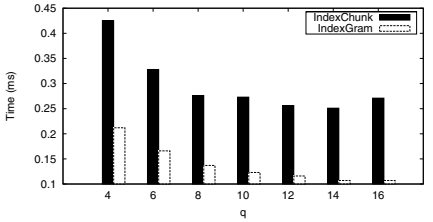
(g) TREC, AVG Query Time



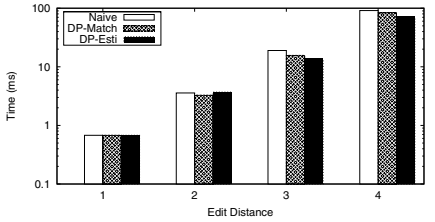
(h) UNIREF, AVG Query Time



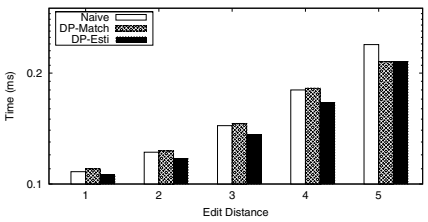
(i) IMDB, Varying q ($\tau = 2$)



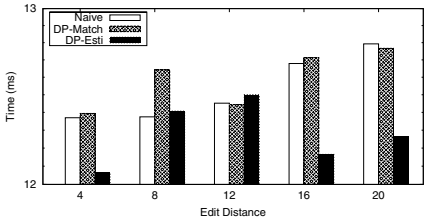
(j) UNIREF, Varying q ($\tau = 8$)



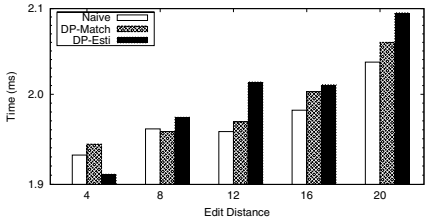
(k) IMDB, AVG Query Time with Diff. Filters



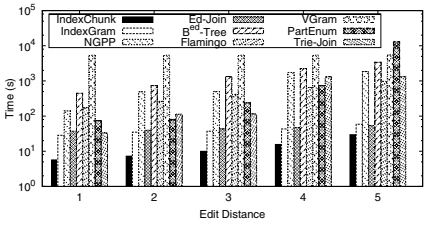
(l) DBLP, AVG Query Time with Diff. Filters



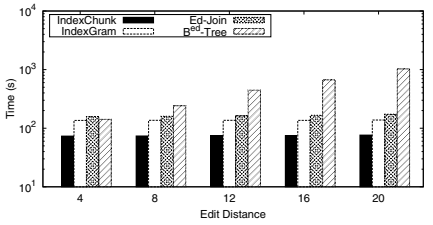
(m) TREC, AVG Query Time with Diff. Filters



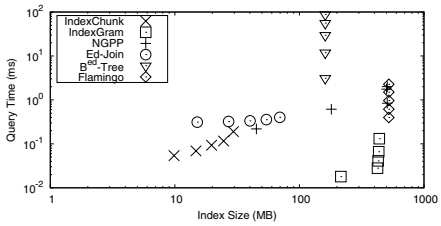
(n) UNIREF, AVG Query Time with Diff. Filters



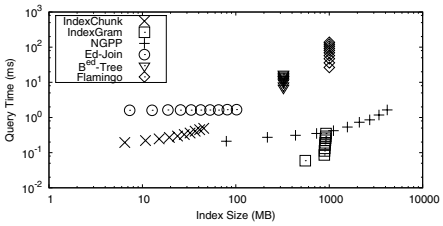
(o) DBLP, Join Time



(p) UNIREF, Join Time



(q) DLBP: Space vs. Time



(r) UNIREF: Space vs. Time

Figure 5: Experiment Results

- **NGPP** [33] is an edit similarity search algorithm originally developed for the approximate dictionary matching problem. It is based on a partitioning scheme together with deletion-neighborhood enumeration. We modify the source to support edit similarity joins.
- **VGRAM** [22, 36] is a novel signature extraction algorithm based on variable length grams. As such, it can be integrated into a variety of similarity search and join algorithms. We obtained the implementation from the authors.
- **PC** and **PF** [25] are two asymmetric methods for substring approximate matching, where the query string is always partitioned into $\tau + 1$ disjoint substrings. We obtained the implementation from the authors, and compared them with IndexGram in Section 6.5.

We selected four publicly available real datasets in the experiment. They cover a wide range of data distributions and application domains, and are used in previous studies.

- **IMDB** is an actor name dataset taken from the IMDB website[6].
- **DBLP** is a snapshot of the bibliography records from the DBLP website[7]. Each record is a concatenation of author name(s) and the title of a publication.
- **UNIREF** is the UniRef90 protein sequence data from the UniProt project.[8] Each sequence is an array of amino acids.
- **TREC** is from the TREC-9 Filtering Track Collections.[9] Each string is a reference from the MEDLINE database with author, title, and abstract information.

Statistics about the datasets are listed in Table 3.

Table 3: Statistics of the Datasets

Dataset	# of Strings	Avg Length	Size (MB)
IMDB	1,060,981	16	17
DBLP	860,751	106	88
UNIREF	377,438	464	281
TREC	239,580	1228	168

All experiments were carried out on a Quad-Core AMD Processor 8378@2.4GHz with 96GB RAM. The operating system is Linux 2.6.32 x86-64. All algorithms with source codes were coded in C++.

Note that

- We abuse the algorithm names to denote both its edit similarity search and join versions.
- In the interest of space, we may show representative results on some datasets.
- Results of certain algorithms are missing under some settings. This is mainly because they cannot finish within a reasonable amount of time, or the implementation has certain restriction.

6.2 Preprocessing Time and Index Size

We first test the preprocessing time for eight algorithms supporting edit similarity searches on all four datasets. We select results on IMDB and UNIREF to show in this section. The preprocessing time is measured as the elapsed time between when the system starts and when it is ready to process queries.[10] The results are shown in Figures 5(a)–5(b).

We can make the following observations.

- In terms of trend, most algorithms have almost flat preprocessing time as τ increases. Flamingo, Trie-Join and B^{ed}-tree are expected so, as they preprocess the entire dataset irrespective of τ. There is little increase in time for Ed-Join, IndexGram, and IndexChunk, as their prefixes and hence indexing time increases linearly with τ. Preprocessing time of PartEnum increases quickly after $\tau = 2$, as it generates more signatures (its asymptotic signature number per string is $O(\tau^{2.39})$ [2]). NGPP's time also increases very fast as τ increases. This is because the number of signatures it creates is $O(\tau^2)$.
- In terms of absolute time, IndexChunk is clearly the fastest on both datasets, as it only needs to index $\tau + 1$ signatures, which is the lower bound for all signature-based schemes. It takes only 20%–25% of the time used by the runner-up on the two datasets. The runner-up on IMDB is IndexGram and on UNIREF is Flamingo.

Next, we measured the *relative index size*, which is defined as the ratio of index size over data size. The results are shown in Figures 5(c)–5(d).

We observe that

- IndexChunk and Ed-Join belong to a group with the smallest index sizes, typically taking 10%–110% size of the data. IndexChunk is clearly the smallest as it indexes only $\tau + 1$ signatures. Its index size is only 3MB for the 270M TREC dataset for $\tau = 1$. Ed-Join index $q\tau + 1$ signatures in the worst case; but as seen here, in practice, it is much smaller than that. The index sizes of both algorithms also grow linearly with τ.
- the next group of algorithms is B^{ed}-tree, IndexGram, and Flamingo, typically taking 200%–800% size of the data. IndexGram always takes little bit less space than Flamingo, as can be expected from theoretical analysis. B^{ed}-tree organizes the index in a B^+-tree, yet usually occupies smaller space than the other two. The index sizes of these algorithms are typically insensitive to τ.
- NGPP's index size is competative only for $\tau \in [1, 2]$. Its index size increase rapidly with τ. PartEnum's index size is also very large. It flattens as we use a fixed (n_1, n_2) combination for large τs.

6.3 Edit Similarity Search Performance

To test the query processing time of all algorithms on four datasets, we generate 1000 random queries for each dataset. We measure the *average query time* and show the results of seven algorithms in Figure 5(e)–Figure 5(h).

We observe that

- Query performances on DBLP, TREC, and UNIREF exhibit certain patterns. (i) The fastest algorithm is IndexGram, followed by IndexChunk. The second runner-up is either NGPP or Ed-Join. The average query time of IndexGram is less than 1ms for all the thresholds tested on the three datasets. This is expected as it only probes the inverted index $\tau + 1$ times per query, hence generating a small candidate set efficiently. Other filters also contribute to keep its query time extremely low. (ii) The slowest algorithms are usually PartEnum, B^{ed}-tree, Flamingo,

[6] http://www.imdb.com

[7] http://www.informatik.uni-trier.de/~ley/db

[8] http://beta.uniprot.org/

[9] http://trec.nist.gov/data/t9_filtering.html

[10] Hence it includes the preprocessing and indexing time. Note that different algorithms may perform different tasks during this amount of time.

and Trie-Join. PartEnum is only competitive for $\tau \in [1, 2]$. Flamingo does not work well for large datasets consisting of long strings such as TREC and UNIREF. B^{ed}-tree, on the other hand, seems to be working better than Flamingo on TREC and UNIREF, but worse on DBLP. Trie-Join works well with small τs but its time increases quickly with large τs.

- The IMDB dataset is hard for all algorithms. NGPP has the best performance for almost all threshold settings, followed by IndexChunk, IndexGram, Flamingo, and Ed-Join. Still our two newly proposed algorithms have substantial lead over Ed-Join— the query time of IndexChunk, IndexGram, and Ed-Join are 12.9, 15.2, and 32.2 ms, respectively, for $\tau = 2$. Then PartEnum works reasonably well for $\tau \in [1, 2]$, but becomes slower than B^{ed}-tree and Flamingo when τ changes from 3 to 4.
- The overall trend for all algorithms is that the query time increases with τ. This is expected as a large τ leads to more candidates and also more results. The query time of most algorithms grows slowly with the increase of τ on DBLP, TREC, and UNIREF, but seems to grow rapidly on IMDB.

6.4 Tuning IndexChunk and IndexGram

We now turn to our two proposed algorithms and study their performances with respect to the choice of q and the filtering methods.

Effect of q. We show the average query time of IndexChunk and IndexGram with different q values in Figures 5(i)–5(j). Results on other datasets are similar.

We can see that the choice of q has substantial impact on the query time. On IMDB, the best q for both algorithms is within $[4, 5]$. On UNIREF, the best q is within $[12, 13]$ for IndexChunk, $[14, 16]$ for IndexGram. For both algorithms, a small q value means q-grams are not very selective and hence their postings lists are long; a large q value will reduce the lower bound of common signatures, hence reducing the effect of count filtering and requiring substantial verification costs to remove false positives.

Effect Of Filtering. We show the average query time of IndexChunk and IndexGram with different candidate filtering methods (Section 5) in Figures 5(k)–5(n).

As we can see, for datasets where strings are relatively short, such as DBLP and IMDB, DPErrEsti usually has the best performance; for datasets where strings are relatively long, such as TREC and UNIREF, NaïveCountFilter has slight advantage over both DPErrEsti and DPTrueMatches for most threshold settings. This is because we use small q for short string collections and large q for long string collections. When q is short, q-grams are not very selective, hence the number of candidate matches could be much higher than the number of true matches. For large q, the number of candidate matches is very close to the number of true matches, and additional filtering is not beneficial.

6.5 Comparing with PF and PC

We compared the IndexGram algorithm with the PF and PC algorithms. We concatenate all strings in a dataset into a single long string, in order to use the author's implementation. The average query times are given in Table 4. We can see that IndexGram outperforms PF and PC on datasets with short strings (IMDB) and long strings (UNIREF). We achieve a speedup of up to 3.5x on IMDB and 500–1,600x on UNIREF. This is mainly because we only probe the inverted index using q-grams with low frequencies.

Table 4: Comparing with PF and PC

(a) Average Query Time on IMDB (ms)

τ	IndexGram	PF	PC
1	0.69	1.03	0.96
2	3.45	7.67	7.89
3	15.15	48.15	48.83
4	59.10	207.99	212.14

(b) Average Query Time on UNIREF (ms)

τ	IndexGram	PF	PC
2	0.06	53.35	98.09
4	0.09	55.32	138.58
6	0.11	55.62	178.6

6.6 Similarity Joins

We now consider edit similarity joins. We first consider self-joining the DBLP dataset with $\tau \in [1, 5]$. We measure the overall time of the algorithms, i.e., including the preprocessing time and join time. The result is shown in Figure 5(o). We can see that

- the best performance is achieved by IndexChunk, followed by Ed-Join and IndexGram. The second best group of algorithms is PartEnum (for $\tau \in [1, 4]$) and Trie-Join. The rest of the algorithms, NGPP, B^{ed}-tree, and VGRAM, are among the slowest.
- the join time of all algorithms grows with the increase of τ. Some of the algorithms, e.g., Trie-Join, is on par with Ed-Join and IndexGram for $\tau = 1$, but its performance deteriorates rapidly with τ.

We also used the UNIREF dataset with $\tau \in [4, 20]$. Only four algorithms can finish within a reasonable amount of time (5 hours). The result is shown in Figure 5(p). We can see that IndexChunk is still the best algorithm, followed by IndexGram, Ed-Join, and finally B^{ed}-tree. The join time is not sensitive to τ for all algorithms except B^{ed}-tree, as the number of join results is relatively small compared to the size of dataset, and hence most running time is spent on preprocessing the data. The join time varies substantially with the choice of the algorithm, with the running time of B^{ed}-tree 10 times that of IndexChunk for $\tau = 20$.

6.7 Additional Observations and Summary

Edit Similarity Searches vs. Joins. One observation is that the relative performance among algorithms for edit similarity searches and joins is generally different. E.g., we can compare Figure 5(f) and Figure 5(o). IndexGram is the fastest search algorithm, but is only the third fastest for joins. Both Flamingo and NGPP perform pretty well on search, but not particularly efficient for joins.

Space-Time Characterization. Another observation is that these algorithms exhibit very different characteristics in terms of their space and time complexity (with varying τ). To illustrate this, we plot the index size and query time for six algorithms under different τ in Figure 5(q) and Figure 5(r). Note that both axes are in logarithmic scale. As

we can observe, the relative positions of these algorithms are fixed even for very different datasets (DBLP vs. UNIREF). As the following table shows, we may roughly categorize these algorithms into the following four quadrants.

Index Size	Query Performance	Algorithm(s)
Small	Very Fast	IndexChunk
Small	Fast	Ed-Join
Large	Very Fast	IndexGram, NGPP
Large	Fast	B^{ed}-tree, Flamingo

7. CONCLUSIONS

In this paper, we study the problem of efficiently processing edit similarity searches and joins. Unlike previous methods which extract equal amount of signatures for the data and query strings, we propose an asymmetric method based on extracting q-grams and q-chunks from the data and query strings. Based on this new signature scheme and prefix filtering, we design two algorithms, IndexChunk and IndexGram. They both achieve the minimum number of signatures as $\tau+1$. Several novel candidate pruning techniques are developed for the two algorithms. Finally, we have performed a comprehensive experimental study comparing our two algorithms with seven other state-of-the-art algorithms. Our algorithms outperform other algorithms in most cases. The experimental results also reveal interesting space-time trends of existing algorithms against threshold τ.

Acknowledgement. We thank all authors who sent us the implementations of their algorithms used in the experiments. We also thank the reviewers for their valuable comments and important references. Wei Wang was partially supported by ARC DP0987273 and DP0881779. Xuemin Lin was partially supported by ARC DP110102937, DP0987557, DP0881035, NSFC61021004, and Google Research Award.

8. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

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

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

[4] A. Z. Broder. On the resemblance and containment of documents. In *SEQS*, 1997.

[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[6] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

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

[8] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, 2009.

[9] A. Chowdhury, O. Frieder, D. A. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[10] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[11] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. Similarity join in metric spaces. In *ECIR*, pages 452–467, 2003.

[12] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using ed-index. In *DEXA*, 2003.

[13] G. Forman, K. Eshghi, and S. Chiocchetti. Finding similar files in large document repositories. In *KDD*, 2005.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[15] 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.

[16] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free (erratum). Technical Report CUCS-011-03, Columbia University, 2003.

[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[18] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.

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

[20] N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 12(1):3–18, 2000.

[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[22] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007.

[23] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[24] G. Navarro and R. A. Baeza-Yates. A practical q -gram index for text retrieval allowing errors. *CLEI Electron. J.*, 1(2), 1998.

[25] G. Navarro and L. Salmela. Indexing variable length substrings for exact and approximate matching. In *SPIRE*, pages 214–221, 2009.

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

[27] D. Sokol, G. Benson, and J. Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):30–35, 2007.

[28] B. Stein. Principles of hash-based text retrieval. In *SIGIR*, pages 527–534, 2007.

[29] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007.

[30] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*, pages 563–570, 2008.

[31] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[32] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit. In *VLDB*, 2010.

[33] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit constraints. In *SIMGOD*, 2009.

[34] 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.

[35] J. Xu, Z. Zhang, A. K. H. Tung, and G. Yu. Efficient and effective similarity search over probabilistic data based on earth mover's distance. *PVLDB*, 3(1):758–769, 2010.

[36] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, pages 353–364, 2008.

[37] R. Zhang, B. C. Ooi, and K.-L. Tan. Making the pyramid technique robust to query types and workloads. In *ICDE*, pages 313–324, 2004.

[38] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. B^{ed}-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD Conference*, pages 915–926, 2010.

[39] Z. Zhang, B. C. Ooi, S. Parthasarathy, and A. K. H. Tung. Similarity search on bregman divergence: Towards non-metric indexing. *PVLDB*, 2(1):13–24, 2009.