Efficient Similarity Joins for Near Duplicate Detection

Chuan Xiao Wei Wang Xuemin Lin
School of Computer Science and Engineering
University of New South Wales
Australia
{chuanx, weiw, lxue}@cse.unsw.edu.au

Jeffrey Xu Yu
Department of Systems Engineering and Engineering Management
Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

With the increasing amount of data and the need to integrate data from multiple data sources, a challenging issue is to find *near duplicate* records efficiently. In this paper, we focus on efficient algorithms to find pairs of records such that their similarities are above a given threshold. Several existing algorithms rely on the *prefix filtering* principle to avoid computing similarity values for all possible pairs of records. We propose new filtering techniques by exploiting the ordering information; they are integrated into the existing methods and drastically reduce the candidate sizes and hence improve the efficiency. Experimental results show that our proposed algorithms can achieve up to 2.6x–5x speed-up over previous algorithms on several real datasets and provide alternative solutions to the near duplicate Web page detection problem.

Categories and Subject Descriptors: H.3.3 [Information Search and Retrieval]: Search Process, Clustering

General Terms: Algorithm, Performance

Keywords: similarity join, near duplicate detection

1. INTRODUCTION

One of the issues accompanying the rapid growth of data on the Internet and the growing need to integrating data from heterogeneous sources is the existence of *near duplicate data*. Near duplicate data bear high similarity to each other, yet they are not bitwise identical. There are many causes for the existence of near duplicate data: typographical errors, versioned, mirrored, or plagiarized documents, multiple representations of the same physical object, spam emails generated from the same template, etc. As a concrete example, a sizeable percentage of the Web pages are found to be near-duplicates by several studies [6, 14, 18]. These studies suggest that around 1.7% to 7% of the Web pages visited by crawlers are near duplicate pages.

Identifying all the near duplicate objects benefits many applications. For example,

- For Web search engines, identifying near duplicate Web pages helps to perform focused crawling, increase the quality and diversity of query results, and identify spams [14, 11, 18].
- Many Web mining applications rely on the ability to accurately and efficiently identify near-duplicate objects. They include document clustering [6], finding replicated Web collections [9], detecting plagiarism [20], community mining in a social network site [25], collaborative filtering [3] and discovering large dense graphs [15].

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

A quantitative way to defining that two objects are near duplicates is to use a similarity function. The similarity function measures degree of similarity between two objects and will return a value in $[0, 1]$. A higher similarity value indicates that the objects are more similar. Thus we can treat pairs of objects with high similarity value as near duplicates. A *similarity join* will find all pairs of objects whose similarities are above a given threshold.

An algorithmic challenge is how to perform the similarity join in an *efficient* and *scalable* way. A naïve algorithm is to compare every pair of objects, thus bearing a prohibitively $O(n^2)$ time complexity. In view of such high cost, the prevalent approach in the past is to solve an *approximate* version of the problem, i.e., finding most of, if not all, similar objects. Several synopsis-based schemes have been proposed and widely adopted [5, 7, 10].

A recent trend is to investigate algorithms that compute the similarity join *exactly*. Recent advances include inverted index-based methods [24], prefix filtering-based methods [8, 3] and signature-based methods [1]. Among them, the recently proposed All-Pairs algorithm [3] was demonstrated to be highly efficient and be scalable to tens of millions of records. Nevertheless, we show that the All-Pairs algorithm, as well as other prefix filtering-based methods, usually generates a huge amount of candidate pairs, all of which need to be verified by the similarity function. Empirical evidence on several real datasets shows that its candidate size grows at a fast *quadratic* rate with the size of the data. Another inherent problem is that it hinges on the hypothesis that similar objects are likely to share rare "features" (e.g., rare words in a collection of documents). This hypothesis might be weakened for problems with a low similarity threshold or with a restricted feature domain.

In this paper, we propose new exact similarity join algorithms with application to near duplicate detection. We propose a *positional filtering* principle, which exploits the ordering of tokens in a record and leads to upper bound estimates of similarity scores. We show that it is complementary to the existing prefix filtering method and can work on tokens both in the prefixes and the suffixes. We conduct an extensive experimental study using several real datasets, and demonstrate that the proposed algorithms outperform previous ones. We also show that the new algorithms can be adapted or combined with existing approaches to produce results with better qualities or improve the runtime efficiency in detecting near duplicate Web pages.

The rest of the paper is organized as follows: Section 2 presents the problem definition and preliminaries. Section 3 summarizes the existing prefix filtering-based approaches. Sections 4 and 5 give our proposed algorithms that integrate positional filtering method on the prefixes and suffixes of the records, respectively. Generalization to other commonly used similarity measures is presented in Section 6. We present our experimental results in Section 7. Related work is covered in Section 8 and Section 9 concludes the paper.

2. PROBLEM DEFINITION AND PRELIMINARIES

2.1 Problem Definition

We define a record as a *set* of tokens taken from a finite universe $\mathcal{U} = \{ w_1, w_2, \ldots, w_{|\mathcal{U}|} \}$. A similarity function, *sim*, returns a similarity value in $[0, 1]$ for two records. Given a collection of records, a similarity function *sim*(), and a similarity threshold t, the *similarity join* problem is to find all pairs of records, $\langle x, y\rangle$, such that their similarities are no smaller than the given threshold t, i.e, $sim(x, y) \geq t$.

Consider the task of identifying near duplicate Web pages for example. Each Web page is parsed, cleaned, and transformed into a *multiset* of tokens: tokens could be stemmed words, q-grams, or shingles [5]. Since tokens may occur multiple times in a record, we will convert a multiset of tokens into a set of tokens by treating each subsequent occurrence of the same token as a new token [8]. We can evaluate the similarity of two Web pages as the Jaccard similarity between their corresponding sets of tokens.

We denote the *size* of a record x as $|x|$, which is the number of tokens in x. The document frequency of a token is the number of records that contain the token. We can *canonicalize* a record by sorting its tokens according to a global ordering $\mathcal{O}$ defined on $\mathcal{U}$. A *document frequency ordering* $\mathcal{O}_{df}$ arranges tokens in $\mathcal{U}$ according to the increasing order of tokens' document frequencies. A record x can also be represented as a $|U|$-dimensional vector, $\vec{x}$, where $x_i = 1$ if $w_i \in x$ and $x_i = 0$ otherwise.

The choice of the similarity function is highly dependent on the application domain and thus is out of the scope of this paper. We do consider several widely used similarity functions. Consider two records x and y,

- *Jaccard similarity* is defined as $J(x, y) = \frac{|x \cap y|}{|x \cup y|}$.
- *Cosine similarity* is defined as $C(x, y) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \cdot \|\vec{y}\|} = \frac{\sum_i x_i y_i}{\sqrt{|x|} \cdot \sqrt{|y|}}$.
- *Overlap similarity* is defined as $O(x, y) = |x \cap y|$.[1]

A closely related concept is the notion of *distance*, which can be evaluated by a distance function. Intuitively, a pair of records with high similarity score should have a small distance between them. The following distance functions are considered in this work.

- *Hamming distance* between x and y is defined as the size of their symmetric difference: $H(x, y) = |(x - y) \cap (y - x)|$.
- *Edit distance*, also known as *Levenshtein distance*, measures the minimum number of edit operations needed to transform one string into the other, where an edit operation is an insertion, deletion, or substitution of a single character. It can be calculated via dynamic programming [26].

[1]For the ease of illustration, we do no normalize the overlap similarity to $[0, 1]$.

Note that the above similarity and distance functions are inter-related. We discuss some important relationships in Section 2.2, and others in Section 6.

In this paper, we will focus on the Jaccard similarity, a commonly used function for defining similarity between sets. Extension of our algorithms to handle other similarity or distance functions appears in Section 6. Therefore, in the rest of the paper, $sim(x, y)$ by default denotes $J(x, y)$, unless otherwise stated.

Example 1. Consider two text document, D_x and D_y as:

$$D_x = \text{“yes as soon as possible”}$$
$$D_y = \text{“as soon as possible please”}$$

They can be transformed into the following two records

$$x = \{ A, B, C, D, E \}$$
$$y = \{ B, C, D, E, F \}$$

with the following word-to-token mapping table:

Word	*yes*	*as*	*soon*	*as*$_1$	*possible*	*please*
Token	*A*	*B*	*C*	*D*	*E*	*F*
Doc. Freq.	*1*	*2*	*2*	*2*	*2*	*1*

Note that the second “as” has been transformed into a token “as$_1$” in both records. Records can be canonicalized according to the document frequency ordering $\mathcal{O}_{df}$ into the following ordered sequences (denoted as $[\ldots]$)

$$x = [A, B, C, D, E]$$
$$y = [F, B, C, D, E]$$

The Jaccard similarity of x and y is $\frac{4}{6} = 0.67$, and the cosine similarity is $\frac{4}{\sqrt{5} \cdot \sqrt{5}} = 0.80$.

2.2 Properties of Jaccard Similarity Constraints

Similarity joins essentially evaluate every pair of records against a similarity constraint of $J(x, y) \geq t$. This constraint can be transformed into several equivalent forms on the overlap similarity or the Hamming distance as follows:

$$J(x, y) \geq t \Longleftrightarrow O(x, y) \geq \alpha = \frac{t}{1 + t} \cdot (|x| + |y|) \quad (1)$$

$$O(x, y) \geq \alpha \Longleftrightarrow H(x, y) \leq |x| + |y| - 2\alpha \quad (2)$$

We can also infer the following constraint on the relative sizes of a pair of records that meets a Jaccard constraint.

$$J(x, y) \geq t \Longrightarrow t \cdot |x| \leq |y| \quad (3)$$

3. PREFIX FILTERING BASED METHODS

A naïve algorithm to compute t-similarity join result is to enumerate and compare every pair of records. This method is obviously prohibitively expensive for large datasets, as the total number of comparisons is $O(n^2)$.

Efficient algorithms exist by converting the Jaccard similarity constraint into an equivalent overlap constraint due to Equation (1). An efficient way to find records that overlap with a given record is to use inverted indices [2]. An inverted index maps a token w to a list of identifiers of records that contain w. After inverted indices for all tokens in the record set are built, we can scan each record x, probe the indices using *every* token in x, and obtain a set of candidates; merging these candidates together gives us their actual overlap with the current record x; final results can be extracted by removing records whose overlap with x is less than $\lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil$ (Equation (1)). The main problem of this approach is that the inverted lists of some tokens, often known as "stop words", can be very long. These long inverted lists incur significant overhead for building and accessing them. In addition, computing the actual overlap by probing indices essentially requires memorizing all pairs of records that share at least one token, a number that is often prohibitively large. Several existing work takes this approach with optimization by *pushing the overlap constraint into the similarity value calculation phase*. For example, [24] employs sequential access on short inverted lists but switches to binary search on the $\alpha - 1$ longest inverted lists.

Another approach is based on the intuition that if two *canonicalized* records are similar, some fragments of them should overlap with each other, as otherwise the two records won't have enough overlap. This intuition can be formally captured by the *prefix-filtering principle* [8, Lemma 1] rephrased below.

Lemma 1 (Prefix Filtering Principle). *Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let the p-prefix of a record x be the first p tokens of x. If $O(x, y) \geq \alpha$, then the $(|x| - \alpha + 1)$-prefix of x and the $(|y| - \alpha + 1)$-prefix of y must share at least one token.*

Since prefix filtering is a necessary but not sufficient condition for the corresponding overlap constraint, we can design an algorithm accordingly as: we first build inverted indices on tokens that appear in the prefix of each record in an **indexing phase**. We then generate a set of *candidate pairs* by merging record identifiers returned by probing the inverted indices for tokens in the prefix of each record in a **candidate generation phase**. The *candidate pairs* are those that have the potential of meeting the similarity threshold and are guaranteed to be a superset of the final answer due to the prefix filtering principle. Finally, in a **verification phase**, we evaluate the similarity of each candidate pair and add it to the final result if it meets the similarity threshold.

A subtle technical issue is that the prefix of a record depends on the sizes of the other record to be compared and thus cannot be determined before hand. The solution is to index the longest possible prefixes for a record x. It can be shown that we only need to index a prefix of length $|x| - \lceil t \cdot |x| \rceil + 1$ for every record x to ensure the prefix filtering-based method does not miss any similarity join result.

The major benefit of this approach is that only smaller inverted indices need to be built and accessed (by a approximately $(1 - t)$ reduction). Of course, if the filtering is not effective and a large number of candidates are generated, the efficiency of this approach might be diluted. We later show that this is *indeed* the case and propose additional filtering methods to alleviate this problem.

There are several enhancements on the basic prefix-filtering scheme. [8] considers implementing the prefix filtering method on top of a commercial database system, while [3] further improves the method by utilizing several other filtering techniques in candidate generation phase and verification phase.

Example 2. Consider a collection of four canonicalized records based on the document frequency ordering, and the Jaccard similarity threshold of $t = 0.8$:

$$\begin{aligned} w &= [\, \underline{C}, D, F \,] \\ z &= [\, \underline{G, A}, B, E, F \,] \\ y &= [\, \underline{A, B}, C, D, E \,] \\ x &= [\, \underline{B, C}, D, E, F \,] \end{aligned}$$

Prefix length of each record u is calculated as $|u| - \lceil t \cdot |u| \rceil + 1$. Tokens in the prefixes are underlined and are indexed. For example, the inverted list for token C is $[\, w, x \,]$.

Consider the record x. To generate its candidates, we need to pair x with all records returned by inverted lists of tokens B and C. Hence, candidate pairs formed for x are $\{\, \langle x, y \rangle, \langle x, w \rangle \,\}$.

The All-Pairs algorithm [3] also includes several other filtering techniques to further reduce the candidate size. For example, it won't consider $\langle x, w \rangle$ as a candidate pair, as $|w| < 4$ and can be pruned due to Equation (3). This filtering method is known as size filtering [1].

4. POSITIONAL FILTERING

We now describe our solution to solve the exact similarity join problem. We first introduce the positional filtering, and then propose a new algorithm, ppjoin, that combines positional filtering with the prefix filtering-based algorithm.

4.1 Positional Filtering

Algorithm 1: ppjoin (R, t)

```
Input  : R is a multiset
         of records sorted by the increasing order of their
         sizes; each record has been canonicalized by a
         global ordering $\mathcal{O}$; a Jaccard similarity threshold $t$
Output : All pairs of records $\langle x, y \rangle$, such that $sim(x, y) \geq t$
1  $S \leftarrow \emptyset$;
2  $I_i \leftarrow \emptyset \; (1 \leq i \leq |U|)$;
3  for each $x \in R$ do
4      $A \leftarrow$ empty map from record id to int;
5      $p \leftarrow |x| - \lceil t \cdot |x| \rceil + 1$;
6      for $i = 1$ to $p$ do
7          $w \leftarrow x[i]$;
8          for each $(y, j) \in I_w$ such
           that $|y| \geq t \cdot |x|$ do      /* size filtering on $|y|$ */
9              $\alpha \leftarrow \lceil \frac{t}{1+t}(|x| + |y|) \rceil$;
10             ubound $\leftarrow 1 + \min(|x| - i, |y| - j)$;
11             if $A[y]$ + ubound $\geq \alpha$ then
12                 $A[y] \leftarrow A[y] + 1$;
13             else
14                 $A[y] \leftarrow 0$;                /* prune $y$ */;
15         $I_w \leftarrow I_w \cup \{(x, i)\}$;
           /* index the current prefix */;
16     Verify$(x, A, \alpha)$;
17 return $S$
```

Although a global ordering is a prerequisite of prefix filtering, no existing algorithm *fully* exploits it when generating the candidate pairs. We observe that *positional information* can be utilized in several ways to further reduce the candidate size. By positional information, we mean the position a token in a canonicalized record (starting from 1). We illustrate the observation in the following example.

Example 3. Consider x and y from the previous example and the same similarity threshold $t = 0.8$

$$y = [\,\underline{A, B}, C, D, E\,]$$
$$x = [\,\underline{B, C}, D, E, F\,]$$

The pair, $\langle x, y \rangle$*, does not meet the equivalent overlap constraint of* $O(x, y) \geq 5$*, hence is not in the final result. However, since they share a common token,* B*, in their prefixes, prefix filtering-based methods will select* y *as a candidate for* x*.*

However, if we look at the positions of the common token B *in the prefixes of* x *and* y*, we can obtain an estimate of the maximum possible overlap as the sum of current overlap amount and the minimum number of* unseen *tokens in* x *and* y*, i.e.,* $1 + \min(3, 4) = 4$*. Since this upper bound of the overlap is already smaller than the threshold of* 5*, we can safely prune* $\langle x, y \rangle$*.*

We now formally state the positional filtering principle in Lemma 2.

Lemma 2 (Positional Filtering Principle). *Consider an ordering* $\mathcal{O}$ *of the token universe* $\mathcal{U}$ *and a set of records, each with tokens sorted in the order of* $\mathcal{O}$*. Let token* $w = x[i]$*,* w *partitions the record into the* left partition $x_l(w) = x[1\,..\,(i-1)]$ *and the* right partition $x_r(w) = x[i\,..\,|x|]$*. If* $O(x, y) \geq \alpha$*, then for every token* $w \in x \cap y$*,* $O(x_l(w), y_l(w)) + \min(|x_r(w)|, |y_r(w)|) \geq \alpha$*.*

4.2 Positional Filtering-Based Algorithm

A natural idea to utilize the positional filtering principle is to combine it with the existing prefix filtering method, which already keeps tracks of the current overlap of candidate pairs and thus gives us $O(x_l(w), y_l(w))$.

Algorithm 1 describes our ppjoin algorithm, an extension to the All-Pairs algorithm [3], to combine positional filtering and prefix-filtering. Like the All-Pairs algorithm, ppjoin algorithm takes as input a collection of canonicalized record already sorted in the ascending ordered of their sizes. It then sequentially scans each record x, finds candidates that intersect x's prefix ($x[1\,..\,p]$, Line 5) and accumulates the overlap in a hash map A (Line 12). The generated candidates are further verified against the similarity threshold (Line 16) to return the correct join result. Note that the internal threshold used in the algorithm is an equivalent overlap threshold α computed from the given Jaccard similarity threshold t. The document frequency ordering $\mathcal{O}_{df}$ is often used to canonicalize the records. It favors rare tokens in the prefixes and hence results in a small candidate size and fast execution speed. Readers are referred to [3] for further details on the All-Pairs algorithm.

Now we will elaborate on several novel aspects of our extension: (i) the inverted indices used (Algorithm 1, Line 15), and (ii) the use of positional filtering (Algorithm 1, Lines 9–14), and (iii) the optimized verification algorithm (Algorithm 2).

In Line 15, we index both tokens *and their positions* for tokens in the prefixes so that our positional filtering can utilize the positional information. In Lines 9–14, we compute an upper bound of the overlap between x and y, and only admit this pair as a candidate pair if its upper bound is no less than the threshold α. Specifically, α is computed according to Equation (1); ubound is an upper bound of the overlap between right partitions of x and y with respect to the current token w, which is derived from the number of unseen tokens in x and y with the help of the positional information in the index I_w; $A[y]$ is the current overlap for left partitions of x and y. It is then obvious that if $A[y] +$ ubound is smaller than α, we can prune the current candidate y (Line 14).

Algorithm 2: Verify(x, A, α)

```
Input : $p_x$ is the
        prefix length of $x$ and $p_y$ is the prefix length of $y$
1  for each $y$ such that $A[y] > 0$ do
2      $w_x \leftarrow$ the last token in the prefix of $x$;
3      $w_y \leftarrow$ the last token in the prefix of $y$;
4      $O \leftarrow A[y]$;
5      if $w_x < w_y$ then
6          ubound $\leftarrow A[y] + |x| - p_x$;
7          if ubound $\geq \alpha$ then
8              $O \leftarrow O + \left|x[(p_x + 1)\,..\,|x|] \bigcap y[(A[y] + 1)\,..\,|y|]\right|$;
9      else
10         ubound $\leftarrow A[y] + |y| - p_y$;
11         if ubound $\geq \alpha$ then
12             $O \leftarrow O + \left|x[(A[y] + 1)\,..\,|x|] \bigcap y[(p_y + 1)\,..\,|y|]\right|$;
13     if $O \geq \alpha$ then
14         $S \leftarrow S \cup (x, y)$;
```

Algorithm 2 is designed to verify whether the actual overlap between x and candidates y in the current candidate set, $\{\, y \mid A[y] > 0 \,\}$, meets the threshold α. Notice that we've already accumulated in $A[y]$ the amount of overlaps that occur in the prefixes of x and y. An optimization is to first compare the last token in both prefixes, and only the suffix of the record with the *smaller* token (denoted the record as u) needs to be intersected with the entire other record (denoted as v). This is because the prefix of u consists of tokens that are smaller than w_u (the last token in u's prefix) in the global ordering and v's suffix consists of tokens that are larger than w_v. Since $w_u \prec w_v$, u's prefix won't intersect with v's suffix. In fact, the workload can still be reduced: we can skip the first $A[y]$ number of tokens in v since at least $A[y]$ tokens have overlapped with u's prefix and hence won't contribute to any overlap with u's suffix. The above method is implemented through Lines 4, 5, 8, and 12 in Algorithm 2. This optimization in calculating the actual overlap immediately gives rise to a pruning method. We can estimate the upper bound of the overlap as the length of the suffix of u (which is either $|x| - p_x$ or $|y| - p_y$). Lines 6 and 10 in the algorithm perform the estimation and the subsequent lines test whether the upper bound will meet the threshold α and prune away unpromising candidate pairs directly.

Experimental results show that utilizing positional information can achieve substantial pruning effects on real datasets. For example, we show the sizes of the candidates generated by ppjoin algorithm and All-Pairs algorithm for the DBLP dataset in Table 1.

4.3 Minimizing Tokens to be Indexed

The following Lemma allows us to further reduce the number of tokens to be indexed and hence accessed.

Table 1: Candidate Size (DBLP, Jaccard)

t	All-Pairs	ppjoin	ppjoin+
0.95	199,268	176,971	32,397
0.90	1,857,987	657,200	36,318
0.80	16,98,3319	3,303,232	63,265

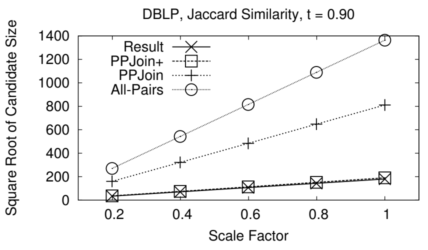


Figure 1: Quadratic Growth of Candidate Size

LEMMA 3. *Given a record x, we only need to index its $(|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1)$-prefix for Algorithm 1 to produce correct join result.*

This optimization requires us to change Line 15 in Algorithm 1 such that it only index the current token w if the current token position i is no larger than $|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$. Note that the length of the prefix used for probing into the indices remains the same.

5. SUFFIX FILTERING

In this section, we first motivate the need to looking for further filtering method, and then introduce a divide-and-conquer based *suffix filtering* method, which is a generalization of the positional filtering to the suffixes of the records.

5.1 Quadratic Growth of the Candidate Size

Let's consider the asymptotic behavior of the size of the candidate size generated by the prefix filtering-base methods. The candidate size is $O(n^2)$ in the worst case. Our empirical evidence on several real datasets suggests that the growth is *indeed* quadratic. For example, we show the *square root* of query result size and candidate sizes of the All-Pairs algorithm and our ppjoin algorithm in Figure 1. It can be observed that while positional filtering helps to further reduce the size of the candidates, it is still growing quadratically (albeit with a much slower rate than All-Pairs).

5.2 Generaling Positional Filtering to Suffixes

Given the empirical observation about the quadratic growth rate of the candidate size, it is desirable to find additional pruning method in order to tackle really large datasets.

Our goal is to develop additional filtering method that prunes candidates that survive the prefix and positional filtering. Our basic idea is to generalize the positional filtering principle to work on the suffixes of candidate pairs. However, the challenge is that the suffixes of records are *not* indexed *nor* their partial overlap has been calculated. Therefore, we face the following two technical issues: (i) how to establish an upper bound in the absence of indices or partial overlap results? (ii) how to find the position of a token without tokens being indexed?

We solve the first issue by converting an overlap constraint to an equivalent Hamming distance constraint, according to Equation (2). We then lower bound the Hamming distance by partitioning the suffixes in an coordinated way. We denote the suffix of a record x as x_s. Consider a pair of records, $\langle x, y\rangle$, that meets the Jaccard similarity threshold t, and without loss of generality, $|y| \leq |x|$. Since their overlap in their prefixes is at most the minimum length of the prefixes, we can derive the following upper bound in terms of the Hamming distance of their suffixes.

$$H(x_s, y_s) \leq H_{\max} = 2|x| - 2\lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil - (\lceil t \cdot |x| \rceil - \lceil t \cdot |y| \rceil) \quad (4)$$

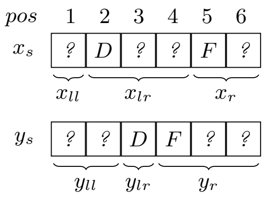
In order to check whether $H(x_s, y_s)$ exceeds the maximum allowable value, we provide an estimate of the lower bound of $H(x_s, y_s)$ below. First we choose an arbitrary token w from y_s, and divide y_s into two partitions: the left partition y_l and the right partition y_r. The criterion for the partitioning is that the left partition contains all the tokens in y_s that precede w in the global ordering and the right partition contains w (if any) and tokens in y_s that succeed w in the global ordering. Similarly, we divide x_s into x_l and x_r using w too (even though w might not occur in x). Since x_l (x_r) shares no common token with y_r (y_l), $H(x_s, y_s) = H(x_l, y_l) + H(x_r, y_r)$. The lower bound of $H(x_l, y_l)$ can be estimated as the difference between $|x_l|$ and $|y_l|$, and similarly for the right partitions. Therefore,

$$H(x_s, y_s) \geq \text{abs}(|x_l| - |y_l|) + \text{abs}(|x_r| - |y_r|)$$

Finally, we can safely prune away candidates whose lower bound Hamming distance is already larger than the allowable threshold $H_{\max}$.

We can generalize the above method to more than one probing token and repeat the test several times independently to improve the filtering rate. However, we will show that if the probings are arranged in a more coordinated way, results from former probings can be taken into account and make subsequent probings more effective. We illustrate this idea in the example below.

Example 4. Consider the following two suffixes of length 6. Cells marked with "?" indicate that we have not accessed those cells and do not know their contents yet.



Assume the allowable Hamming distance is 2. If we probe the 4th token in y_s ("F"), we have the following two partitions of y_s: $y_l = y_s[1\,..\,3]$ and $y_r = y_s[4\,..\,6]$. Assuming a magical "partition" function, we can partition x_s into $x_s[1\,..\,4]$ and $x_s[5\,..\,6]$ using F. The lower bound of Hamming distance is $\text{abs}(3-4) + \text{abs}(3-2) = 2$.

If we perform the same test independently, *say, using the 3rd token of y_s ("D"), the lower bound of Hamming distance is still 2. Therefore, $\langle x, y\rangle$ is not pruned away.*

However, we can actually utilize the previous test result. The result of the second probing can be viewed as a recursive *partitioning of x_l and y_l into x_{ll}, x_{lr}, y_{ll}, and y_{lr}. Obviously the total absolute differences of the sizes of the three partitions from two suffixes is an lower bound of their Hamming distance, which is*

$$\begin{aligned}&\text{abs}(|x_{ll}| - |y_{ll}|) + \text{abs}(|x_{lr}| - |y_{lr}|) + \text{abs}(|x_r| - |y_r|)\\ =&\,\text{abs}(1-2) + \text{abs}(3-1) + \text{abs}(2-3) = 4\end{aligned}$$

Therefore, $\langle x, y\rangle$ can be safely pruned.

Algorithm 3: SuffixFilter($x, y, H_{\max}, d$)

```
Input  : Two set of tokens $x$ and $y$,
         the maximum allowable hamming distance $H_{\max}$
         between $x$ and $y$, and current recursive depth $d$
Output : The lower
         bound of hamming distance between $x$ and $y$
1  if $d >$ MAXDEPTH then return abs($|x| - |y|$) ;
2  $mid \leftarrow \lceil \frac{|y|}{2} \rceil$; $w \leftarrow y[mid]$;
3  $o \leftarrow \frac{H_{\max} - \text{abs}(|x|-|y|)}{2}$;            /* always divisible */;
4  if $|x| < |y|$ then $o_l \leftarrow 1, o_r \leftarrow 0$ else $o_l \leftarrow 0, o_r \leftarrow 1$;
5  $(y_l, y_r, f, \text{diff}) \leftarrow$ Partition($y, w, mid, mid$);
6  $(x_l, x_r, f, \text{diff}) \leftarrow$ Partition($x, w, mid -$
   $o - \text{abs}(|x| - |y|) \cdot o_l, mid + o + \text{abs}(|x| - |y|) \cdot o_r$);
7  if $f = 0$ then
8      return $H_{\max} + 1$
9  $H \leftarrow \text{abs}(|x_l| - |y_l|) + \text{abs}(|x_r| - |y_r|) + \text{diff}$;
10 if $H > H_{\max}$ then
11     return $H$
12 else
13     $H_l \leftarrow$ SuffixFilter($x_l, y_l, H_{\max} - \text{abs}(|x_r| - |y_r|) - \text{diff}, d+1$) ;
14     $H \leftarrow H_l + \text{abs}(|x_r| - |y_r|) + \text{diff}$;
15     if $H \leq H_{\max}$ then
16         $H_r \leftarrow$ SuffixFilter($x_r, y_r, H_{\max} - H_l - \text{diff}, d+1$) ;
17         return $H_l + H_r + \text{diff}$
18     else
19         return $H$
```

The algorithm we designed to utilize above observations is a divide-and-conquer one (Algorithm 3). First, the token in the middle of y is chosen, and x and y are partitioned into two parts respectively. The lower bounds of Hamming distance on both left and right partitions are computed and summed up to judge if the overall hamming distance is within the allowable threshold (Lines 9–10). Then we call the SuffixFilter function recursively first on the left and then on the right partition (Lines 13–19). Probing results in the previous tests are used to help reduce the maximum allowable Hamming distance (Line 16) and to break the recursion if the Hamming distance lower bound has exceeded the threshold $H_{\max}$ (Lines 14–15 and 19). Finally, only those pairs such that their lower bounding Hamming distance meets the threshold will be considered as candidate pairs. We also use a parameter MAXDEPTH to limit the maximum level of recursion (Line 1); this is aimed to strike a balance between filtering power and filtering overhead.

The second technical issue is how to perform the partition efficiently, especially for x_s. A straight-forward approach is to perform binary search on the whole suffix, an idea which was also adopted by the ProbeCount algorithm [24]. The partitioning cost will be $O(\log |x_s|)$. Instead, we found that the search only needs to be performed in a much smaller area approximately centered around the position of the partitioning token w in y, due to the Hamming distance constraint. We illustrate this using the following example.

Example 5. Continuing the previous example, consider partitioning x_s according to the probing token F. The only possible area where F (for simplicity, assume F exists in x_s) can occur is within $x_s[3\,..\,5]$, as otherwise, the Hamming distance between x_s and y_s will exceed 2. We only need to perform binary search within $x_s[3\,..\,5]$ to find the first token that is no smaller than F.

The above method can be generalized to the general case where x_s and y_s have different lengths. This is described in Lines 4–6 in Algorithm 3. The size of the search range is bounded by $H_{\max}$, and is likely to be smaller within the subsequent recursive calls.

Algorithm 4 implements the partitioning process using a partitioning token w. One thing that deviates from Example 4 is that the right partition now does *not* include the partitioning token, if any (Line 7). This is mainly to simplify the pseudocode while still ensuring a tight bound on the Hamming distance when the token w cannot be found in x_s.

Algorithm 4: Partition(s, w, l, r)

```
Input  : A set of tokens $s$, a token
         $w$, left and right bounds of searching range $l$, $r$
Output : Two subsets
         of $s$: $s_l$ and $s_r$, a flag $f$ indicating whether $w$
         is in the searching range, and a flag diff indicating
         whether the probing token $w$ is not found in $y$
1  $s_l \leftarrow \emptyset$; $s_r \leftarrow \emptyset$;
2  if $s[l] > w$ or $s[r] < w$ then
3      return $(\emptyset, \emptyset, 0, 1)$
4  $p \leftarrow$ binary search for the position of the first token in $s$ that
   is no smaller than $w$ in the global ordering within $s[l\,..\,r]$;
5  $s_l \leftarrow s[1\,..\,p-1]$;
6  if $s[p] = w$ then
7      $s_r \leftarrow s[(p+1)\,..\,|s|]$;           /* skip the token $w$ */;
8      diff $\leftarrow 0$;
9  else
10     $s_r \leftarrow s[p\,..\,|s|]$;
11     diff $\leftarrow 1$;
12 return $(s_l, s_r, 1, \text{diff})$
```

Algorithm 5: Replacement of Line 12 in Algorithm 1

```
1  if $A[y] = 0$ then
2      $H_{\max} \leftarrow |x| + |y| - 2 \cdot \lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil - (i + j - 2)$;
3      $H \leftarrow$ SuffixFilter($x[(i+1)\,..\,|x|], y[(j+1)\,..\,|y|], H_{\max}, 1$);
4      if $H \leq H_{\max}$ then
5          $A[y] \leftarrow A[y] + 1$;
6      else
7          $A[y] \leftarrow -\infty$;     /* avoid considering $y$ again */;
```

Finally, we can integrate the suffix filtering into the ppjoin algorithm and we name the new algorithm ppjoin+. To that end, we only need to replace the original Line 12 in Algorithm 1 with the lines shown in Algorithm 5. We choose to perform suffix filtering only *once* for each candidate pair on the *first* occasion that it is formed. This is because suffix filtering probes the unindexed part of the records, and is relative expensive to carry out. An additional optimization opportunity enabled by this design is that we can further reduce the initial allowable Hamming distance threshold to $|x| + |y| - 2\lceil \frac{t}{1+t} \cdot (|x| + |y|)\rceil - (i + j - 2)$, where i and j stand for the positions of the first common token w in x and y, respectively (Line 2). Intuitively, this improvement is due to the fact that $x[1\,..\,(i-1)] \cap y[1\,..\,(j-1)] = \emptyset$ since the current token is the *first* common token between them.

The suffix filtering employed by the ppjoin+ algorithm is orthogonal and complementary to the prefix and positional filtering, and thus helps further reduce the candidate size. Its effect on the DBLP dataset can be seen in Table 1 and Figure 1.

6. EXTENSION TO OTHER SIMILARITY MEASURES

In this section, we briefly comment on necessary modifications to adapt both ppjoin and ppjoin+ algorithms to other commonly used similarity measures. The major changes are related to the length of the prefixes used for indexing (Line 15, Algorithm 1) and used for probing (Line 5, Algorithm 1), the threshold used by size filtering (Line 8, Algorithm 1) and positional filtering (Line 9, Algorithm 1), and the Hamming distance threshold calculation (Line 2, Algorithm 5).

Overlap Similarity $O(x, y) \geq \alpha$ is inherently supported in our algorithms. The prefix length for a record x will be $x - \alpha + 1$. The size filtering threshold is α. It can be shown that positional filtering will not help pruning candidates, but suffix filtering is still useful. The Hamming distance threshold, $H_{\max}$, for suffix filtering will be $|x|+|y|-2\alpha-(i+j-2)$.

Edit Distance Edit distance is a common distance measure for strings. An edit distance constraint can be converted into *weaker* constraints on the overlap between the q-gram sets of the two strings. Specifically, let $|u|$ be the length of the string u, a necessary condition for two strings to have less than δ edit distance is that their corresponding q-gram sets must have overlap no less than $\alpha = (\max(|u|, |v|) + q - 1) - q\delta$ [17].

The prefix length of a record x (which is now a set of q-grams) is $q\delta + 1$. The size filtering threshold is $|x| - \delta$. Positional filtering will use an overlap threshold $\alpha = |x| - q\delta$. The Hamming distance threshold, $H_{\max}$, for suffix filtering will be $|y| - |x| + 2q\delta - (i + j - 2)$.

Cosine Similarity We can convert a constraint on cosine similarity to an equivalent overlap constraint as:

$$C(x, y) \geq t \Longleftrightarrow O(x, y) \geq \left\lceil t \cdot \sqrt{|x| \cdot |y|} \right\rceil$$

The length of the prefix for a record x is $|x| - \lceil t^2 \cdot |x| \rceil + 1$, yet the length of the tokens to be indexed can be optimized to $|x| - \lceil t \cdot |x| \rceil + 1$. The size filtering threshold is $\lceil t^2 \cdot |x| \rceil$.[2] Positional filtering will use an overlap threshold $\alpha = \left\lceil t \cdot \sqrt{|x| \cdot |y|} \right\rceil$. The Hamming distance threshold, $H_{\max}$, for suffix filtering will be $|x| + |y| - 2\left\lceil t \cdot \sqrt{|x| \cdot |y|} \right\rceil - (i + j - 2)$.

7. EXPERIMENTAL EVALUATION

In this section, we present our experimental results.

7.1 Experiment Setup

We implemented and used the following algorithms in the experiment.

All-Pairs is an efficient prefix filtering-based algorithm capable of scaling up to tens of millions of records [3].

ppjoin, ppjoin+ are our proposed algorithms. ppjoin integrates positional filtering into the All-Pairs algorithm, while ppjoin+ further employes suffix filtering.

[2]These are the same bounds obtained in [3].

Dataset	n	avg_len	$\lvert U \rvert$
DBLP	873,524	14.0	566,518
DBLP-3GRAM	873,524	102.5	113,169
ENRON	517,386	142.4	1,180,186
TREC-4GRAM	348,566	866.9	1,701,746
TREC-Shingle	348,566	32.0	9,788,436

Figure 2: Statistics of Datasets

All algorithms were implemented in C++. To make fair comparisons, all algorithms use Google's `dense_hash_map` class for accumulating overlap values for candidates, as suggested in [3]. All-Pairs has been shown to consistently outperform alternative algorithms such as ProbeCount-Sort [24], PartEnum [1] and LSH [16], and therefore we didn't consider them [3].

All experiments were performed on a PC with Pentium D 3.00GHz CPU and 2GB RAM. The operating system is Debian 4.1. The algorithms were compiled using GCC 4.1.2 with `-O3` flag.

We measured both the size of the candidate pairs and the running time for all the experiments.

Our experiments covered the following similarity measures: **Jaccard** similarity, and **Cosine** similarity.

We used several publicly available real datasets in the experiment. They were selected to cover a wide spectrum of different characteristics (See Figure 2).

DBLP This dataset is a snapshot of the bibliography records from the DBLP Web site. It contains almost 0.9M records; each record is a concatenation of author name(s) and the title of a publication. We tokenized each record using white spaces and punctuations. The same DBLP dataset (with smaller size) was also used in previous studies [1, 3].

DBLP-3GRAM This is the same DBLP dataset, but further tokenized into 3-grams. Specifically, tokens in a record are concatenated with a single whitespace, and then every 3 consecutive letters is extract as a 3-gram.

ENRON This dataset is from the Enron email collection[3]. It contains about 0.5M emails from about 150 users, mostly senior management of Enron. We tokenize the email title and body into words using the same tokenization procedure as DBLP.

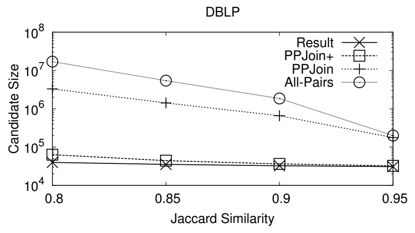
TREC-4GRAM This dataset is from TREC-9 Filtering Track Collections.[4] It contains 0.35M references from the MEDLINE database. We extracted author, title, and abstract fields to from records. Records are subsequently tokenized as in DBLP.

TREC-Shingle We applied Broder's shingling method [5] on TREC-4GRAM to generate 32 shingles of 4 bytes per record, using min-wise independent permutations. TREC-4GRAM and TREC-Shingle are dedicated to experiment on near duplicate Web page detection (Section 7.5).

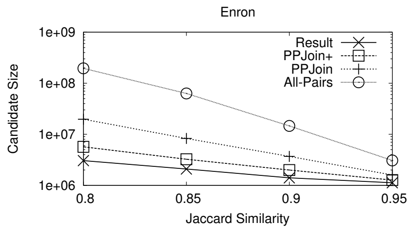
Some important statistics about the datasets are listed in Figure 2.

[3]Available at `http://www.cs.cmu.edu/~enron/`

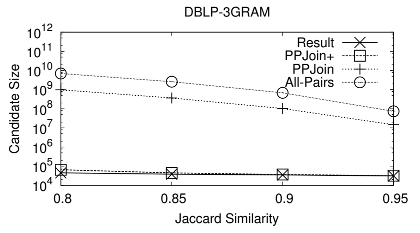
[4]Available at `http://trec.nist.gov/data/t9_filtering.html`.



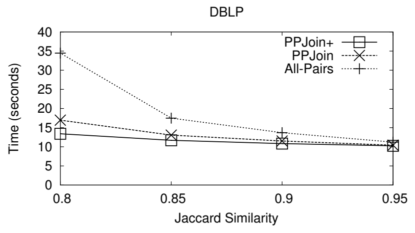
(a) Jaccard, DBLP, Candidate Size



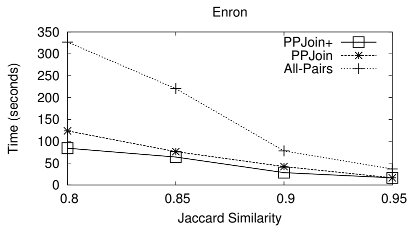
(b) Jaccard, Enron, Candidate Size



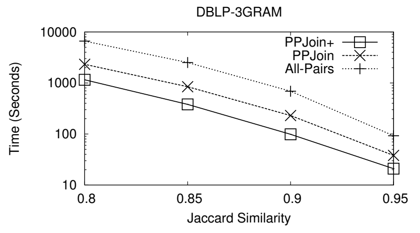
(c) Jaccard, DBLP-3GRAM, Candidate Size



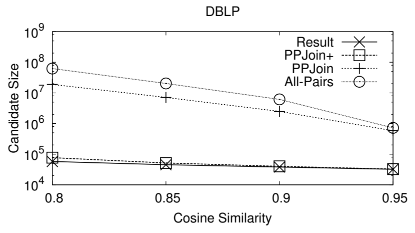
(d) Jaccard, DBLP, Time



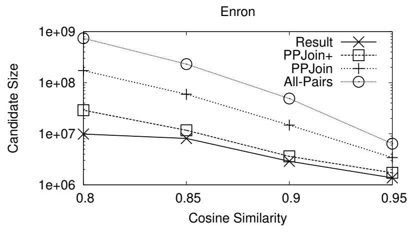
(e) Jaccard, Enron, Time



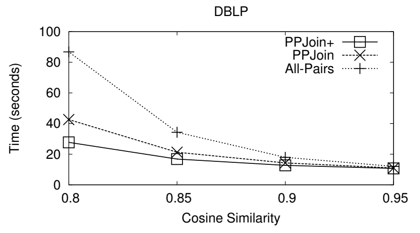
(f) Jaccard, DBLP-3GRAM, Time



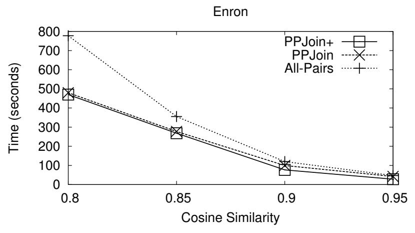
(g) Cosine, DBLP, Candidate Size



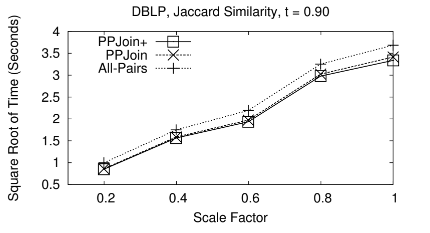
(h) Cosine, Enron, Candidate Size



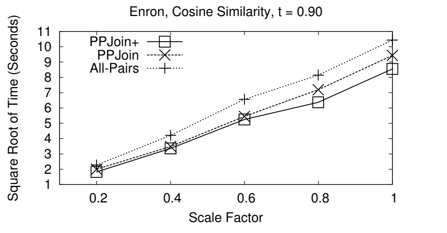
(i) Cosine, DBLP, Time



(j) Cosine, Enron, Time



(k) Jaccard, DBLP, Time



(l) Cosine, Enron, Time

Figure 3: Experimental Results

7.2 Jaccard Similarity

Candidate Size Figures 3(a) to 3(c) show the sizes of candidate pairs generated by the algorithms and the size of the join result on the DBLP, Enron, and DBLP-3GRAM datasets, with varying similarity thresholds from 0.80 to 0.95. Note that y-axis is in logarithm scale.

Several observations can be made:

- The size of the join result grows modestly when the similarity threshold decreases.
- All algorithms generate more candidate pairs with the decrease of the similarity threshold. Obviously, the candidate size of All-Pairs grows the fastest. ppjoin has a decent reduction on the candidate size of All-Pairs, as the positional filtering prunes many candidates. ppjoin+ produces the fewest candidates thanks to the additional suffix filtering.
- The candidate sizes of ppjoin+ are usually in the same order of magnitude as the sizes of the join result for a wide range of similarity thresholds. The only outlier is the Enron dataset, where ppjoin+ only produces modestly smaller candidate set than ppjoin. There are at least two reasons: (a) the average record size of the enron dataset is large; this allows for a larger initial Hamming distance threshold $H_{\max}$ for the suffix filtering. Yet we only use $\texttt{MAXDEPTH} = 2$ (for efficiency reasons; also see the Enron's true positive rate below). (b) Unlike other datasets used, an extraordinary high percentage of candidates of ppjoin *are* join results. The ratio of sizes of query result over candidate size by ppjoin algorithm is 38.1%, 4.9%, and 0.03% for Enron, DBLP, and DBLP-3GRAM, respectively. In other words, ppjoin has already removed the majority of false positive candidate pairs on Enron and hence it is hard for suffix filtering to further reduce the candidate set.

Running Time Figures 3(d) to 3(f) show the running time of all algorithms on the three datasets with varying Jaccard similarity thresholds.

In all the settings, ppjoin+ is the most efficent algorithm, followed by ppjoin. Both algorithms outperform the All-Pairs algorithm. The general trend is that the speed-up increases with the decrease of the similarity threshold. This is because (i) index construction, probing, and other overheads are more noticeable with a high similarity threshold, as the result is small and easy to compute. (ii) inverted lists in the indices are longer for a lower similarity threshold; this increases the candidate size which in turn slows down the All-Pairs algorithm as it does not have any other *additional* filtering mechanism. In contrast, many candidates are quickly discarded by failing the positional or suffix filtering used in ppjoin and ppjoin+ algorithms.

The speed-up that our algorithms can achieve against the All-Pairs algorithm is also dependent on the dataset. At the 0.8 threshold, ppjoin can achieve up to 3x speed-up against All-Pairs on both Enron and DBLP-3GRAM, and up to 2x speed-up on DBLP. At the same threshold, ppjoin+ can achieve 5x speed-up on DBLP-3GRAM, 4x speed-up on Enron, and 2.6x speed-up on DBLP. This trend can be explained as All-Pairs algorithm is not good at dealing with long records and/or a small token domain.

The performance between ppjoin and ppjoin+ is most substantial on DBLP-3GRAM, where filtering on the suffixes helps to improve the performance drastically. The reason why ppjoin+ has only modest performance gain over ppjoin on Enron is because 38% of the candidates are final results, hence the additional filtering employed in ppjoin+ won't contribute to much runtime reduction. The difference of the two is also moderate on DBLP. This is mainly because the average size of DBLP records is only 14 and even a brute-force verification using the entire suffix is likely to be fast, especially in modern computer architectures.

7.3 Cosine Similarity

We ran all three algorithms on the DBLP and ENRON datasets using the cosine similarity function, and plot the candidate sizes in Figures 3(g) to 3(h) and running times in Figures 3(i) to 3(j). For both metrics, the general trends are similar to those using Jaccard similarity. A major difference is that all algorithms now run slower for the same similarity threshold, mainly because a cosine similarity constraint is inherently looser than the corresponding Jaccard similarity constraint. At the 0.8 threshold, the speed-ups of the ppjoin and ppjoin+ algorithm is 2x and 3x on DBLP, respectively; on Enron, the speed-ups are 1.6 and 1.7, respectively.

7.4 Varying Data Sizes

We performed the similarity join using Jaccard similarity on subsets of the DBLP dataset and measured running times.[5] We randomly sampled about 20% to 100% of the records. We scaled down the data so that the data and result distribution could remain approximately the same. We show the *square root* of the running time with Jaccard similarity for the DLBP dataset and cosine similarity for the Enron dataset in Figures 3(k) and 3(l) (both thresholds are fixed at 0.9).

It is clear that the running time of all three algorithms grow quadratically. This is not surprising given the fact that the actual result size already grows quadratically (e.g., See Figure 1). Our proposed algorithms have demonstrated a slower growth rate than the All-Pairs algorithm for both similarity functions and datasets.

7.5 Near Duplicate Web Page Detection

We also investigate a specific application of the similarity join: near duplicate Web page detection. A traditional method is based on performing *approximate* similarity join on shingles computed from each record [6]. Later work proposed further approximations mainly to gain more efficiency at the cost of result quality.

Instead, we designed and tested three algorithms that perform *exact* similarity join on q-grams or shingles. (i) qp algorithm where we use the ppjoin+ algorithm to join directly on the set of 4-grams of each record. (ii) qa algorithm is similar to qp except that All-Pairs algorithm is used as the exact similarity join algorithm. (iii) sp algorithm where we use the ppjoin+ algorithm to join on the set of shingles.

The metrics we measured are: running times, *precision* and *recall* of the join result. Since algorithm qp returns exact answer based on the q-grams of the records, its result is a good candidate for the correct set of near duplicate documents. Hence, we define precision and recall as follows:

$$\text{Precision} = \frac{|R_{\mathsf{sp}}| \cap |R_{\mathsf{qp}}|}{|R_{\mathsf{sp}}|} \qquad \text{Recall} = \frac{|R_{\mathsf{sp}}| \cap |R_{\mathsf{qp}}|}{|R_{\mathsf{qp}}|}$$

where R_{x} is the set of result returned by algorithm x.

We show the results in Table 2 with varying similarity threshold values.

Table 2: Quality vs. Time Trade-off of Approximate and Exact Similarity Join

t	**Precision**	**Recall**	$time_{\mathsf{ap}}$	$time_{\mathsf{qp}}$	$time_{\mathsf{sp}}$
0.95	0.38	0.11	41.98	11.76	1.00
0.90	0.48	0.06	245.03	43.37	1.03
0.85	0.58	0.04	926.54	202.65	1.03
0.80	0.57	0.03	2467.31	775.00	1.05

Several observations can be made

- Shingling-based methods will mainly suffer from low recalls in the result, meaning that only a *small* fraction of truly similar Web pages will be returned. We manually examined some similar pairs missing from R_{sp} ($t = 0.95$), and most of the sampled pairs are likely to be near duplicates (e.g., they differ only by typos, punctuations, or additional annotations). Note that other variants of the basic shingling method, e.g., systematic sampling of shingles or super-shingling [6] were designed to trade result quality for efficiency, and are most likely to have even worse precision and recall values.

  In contrast, exact similarity join algorithms (qp or qa) have the appealing advantage of finding all the near duplicates given a similarity function.
- qp, while enjoying good result quality, requires longer running time. However, with reasonably high similarity threshold (0.90+), qp can finish the join in less than 45 seconds. On the other hand, qa takes substantially longer time to perform the same join.
- sp combines the shingling and ppjoin+ together and is extremely fast even for modest similarity threshold of 0.80. This method is likely to offer better result quality than, e.g., super-shingling, while still offering high efficiency.

In summary, ppjoin+ algorithm can be combined with q-grams or shingles to provide appealing alternative solutions to tackle the near duplicate Web page detection tasks.

[5]We also measured the candidate sizes (e.g., see Figure 1).

8. RELATED WORK

Near Duplicate Object Detection Near duplicate object detection has been studied under different names in several areas, including record linkage [27], merge-purge [19], data deduplication [23], name matching [4], just to name a few. [12] is a recent survey on this topic.

Similarity functions are the key to the near duplicate detection task. For text documents, edit distance [26] and Jaccard similarity on q-grams [17] are commonly used. Due to the huge size of Web documents, similarity among documents is evaluated by Jaccard or overlap similarity on small or fix sized sketches [5, 10]. Soundex is a commonly used phonetic similarity measures for names [22].

Exact Near Duplicate Detection Algorithm Existing methods for exact near duplicate detection usually convert constraints defined using one similarity function into equivalent or weaker constraints defined on another similarity measure. [17] converts edit distance constraints to overlap constraints on q-grams. Jaccard similarity constraints and 1/2-sided normalized overlap constraints can be converted to overlap constraints [24, 8]. Constraints on overlap, dice and Jaccard similarity measures can be coverted to constraints on cosine similarity [3]. [1] transforms Jaccard and edit distance constraints to Hamming distance constraints.

The techniques proposed in previous work fall into two categories. In the first category, exact near duplicate detection problems are addressed by inverted list based approaches [3, 8, 24], as discussed above. The second category of work [1] is based on the pigeon hole principle. The records are carefully divided into partitions and then hashed into signatures, with which candidate pairs are generated, followed by a post-filtering step to eliminate false positives.

Approximate Near Duplicate Object Detection Several previous work [6, 7, 10, 16] has concentrated on the problem of retrieving approximate answers to similarity functions. LSH (Locality Sensitive Hashing) [16] is a well-known approximate algorithm for the problem. It regards each record as a vector and generates signatures for each record with random projections on the set of dimensions. Broder et al. [6] addressed the problem of identifying near duplicate Web pages approximately by compressing document records with a sketching function based on min-wise independent permutations. The near duplicate object detection problem is also a generalization of the well-known nearest neighbor problem, which is studied by a wide body of work, with many approximation techniques considered by recent work [7, 13, 16, 21].

9. CONCLUSIONS

In this paper, we propose efficient similarity join algorithms by exploiting the ordering of tokens in the records. The algorithms provide efficient solutions for an array of applications, such as duplicate Web page detection on the Web. We show that positional filtering and suffix filtering are complementary to the existing prefix filtering technique. They successfully alleviate the problem of quadratic growth of candidate pairs when the data grows in size. We demonstrate the superior performance of our proposed algorithms to the existing prefix filtering-based algorithms on several real datasets under a wide range of parameter settings. The proposed methods can also be adapted or integrated with existing near duplicate Web page detection methods to improve the result quality or accelerate the execution speed.

REFERENCES

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

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1st edition edition, May 1999.

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

[4] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Sys.*, 18(5):16–23, 2003.

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

[6] 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.

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

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

[9] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *SIGMOD*, 2000.

[10] 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.

[11] J. G. Conrad, X. S. Guo, and C. P. Schriber. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *CIKM*, 2003.

[12] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[13] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, 2003.

[14] D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB*, 2003.

[15] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, 2005.

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

[17] 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.

[18] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.

[19] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[20] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *JASIST*, 54(3):203–215, 2003.

[21] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[22] R. C. Russell. Index, U.S. patent 1,261,167, April 1918.

[23] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

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

[25] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *KDD*, 2005.

[26] E. Ukkonen. On approximate string matching. In *FCT*, 1983.

[27] W. E. Winkler. The state of record linkage and current research problems. Technical report, U.S. Bureau of the Census, 1999.