Efficient Similarity Joins for Near Duplicate Detection

Chuan Xiao, The University of New South Wales
Wei Wang, The University of New South Wales
Xuemin Lin, The University of New South Wales & East China Normal University
Jeffrey Xu Yu, Chinese University of Hong Kong
Guoren Wang, Northeastern University, China

With the increasing amount of data and the need to integrate data from multiple data sources, one of the challenging issues is to identify *near duplicate* records efficiently. In this paper, we focus on efficient algorithms to find pair of records such that their similarities are no less than 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 token ordering information; they are integrated into the existing methods and drastically reduce the candidate sizes and hence improve the efficiency. We have also studied the implementation of our proposed algorithm in stand-alone and RDBMS-based settings. Experimental results show our proposed algorithms can outperforms previous algorithms on several real datasets.

1. INTRODUCTION

Near duplicate data is one of the issues accompanying the rapid growth of data on the Internet and the growing need to integrating data from heterogeneous sources. As a concrete example, a sizeable percentage of the Web pages are found to be near-duplicates by several studies [Broder et al. 1997; Fetterly et al. 2003; Henzinger 2006]. These studies suggest that around 1.7% to 7% of the Web pages visited by crawlers are near duplicate pages. 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.

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 [Fetterly et al. 2003; Conrad et al. 2003; Henzinger 2006].

— Many Web mining applications rely on the ability to accurately and efficiently identify near-duplicate objects. They include document clustering [Broder et al. 1997], finding replicated Web collections [Cho et al. 2000], detecting plagiarism [Hoad and Zobel 2003], community mining in a social network site [Spertus et al. 2005], collaborative filtering [Bayardo et al. 2007] and discovering large dense graphs [Gibson et al. 2005].

A quantitative way to define 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 no less than a given threshold. Throughout this paper, we will mainly focus on the Jaccard similarity; extensions to other similarity measures such as Overlap and cosine similarities are given in Section 6.1.

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 challenges, 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 [Broder 1997; Charikar 2002; Chowdhury et al. 2002].

Recently, researchers started to investigate algorithms that compute the similarity join for some common similarity/distance functions *exactly*. Proposed methods include inverted index-based methods [Sarawagi and Kirpal 2004], prefix filtering-based methods [Chaudhuri et al. 2006; Bayardo et al. 2007] and signature-based methods [Arasu et al. 2006]. Among them, the All-Pairs algorithm [Bayardo et al. 2007] 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 that works for several commonly used similarity or distance functions, such as Jaccard, cosine similarities, Hamming and edit distances. 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 discuss several implementation alternatives of the proposed similarity join algorithms on relational database systems. We conduct an extensive experimental study using several real datasets on both stand-alone and DBMS implementation, and demonstrate that the proposed algorithms outperform previous algorithms. We also show that the new algorithms can be adapted or combined with existing approaches to produce better quality results 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 by integrating positional filtering method on the prefixes and suffixes of the records. Generalization to other similarity measures is presented in Section 6. Several alternative implementations of similarity join algorithms on relational

databases are discussed in Section 7. We present our experimental results in Section 8. Related work is covered in Section 9 and Section 10 concludes the paper.

2. PROBLEM DEFINITION AND PRELIMINARIES

2.1. Problem Definition

We define a record as a *set* of tokens drawn from a finite universe $\mathcal{U} = \{w_1, w_2, \ldots\}$. 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 [Broder 1997]. 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 [Chaudhuri et al. 2006]. This conversion enables us to perform multiset intersections and capture the similarity between multisets with commonly used similarity functions such as Jaccard [Theobald et al. 2008]. 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. Sorting tokens in this order is a heuristic to speed up similarity joins [Chaudhuri et al. 2006]. A record x can also be represented as a $|\mathcal{U}|$-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 follong distance functions are considered in this work.

— *Hamming distance* between x and y is defined as the size of their symmetric difference: $H(x, y) = |(x - y) \cup (y - x)|$.
— *Edit distance*, also known as *Levenshtein distance*, is defined between two strings. It 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 [Ukkonen 1983]. It is usually converted into a *weaker* constraint on the overlap between the q-gram sets of the two strings [Gravano et al. 2001; Li et al. 2008; Xiao et al. 2008].

Note that the above similarity and distance functions are inter-related. We discuss some important relationships in Section 2.2, and others in Section 6.1.

---

[1]For the ease of illustration, we do not normalize the overlap similarity to $[0, 1]$.

In this paper, we will focus on the Jaccard similarity, a common function for defining similarity between sets. Extension of our algorithms to handle other similarity or distance functions appears in Section 6.1. Therefore, in the rest of the paper, $sim(x, y)$ defaults to $J(x, y)$, unless otherwise stated. In addition, we will focus on in-memory implementation when describing algorithms. The disk-based implementation using database systems will be presented in Section 7.

Example 2.1. *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, as we convert each subsequent occurrence of the same token as a new token. 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 based on the overlap similarity or the Hamming distance as follows:

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

PROOF. By definition,

$$J(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{O(x, y)}{|x| + |y| - O(x, y)}.$$

Since $J(x, y) \geq t$, we know

$$\frac{O(x, y)}{|x| + |y| - O(x, y)} \geq t \iff (1 + t)O(x, y) \geq t(|x| + |y|),$$

$$\iff O(x, y) \geq \frac{t}{1 + t} \cdot (|x| + |y|).$$

□

$$O(x, y) \geq \alpha \iff H(x, y) \leq |x| + |y| - 2\alpha. \tag{2}$$

PROOF. By definition,

$$H(x,y) = |(x-y) \cup (y-x)| = |x| - O(x,y) + |y| - O(x,y).$$

Since $O(x,y) \geq \alpha$, we know

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

□

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

$$J(x,y) \geq t \Longrightarrow t \cdot |x| \leq |y|, \tag{3}$$

and by applying Equation 1,

$$J(x,y) \geq t \Longrightarrow O(x,y) \geq t \cdot |x|. \tag{4}$$

3. PREFIX FILTERING BASED METHODS

A naïve algorithm to compute similarity join results 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 $O(n^2)$, where n is the number of records.

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 indexes [Baeza-Yates and Ribeiro-Neto 1999]. An inverted index maps a token w to a list of record identifiers that contain w. After inverted indexes for all tokens in the record set are built, we can scan each record x, probing the index 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 indexes essentially requires keeping the state for 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, [Sarawagi and Kirpal 2004] employs sequential access on short inverted lists but switches to binary search on the $\alpha - 1$ longest inverted lists, where α is the Overlap similarity threshold.

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* [Chaudhuri et al. 2006, Lemma 1] rephrased below.

LEMMA 3.1 ((PREFIX FILTERING PRINCIPLE)[CHAUDHURI ET AL. 2006]). *Given 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 indexes 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 indexes 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. From Equation 4, 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 indexes need to be built and accessed (by an 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. [Chaudhuri et al. 2006] considers implementing the prefix filtering method on top of a commercial database system, while [Bayardo et al. 2007] further improves the method by utilizing several other filtering techniques in candidate generation phase and verification phase.

Example 3.2. *Consider a collection of four canonicalized records based on the document frequency ordering, and the Jaccard similarity threshold of $t = 0.8$:*

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

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 [Bayardo et al. 2007] 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 length filtering *[Arasu et al. 2006].*

4. POSITIONAL FILTERING

We now describe our proposal 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

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 of a token in a canonicalized record (starting from 1). We illustrate the observation in the following example.

Example 4.1. *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 4.2.

LEMMA 4.2 ((POSITIONAL FILTERING PRINCIPLE)). *Given 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 $x_l(w) = x[1 \mathinner{.\,.} i]$ *and the* right partition $x_r(w) = x[(i+1) \mathinner{.\,.} |x|]$. *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 [Bayardo et al. 2007], to combine positional filtering and prefix-filtering. Like the All-Pairs algorithm, the ppjoin algorithm takes as input a collection of canonicalized records already sorted in the ascending order of their sizes. It then sequentially scans each record x, finds candidates that intersects x's prefix ($x[1 \mathinner{.\,.} p]$, Line 5) and accumulates the overlap in a hash map A (Line 12). It makes use of an inverted index built on-the-fly, i.e., I_w returns the postings list associated with w, and each element in the list is of the form rid, indicating that the record rid's prefix contains w. 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 [Bayardo et al. 2007] for further details on the All-Pairs algorithm.

Now we will elaborate on several novel aspects of our extension: (i) the inverted indexes 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. Each element in the postings list of w (i.e., I_w) is of the form (rid, pos), indicating that the *pos*-th token in record rid's prefix is w. In Lines 9–14, we consider the positions of the common token in both x and y (denoted i and j), 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 $A[y] + \mathsf{ubound}$ is smaller than α, we can prune the current candidate y (Line 14).

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**: S is the set of all pairs of records $\langle x, y \rangle$, such that $x \in R, y \in R$, $x.rid > y.rid$, and $sim(x, y) \geq t$

1   $S \leftarrow \emptyset$;
2   $I_w \leftarrow \emptyset \ (1 \leq w \leq |\mathcal{U}|)$;   /* initialize inverted index */;
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           $\mathsf{ubound} \leftarrow 1 + \min(|x| - i, |y| - j)$;
11   **if** $A[y] + \mathsf{ubound} \geq \alpha$ **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, α);
17   **return** S

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         $\mathsf{ubound} \leftarrow A[y] + |x| - p_x$;
7   **if** $\mathsf{ubound} \geq \alpha$ **then**
8            $O \leftarrow O + \left| x[(p_x + 1) \mathinner{.\,.} |x|] \bigcap y[(A[y] + 1) \mathinner{.\,.} |y|] \right|$;
9   **else**
10        $\mathsf{ubound} \leftarrow A[y] + |y| - p_y$;
11   **if** $\mathsf{ubound} \geq \alpha$ **then**
12           $O \leftarrow O + \left| x[(A[y] + 1) \mathinner{.\,.} |x|] \bigcap y[(p_y + 1) \mathinner{.\,.} |y|] \right|$;
13   **if** $O \geq \alpha$ **then**
14        $S \leftarrow S \cup (x, y)$;

Table I. Candidate Size (DBLP, Jaccard)

t	All-Pairs	ppjoin	ppjoin+ (MAXDEPTH $= 2$)	**Join Result**
0.95	135,470	132,190	1,492	90
0.90	756,323	571,147	5,053	1,530
0.85	2,686,012	1,286,909	12,505	4,158
0.80	7,362,912	3,040,972	30,443	8,112

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 immediate 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.

As shown in Algorithm 1, the ppjoin algorithm can be implemented as a batch join, i.e., loading inputs and building indexes on-the-fly. It can be also implemented as an indexed join, where the inverted indexes for prefixes are precomputed. Since prefixes under lower similarity thresholds always subsume prefixes under higher similarity thresholds, we can precompute and index prefixes under a similarity threshold $t_{\min}$, and use it for all similarity joins with threshold $t \geq t_{\min}$. For example, Section 7 discuss this option in more detail.

It is worth mentioning that positional filtering will benefit Jaccard and cosine similarities, but won't be useful for the Overlap similarity. We will show how to extend the positional filtering condition to other similarity measures in Section 6.1.

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 I.

4.3. The Indexing Prefix and the Probing Prefix

Another significant optimization is the introduction of an index reduction technique. We illustrate the basic idea in the following example.

Example 4.3. *Consider the following two records x and y, and the similarity threshold of 0.8. Tokens "?" indicate that we have not accessed those tokens and do not know their contents yet.*

$$x = [A, B, ?, ?, ?]$$
$$y = [B, ?, ?, ?, ?]$$

The prefix length of x is 2. If y contains the token B but not A, the maximum possible similarity of the pair $\langle x, y \rangle$ is at most $\frac{4}{5+5-4} = 0.67$. Therefore this pair cannot meet the similarity threshold though they share a common token B in their prefix.

This suggests that we do not need to index token B for x. It was first used implicitly in [Bayardo et al. 2007] for cosine similarity function, and was extended to other similarity

functions in [Xiao et al. 2009]. We formally state this observation in the following lemma
that can further reduce the number of tokens to be indexed and hence accessed,

LEMMA 4.4. *Given a record x, we only need to index its $l_i = |x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$-prefix
(and use $l_p = |x| - \lceil t|x| \rceil + 1$ prefix for inverted index probing) for Algorithm 1 to produce
correct join result.*

PROOF. Based on the definition of the Jaccard similarity, we know that the Jaccard
similarity increases when $O(x, y)$ increases, as

$$J(x, y) = \frac{|x \cap y|}{|x \cup y|} = \frac{O(x, y)}{|x| + |y| - O(x, y)}$$

We will find an upper bound for $O(x, y)$ below, which in turn induces an upper bound on
the Jaccard similarity.
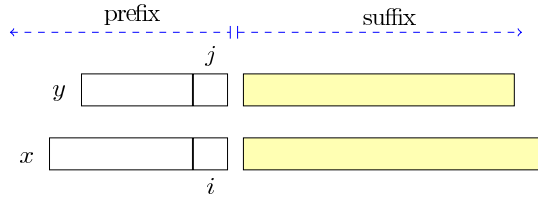
Fig. 1. Indexing Prefix and Probing Prefix

Consider two records x and y, and $|x| \geq |y|$. We also know that $t \cdot |x| \leq |y|$ based on
Equation (3).

Suppose we use the first j tokens of y as its prefix and use the first i tokens of x as its
prefix. If there is no common token in $x[1 .. i]$ and $y[1 .. j]$, then the only overlap comes
from the suffix of one record with another record.

We distinguish two possible cases, based on the ordering of the last tokens in x's and y's
prefixes.

Case 1 when $x[i] \leq y[j]$. In this case, $O(x, y) \leq \min(|x| - i, |y|)$.
   (a) If $|x| - i \leq |y|$, $O(x, y) \leq |x| - i$. We then have

$$J(x, y) \leq \frac{|x| - i}{|x| + |y| - |x| + i} \leq \frac{|x| - i}{t|x| + i}.$$

   Let the right hand side be less than t, we have

$$\frac{|x| - i}{|x| + i} < t \Longleftrightarrow i > (1 - t)|x|.$$

   It can be verified that $|x| - i$ is indeed less than $|y|$.
   (b) If $|x| - i > |y|$, $O(x, y) \leq |y|$. We then have

$$J(x, y) \leq \frac{|y|}{|x| + |y| - |y|} = \frac{|y|}{|x|}.$$

   In this subcase, if $i \geq |x| - |y| + 1$, we have $J(x, y) < t$. This gives a same lower
   bound on the choice of i.
   Therefore, in this case, when $i \geq |x| - \lceil t|x| \rceil + 1$, $J(x, y) < t$ if there is no intersection in
   x's and y's prefixes (regardless of j's value).

Case 2 when $x[i] > y[j]$. In this case, $O(x,y) \leq \min(|y|-j, |x|) = |y|-j$.
We then have

$$J(x,y) \leq \frac{|y|-j}{|x|+|y|-|y|+j} \leq \frac{|y|-j}{|y|+j}.$$

Let the right hand side less than t, we have

$$\frac{|y|-j}{t|y|+j} < t \iff j > \frac{1-t}{1+t}|y|.$$

Therefore, in this case, when $j \geq |y| - \lceil \frac{2t}{1+t}|y| \rceil + 1$, $J(x,y) < t$ if there is no intersection in x's and y's prefixes (regardless of i's value).

□

Remark 4.5. The root of this improvement comes from (1) the fact that records are sorted in increasing order of lengths (thus we can only use indexing prefix for short records and probing prefix for longer record) and (2) the fact that Jaccard similarity is relative to the sizes of both records.

Remark 4.6. The above prefixes lengths are for the worst case scenario and are not tight otherwise, however, our positional filtering effectively removes all the spurious candidates by taking the actual value of x and y (and is thus optimal if only this information is given).

We may extend the prefix length by k tokens, and then require a candidate pair having no less than $k+1$ common tokens.

COROLLARY 4.7. *Consider using a $l_i + k$ indexing prefix and $l_p + k$ probing prefix. If two records do not have at least $k+1$ common tokens in their respective prefixes, the Jaccard similarity between them is less than t.*

This optimization requires us to change Line 15 in Algorithm 1 such that it only indexes the current token w if the current token position i is no larger than $|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$.

In order not to abuse the term "prefix", we denote "prefix" by default *probing prefix* in later sections unless otherwise specified. We also exclude the the optimization using indexing prefixes for the ease of illustration in later sections, but integrate it into the implementations in our experiments.

5. SUFFIX FILTERING

In this section, we first motivate the need of additional 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. Quick Growth of the Candidates

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 2. 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).
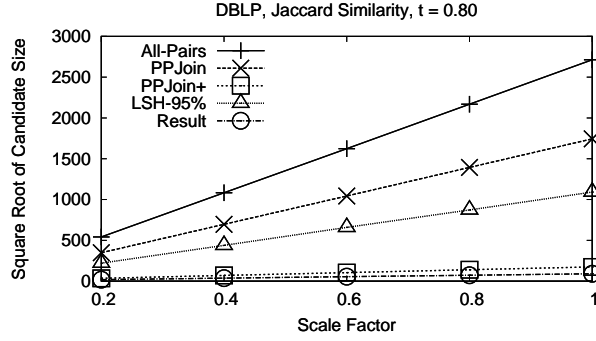
DBLP, Jaccard Similarity, t = 0.80

Fig. 2. Quadratic Growth of Candidate Size

5.2. Generalization of Positional Filtering to Suffixes

Given the empirical observation about the quadratic growth rate of the candidate size, it is desirable to find an additional pruning method in order to alleviate the problem and 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 that survives after positional filtering, where the term "suffix" denotes the tokens not included in the prefix of a record. 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 indexes or partial overlap results? (ii) how to find 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 a 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, $|y| \leq |x|$. We can derive the following upper bound in terms of the Hamming distance of their suffixes:

$$H(x_s, y_s) \leq H_{\max} = 2|y| - 2\lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil - (\lceil t \cdot |y| \rceil - \lceil t \cdot |x| \rceil). \qquad (5)$$

PROOF. According to Jaccard similarity, the overlap between x and y is at least $\frac{t}{1+t} \cdot (|x| + |y|)$. Therefore the Hamming distance of x and y

$$H(x, y) \leq |x| + |y| - 2\lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil.$$

The Hamming distance between x's and y's prefixes is at least the difference of the lengths of the two prefixes, i.e.,

$$H(x_p, y_p) \geq \lfloor ((1-t) \cdot |x| + 1) \rfloor - \lfloor ((1-t) \cdot |y| + 1) \rfloor,$$
$$= \lfloor (1-t) \cdot |x| \rfloor - \lfloor (1-t) \cdot |y| \rfloor.$$

Therefore, we have

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

□

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,

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

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 later probings more effective. We illustrate this idea in the example below.

Example 5.1. *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 $\mathrm{abs}(3 - 4) + \mathrm{abs}(3 - 2) = 2$.

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

$$\mathrm{abs}(|x_{ll}| - |y_{ll}|) + \mathrm{abs}(|x_{lr}| - |y_{lr}|) + \mathrm{abs}(|x_r| - |y_r|)$$
$$= \mathrm{abs}(1 - 2) + \mathrm{abs}(3 - 1) + \mathrm{abs}(2 - 3) = 4.$$

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

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 (Line 9–10). Then we call the SuffixFilter function recursively first on the left and then on the right partitions (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 [Sarawagi and Kirpal 2004]. 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 partition token w in y, due to the Hamming distance constraint. We illustrate this using the following example.

Example 5.2. *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 5.1 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 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 $H \leftarrow$ abs$(|x_l| - |y_l|) +$ abs$(|x_r| - |y_r|) +$ diff;

10 **if** $H > H_{\max}$ **then**

11 | **return** H

12 **else**

13 | $H_l \leftarrow$ SuffixFilter$(x_l, y_l, H_{\max} -$ abs$(|x_r| - |y_r|) -$ diff, $d + 1)$;

14 | $H \leftarrow H_l +$ abs$(|x_r| - |y_r|) +$ diff;

15 | **if** $H \leq H_{\max}$ **then**

16 | $H_r \leftarrow$ SuffixFilter$(x_r, y_r, H_{\max} - H_l -$ diff, $d + 1)$;

17 | **return** $H_l + H_r +$ diff

18 | **else**

19 | **return** H

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

Input : An array of tokens s, a token w, left and right bounds of searching range l, r
Output: Two partitions 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 larger than w in the global ordering within $s[l\mathinner{.\,.}r]$;
5   $s_l \leftarrow s[1\mathinner{.\,.}p-1]$;
6   **if** $s[p] = w$ **then**
7     $s_r \leftarrow s[(p+1)\mathinner{.\,.}|s|]$; diff $\leftarrow 0$;   /* skip the token w */;
8   **else**
9     $s_r \leftarrow s[p\mathinner{.\,.}|s|]$; diff $\leftarrow 1$;
10 **return** $(s_l, s_r, 1, \text{diff})$

Algorithm 5: Replacement of Line 12 in Algorithm 1

---

1 **if** $A[y] = 0$ **then**   /* only true if y first becomes a candidate for x */
2    $H_{\max} \leftarrow |x| + |y| - 2 \cdot \lceil \frac{t}{1+t} \cdot (|x|+|y|) \rceil - (i+j-2)$;
3    $H \leftarrow \mathsf{SuffixFilter}(x[(i+1)\mathinner{.\,.}|x|], y[(j+1)\mathinner{.\,.}|y|], H_{\max}, 1)$;
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, and put it *after* the first invocation of positional filtering for this pair. This is because (1) suffix filtering probes the unindexed part of the records, and is relatively expensive to carry out; (2) candidate sizes after applying suffix filtering are usually drastically reduced, and in the same order of magnitude as the sizes of join result for a wide range of similarity thresholds (See Table I, Figure 2, and the experiment report in Section 8). 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 $x[1\mathinner{.\,.}(i-1)] \cap y[1\mathinner{.\,.}(j-1)] = \emptyset$ 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 I and Figure 2.

It is worth mentioning that the reduction of candidate size won't equally contribute to the reduction on the running time of similarity join. This is because that suffix filtering performs binary searches to find tokens in the suffix, and the overhead increases with the level of recursion. Currently, the maximum level of recursion, MAXDEPTH, is determined

heuristically, as it depends on the data distribution and similarity threshold. We determine its optimal value by running the algorithm on a sample of the dataset with different values and choosing the one giving the best performance. A larger value of `MAXDEPTH` reduces the size of the candidates to be verified, yet incurs overhead for the pruning. For all the datasets we have tested, the optimal `MAXDEPTH` value ranges from 2 to 5. In Section 8.1.6, we will study the effect of the parameter `MAXDEPTH` using experimental evaluation to find the overall most efficient algorithm.

6. EXTENSIONS

6.1. 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 $\alpha = (\max(|u|, |v|) + q - 1) - q\delta$ [Gravano et al. 2001].

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 $|x| + |y| - 2 \left\lceil t \cdot \sqrt{|x| \cdot |y|} \right\rceil - (i + j - 2)$.

6.2. Generalization to the Weighted Case

To adapt the ppjoin and ppjoin+ algorithms to the weighted case, we mainly need to modify the computation of the prefix length. In the following, we use weighted Jaccard similarity (defined below) as the example, and illustrate important changes.

$$J^w(x, y) = \frac{\sum_{w \in |x \cap y|} weight(w)}{\sum_{w \in |x \cup y|} weight(w)}$$

The binary Jaccard similarity we discussed before is just a special case when all the weights are 1.0.

---

[2]These are the same bounds obtained in [Bayardo et al. 2007].

Reconsider Example 2.1 and assume the weights of the tokens as follows:

Word	yes	as	soon	as$_1$	possible	please
Token	A	B	C	D	E	F
Weight	0.3	0.2	0.6	0.5	0.7	0.4

Then the weighted Jaccard similarity between x and y is:

$$J^w(x, y) = \frac{0.2 + 0.6 + 0.5 + 0.7}{0.3 + 0.2 + 0.6 + 0.5 + 0.7 + 0.4} = 0.74$$

To choose the global ordering of tokens, one option is to sort the tokens by decreasing order of weight. In addition, records can be sorted to the total weight of each record (denoted as σ_x) rather than its length, i.e., $\sigma_x = \sum_{i=1}^{|x|} weight(x[i])$.

The probing prefix length of a record x is

$$l_p = \min \left\{ j \left| \frac{\sigma_x - \sum_{i=1}^{j} weight(x[i])}{\sigma_x} < t \right. \right\}$$

It can be shown that if there is no overlap within the prefix, the maximum weighted Jaccard similarity value that can be achieved will be less than t.

The indexing prefix length for x is

$$l_i = \min \left\{ j \left| \frac{\sigma_x - \sum_{i=1}^{j} weight(x[i])}{\sigma_x + \sum_{i=1}^{j} weight(x[i])} < t \right. \right\}$$

In order to apply the positional filtering, we index the sum of the weights of unseen tokens in the inverted list, instead of the token's position. Formally, if a token $x[i]$ is indexed, then $\sigma_{x[i\,..|x|]} = \sum_{j=i}^{|x|} weight(x[j])$ is calculated and stored as if it is the positional information. The positional filtering test in Lines 9 – 14 in Algorithm 1 is replaced by Algorithm 6.

Algorithm 6: Replacement of Lines 9 – 14 in Algorithm 1

---

1 $\alpha \leftarrow \lceil \frac{t}{1+t}(\sigma_x + \sigma_y) \rceil$;
2 ubound $\leftarrow \min(\sigma_{x[i\,..|x|]}, \sigma_{y[j\,..|y|]})$;
3 **if** $A[y] + $ ubound $\geq \alpha$ **then**
4 $A[y] \leftarrow A[y] + weight(w)$; /* $A[y]$ is the current weighted overlap of x and y */;
5 **else**
6 $A[y] \leftarrow 0$;   /* prune y */;

Next, we develop the suffix filtering for the weighted Jaccard similarity function. The algorithm framework is the same as the binary version, except that some of the values need to be converted to weighted forms. The Hamming distance threshold $H_{\max}$ is replaced by the weighted Hamming distance threshold

$$H_{\max}^w = \sigma_x + \sigma_y - 2\lceil \frac{t}{1+t} \cdot (\sigma_x + \sigma_y) \rceil - (\sigma_{x[1\,..i-1]} + \sigma_{y[1\,..j-1]})$$

The pseudocode of suffix filtering for weighted Jaccard similarity is given in Algorithm 7. It has the following major modifications:

Algorithm 7: SuffixFilterWeighted($x, y, H_{\max}^w, d$)

Input : Two set of tokens x and y, the maximum allowable weighted hamming
            distance $H_{\max}^w$ between x and y, and current recursive depth d
Output: The lower bound of weighted hamming distance between x and y

1 **if** $d > $ MAXDEPTH **then**
2 | **if** $|x| \geq |y|$ **then**
3 | | **return** $weight(x[|x|]) \cdot (|x| - |y|)$
4 | **else**
5 | | **return** $weight(y[|y|]) \cdot (|y| - |x|)$

6 $mid \leftarrow \lceil \frac{|y|}{2} \rceil$; $w \leftarrow y[mid]$;
7 $o \leftarrow \lceil \frac{H_{\max}^w}{weight(w)} \rceil$;
8 $(y_l, y_r, f, \mathsf{diff}) \leftarrow \mathsf{Partition}(y, w, mid, mid)$;
9 $(x_l, x_r, f, \mathsf{diff}) \leftarrow \mathsf{Partition}(x, w, mid - o, mid + o)$;
10 **if** $f = 0$ **then**
11 | **return** $H_{\max}^w + 1$
12 **if** $|x_l| \geq |y_l|$ **then**
13 | $H_l^w \leftarrow weight(x_l[|x_l|]) \cdot (|x_l| - |y_l|)$;
14 **else**
15 | $H_l^w \leftarrow weight(y_l[|y_l|]) \cdot (|y_l| - |x_l|)$;
16 **if** $|x_r| \geq |y_r|$ **then**
17 | $H_r^w \leftarrow weight(x_r[|x_r|]) \cdot (|x_r| - |y_r|)$;
18 **else**
19 | $H_r^w \leftarrow weight(y_r[|y_r|]) \cdot (|y_r| - |x_r|)$;
20 $H^w \leftarrow H_l^w + H_r^w + \mathsf{diff} \cdot weight(w)$;
21 **if** $H^w > H_{\max}^w$ **then**
22 | **return** H^w
23 **else**
24 | $H_l^w \leftarrow \mathsf{SuffixFilterWeighted}(x_l, y_l, H_{\max}^w - H_r^w - \mathsf{diff} \cdot weight(w), d+1)$;
25 | $H^w \leftarrow H_l^w + H_r^w + \mathsf{diff} \cdot weight(w)$;
26 | **if** $H^w \leq H_{\max}^w$ **then**
27 | | $H_r^w \leftarrow \mathsf{SuffixFilterWeighted}(x_r, y_r, H_{\max}^w - H_l^w - \mathsf{diff} \cdot weight(w), d+1)$;
28 | | **return** $H_l^w + H_r^w + \mathsf{diff} \cdot weight(w)$
29 | **else**
30 | | **return** H^w

— *Lines 1 – 5*: If the recursive depth d is greater than MAXDEPTH, we compare the length of x and y, and select the longer one. Its last token has the least weight. The weight is multiplied by the length difference of x and y, and then returned as a lower bound of weighted hamming distance. Note that the inputs to the algorithm x and y can be not only records but also partitions, therefore we are unable to obtain the exact value of the weighted sum of x's or y's tokens unless it is calculated on-the-fly or is calculated beforehand. We choose to return the lower bound as this will save time and space costs, though the bound is generally not as tight as the exact value.

— *Lines 7 – 9*: New search ranges are used to perform binary search efficiently for weighted cases.

— *Lines 12 – 20*: We estimate the lower bound within the left (or right) partition by multiplying the weight of the last token in the longer partition and the length difference.

7. IMPLEMENTING SIMILARITY JOINS ON RELATIONAL DATABASE SYSTEMS

In this section, we discuss and compare several alternatives to implement the similarity join algorithms (All-Pairs, ppjoin, and ppjoin+) on relational database management systems. Such implementations are disk-based, and of interest to both academia [Gravano et al. 2001] and industry [Chaudhuri et al. 2006]. We focus on the case of self-join and discuss necessary modifications to accommodate non-self-join at the end of this section.

We consider an input relation R with the following schema:

```
CREATE TABLE R (
   rid    INTEGER        PRIMARY KEY,
   len    INTEGER,
   toks   VARRAY(2048)
);
```

where `rid` is the identifier of a record, `len` is the length of this record, and `toks` is a variable-length array of token identifiers sorted in the increasing *df* order.[3]

The naïve scheme can be implemented as the following SQL query:

```
   SELECT    R1.rid, R2.rid
   FROM      R R1, R R2
   WHERE     R1.rid1 < R2.rid
     AND     SIM(R1.text, R2.text) >= t
```

where `SIM` is the similarity function implemented in UDF.

It is prohibitively expensive for large datasets, partly because the query entails (almost half of) a cross self join on R. Although this scheme can be improved by keeping track of the lengths of each record and imposing a length difference constraint in the query, it still has prohibitively high cost in practice.

In order to implement the prefix-filtering in SQL, there are few alternatives, which will be discussed below.

7.1. All-Pairs

In order to incorporate the prefix filtering, a prefix table, `PrefixR`, is generated from relation R, with the following schema:

```
CREATE TABLE PrefixR (
   rid    INTEGER,
   len    INTEGER,
   tid    INTEGER,
   PRIMARY KEY (tid, rid)
);
```

where `rid` is the identifier of a record, `len` is the length of a record, and `tid` is a token that appears in the record identified by `rid`. Note that all tokens in a record are ordered by increasing *df* values and the prefix of appropriate length are recorded in the `PrefixR` table. Note that, unlike our non-DBMS implementation, we do not require the input records sorted on their lengths. Instead, length filtering in the SQL query automatically enforces such a constraint.

If the similarity threshold is known in advance, we can have a *precise* prefix table, where the (probing) prefix of length $|x| - \lceil t|x| \rceil + 1$ is record in Table `PrefixR`. We consider generic version in Section 8.2.

The SQL queries can be conceptually divided into two parts: (a) generating the candidate pairs and (b) verifying them.

---

[3]The maximum capacity of `VARRAY` can be adjusted accordingly or other appropriate attribute type for strings can be used.

```
CREATE VIEW CANDSET AS
    SELECT    DISTINCT PR1.rid AS rid1, PR2.rid As rid2
    FROM      PrefixR PR1, PrefixR PR2
    WHERE     PR1.rid < PR2.rid
      AND     PR1.tid = PR2.tid
      AND     PR1.len >= CEIL(t * PR2.len)


    SELECT    R1.rid, R2.rid
    FROM      R R1, R R2, CANDSET C
    WHERE     C.rid1 = R1.rid
      AND     C.rid2 = R2.rid
      AND     VERIFY(R1.toks, R2.toks, t) = 1
```

This scheme results in significant improvement over the naïve scheme for two main reasons: the reduction of candidate set due to the prefix filtering, and the use of length filtering that reduces the complexity of generating the candidates.

We note that this scheme is similar to the scheme proposed in [Chaudhuri et al. 2006] with the following difference:

— The main difference is the use of length filtering, which is not possible in [Chaudhuri et al. 2006] as they mainly consider the overlap constraint.
— The rid order in our scheme implies the length order.

7.2. Implementing ppjoin and ppjoin+

To achieve the best performance, we need an enhanced version of the `PrefixR` table by including an additional attribute `pos`. The SQL is listed below:

```
CREATE TABLE PrefixR (
    rid     INTEGER,
    len     INTEGER,
    tid     INTEGER,
    pos     INTEGER,
    PRIMARY KEY (tid, rid)
);
```

ppjoin can be implemented as

```
1    CREATE VIEW CANDSET AS
2        SELECT    DISTINCT PR1.rid AS rid1, PR2.rid As rid2
3        FROM      PrefixR PR1, PrefixR PR2
4        WHERE     PR1.rid < PR2.rid
5          AND     PR1.tid = PR2.tid
6          AND     PR1.len >= CEIL(t * PR2.len)
7          AND     PR1.len - PR1.pos >= CEIL(PR1.len * 2 * t / (1+t))
8          AND     DECODE( SIGN((PR1.len - PR1.pos) - (PR2.len - PR2.pos)),
9                            -1,
10                           (PR1.len - PR1.pos),
11                           (PR2.len - PR2.pos)
12                  ) >=
13                CEIL( (PR1.len + PR2.len) * t / (1+t) );


    SELECT    R1.rid, R2.rid
    FROM      R R1, R R2, CANDSET C
    WHERE     C.rid1 = R1.rid
      AND     C.rid2 = R2.rid
      AND     VERIFY(R1.len, R1.toks, R2.len, R2.toks, t) = 1
```

Note that

— The positional filtering is implemented in Lines 8–13. A subtlty is that the positional filtering is only correct on the first common token in a candidate pair's prefixes. However, this implementation does not affect the correctness of the algorithm.

— Line 7 imposes the restriction the tokens in the shorter record must be from the indexing prefix. Therefore, we only consider joins between indexing prefixes and probing prefixes.

For ppjoin+, the only difference is in the second SQL statement, where `SUFFIX_FILTER` is the PL/SQL code implementing the suffix filtering.

```
SELECT    R1.rid, R2.rid
FROM      R R1, R R2, CANDSET C
WHERE     C.rid1 = R1.rid
   AND    C.rid2 = R2.rid
   AND    SUFFIX_FILTER(R1.toks, R2.toks, R1.len, R2.len, t) = 1
   AND    VERIFY(R1.len, R1.toks, R2.len, R2.toks, t) = 1
```

7.2.1. Generic Prefix Table. The above scheme requires a prefix table for every similarity threshold. Observing that the prefixes under lower similarity thresholds always subsumes the prefixes under higher similarity thresholds, we can use only one prefix table built for similarity threshold $t_{\min}$ and use it for all similarity joins with threshold $t \geq t_{\min}$. With the generic prefix table, we use the following SQL statement to generate the candidate pairs:

```
1   CREATE VIEW CANDSET AS
2     SELECT   DISTINCT PR1.rid AS rid1, PR2.rid As rid2
3     FROM     PrefixR PR1, PrefixR PR2
4     WHERE    PR1.rid < PR2.rid
5        AND   PR1.tid = PR2.tid
6        AND   PR1.len >= CEIL(t * PR2.len)
7        AND   DECODE( SIGN((PR1.len - PR1.pos) - (PR2.len - PR2.pos)),
8                       -1,
9                       (PR1.len - PR1.pos),
10                      (PR2.len - PR2.pos)
11                   ) >=
12             CEIL( (PR1.len + PR2.len) * t / (1+t) );
13       AND   PR1.len - PR1.pos >= CEIL(PR1.len * 2 * t / (1+t))
14       AND   PR2.len - PR2.pos >= CEIL(PR2.len * t)
```

The main difference when compared to the specific prefix table version is

• We use an additional predicates (Lines 14) to calculate and use only the appropriate probing indexing prefixes to produce the candidates.

7.3. An Alternative Implementation Using GROUP BYs

In this scheme, we requires the basic records stored in a normalized table. This eliminates the need of a dedicated prefix table and the verification can be computed using SQL statements without UDFs.

The schema for table R is:

```
CREATE TABLE R (
   rid    INTEGER,
   len    INTEGER,
   tid    INTEGER,
   pos    INTEGER,
   PRIMARY KEY (tid, rid)
);
```

The above schema is identical to the `PrefixR` table in the dedicated prefix version of implementation, except that all the tokens of the records are stored in the R table here.

The SQL statements are

```
CREATE VIEW CANDSET AS
   SELECT   PR1.rid AS rid1, PR2.rid AS rid2, MAX(PR1.pos) AS maxPosX,
            MAX(PR2.pos) AS maxPosY, COUNT(*) AS prefixOverlap
```

A:22

```
FROM       R PR1, R PR2
WHERE      PR1.rid < PR2.rid
   AND     PR1.tid = PR2.tid
   AND     PR1.len >= CEIL(t * PR2.len)
   AND     PR1.len - PR1.pos >= CEIL(PR1.len * 2 * t / (1+t))
   AND     PR2.len - PR2.pos >= CEIL(PR2.len * t)
   AND     DECODE( SIGN((PR1.len - PR1.pos) - (PR2.len - PR2.pos)),
                    -1,
                    (PR1.len - PR1.pos),
                    (PR2.len - PR2.pos)
                  ) >=
           CEIL( (PR1.len + PR2.len) * t / (1+t) )
GROUP BY PR1.rid, PR2.rid;


SELECT    R1.rid, R2.rid
FROM      R R1, R R2, CANDSET C
WHERE     C.rid1 = R1.rid
   AND    C.rid2 = R2.rid
   AND    R1.tid = R2.tid
   AND    R1.pos > maxPosX
   AND    R2.pos > maxPosY
GROUP BY R1.rid, R2.rid
HAVING    COUNT(*) + prefixOverlap >=
          (R1.len + R2.len) * t / (1+t) - C.cnt
```

Compared with the non-GROUP-BY implementation of the similarity join algorithms, the GROUP-BY implementation suffers from the fact that

— Positional filtering can only be applied after the GROUP BY operator.
— The GROUP BY operator potentially has more overhead than the DISTINCT operator.

7.3.1. Using Longer Prefixes. We now consider using longer prefixes in the GROUP BY scheme. However, we use dedicated table to store longer prefixes here, but calculate the overlap within the prefixes using GROUP BY clause. According to Corollary 4.7, we can index extra k tokens in the PrefixR table, and then require a candidate pair having no less than $k + 1$ common tokens.

For example, we implement ppjoin as

```
1    CREATE VIEW CANDSET AS
2      SELECT    PR1.rid AS rid1, PR2.rid As rid2
3      FROM      PrefixR PR1, PrefixR PR2
4      WHERE     PR1.rid < PR2.rid
5         AND    PR1.tid = PR2.tid
6         AND    PR1.len >= CEIL(t * PR2.len)
7         AND    PR1.len - PR1.pos >= CEIL(PR1.len * 2 * t / (1+t)) - k
8         AND    PR2.len - PR2.pos >= CEIL(PR2.len * t) - k
9         AND    DECODE( SIGN((PR1.len - PR1.pos) - (PR2.len - PR2.pos)),
10                        -1,
11                        (PR1.len - PR1.pos),
12                        (PR2.len - PR2.pos)
13                      ) >=
14               CEIL( (PR1.len + PR2.len) * t / (1+t) ) - k;
15      GROUP BY PR1.rid, PR2.rid
16      HAVING    COUNT(*) >= k + 1;


     SELECT    R1.rid, R2.rid
     FROM      R R1, R R2, CANDSET C
     WHERE     C.rid1 = R1.rid
        AND    C.rid2 = R2.rid
        AND    VERIFY(R1.len, R1.toks, R2.len, R2.toks, t) = 1
```

Table II. Distributions of Token Frequencies and Record Sizes

Dataset	n	avg_len	$|\mathcal{U}|$	Comment
DBLP	861,567	14.3	580,026	author, title
ENRON	517,386	142.4	1,180,186	email
DBLP-5GRAM	863,516	100.8	1,581,808	author, title
TREC-8GRAM	347,978	864.2	19,767,130	author, title, abstract
TREC-Shingle	347,978	32.0	4,226,495	author, title, abstract

7.4. Further Discussions

7.4.1. Implementation Details. For the group-by version of implementation, [Chaudhuri et al. 2006] proposed an inline implementation that carries the content for the records selected as candidates when generating pairs. This avoids joining the candidate set with the R relation, but requires an additional attribute in the schema to store the tokens for the record in each tuple. We do not use the inline option because it will incur high space and time overheads when records are not short.

We tried using two inequalities to replace the DECODE function as the positional filtering. Another option is to LEAST function to calculate the less value of two records' lengths. Our experiment shows similar running time for all these three implementation alternatives, and thus we use DECODE function in the experimental evaluation.

Our implementation of the VERIFY function exploits the possibility to terminate earlier if the upper bound of the overlap between two records is below the required threshold.

7.4.2. Extending to Non-Self Join. One way to reduce the similarity join between two relations, R and S, to self-join is as follows:

— Concatenate the two relations together, and create an additional attribute *src*.
— Add the following predicate to the SQL statement

```
PR1.src = 'R' AND PR2.src = 'S'
```

To determine the global ordering of the tokens in both relation, the goal is to minimize the number of candidate size. For each token, we calculate the product of its document frequency in both R and S, and then sort the tokens in each record by increasing order of the product. This is a heuristic to replace the increasing *df* order used for the self-join case. When using prefix tables, we only need to extract the appropriate indexing prefixes for records in R and probing prefixes for records in S. To choose one relation as R, we should minimize the total number of tokens in the indexing prefix, thus minimizing the candidate size. One heuristic is to calculate total number of *non-widow* tokens in the indexing prefixes from both set and choose a smaller one as R. A *widow* token is the one that only appears in one of the relations.

8. EXPERIMENTAL EVALUATION

In this section, we present our experimental results on stand-alone and RDBMS-based implementations, respectively.

8.1. Experiments on the Stand-alone Implementation

We first report the experimental results on a stand-alone implementation.

8.1.1. Experiment Setup. We implemented and used the following algorithms in this set of experiments.

— **All-Pairs** is an efficient prefix filtering-based algorithm capable of scaling up to tens of millions of records [Bayardo et al. 2007].

Table III. k Parameters for LSH (recall $= 95\%$)

Dataset	Jaccard	Cosine	Weighted Cosine
DBLP	4	5	5
ENRON	5	6	5
DBLP-5GRAM	5	5	5
TREC-8GRAM	3	4	4

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

— **LSH** is an algorithm to retrieve approximate answers to the problem [Gionis et al. 1999]. Its basic idea is to hash the records using several hash functions so as to ensure that similar records have much higher probability of collision than dissimilar records. We adopt the LSH algorithm in [Theobald et al. 2008]. We concatenate k min-hash signatures from each record into a single signature, and repeat this for a total l times using independent hash functions. Therefore there are $k \cdot l$ hash functions in all, and two records will be regarded as a candidate pair if one of their l signatures is the same. Larger k benefits the selectivity of a signature, while more real results will be missed. Hence larger l is required to keep the recall when k is increasing, and this will increase the preprocessing time to generate min-hash signatures as we have more signatures for a record. Suppose the probability that two min-hashes from two records collide equals to the Jaccard similarity of the two records, we can compute the l value for a given similarity threshold t, the number of min-hashes for a signature k, and a recall rate r: $l = \lceil \log_{(1-t^k)}(1-r) \rceil$. We set the value of recall as 95%, i.e., LSH reports about 95% of the real results, and choose the parameters k that yield the best runtime performance for these datasets, as given in Table III. Note that we do not choose the random projection-based LSH as was used in [Bayardo et al. 2007]. This is because the records are usually sparse vectors, as shown in Table II. Random projection-based method will generate many "zero" signatures such that the values on the projected dimensions of a record are all zero, and hence reduce the selectivity of signatures.

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 [Bayardo et al. 2007]. The index reduction technique proposed in Section 4.3 is applied to All-Pairs, ppjoin, and ppjoin+. All-Pairs has been shown to consistently outperform alternative algorithms such as ProbeCount-Sort [Sarawagi and Kirpal 2004], PartEnum [Arasu et al. 2006], and therefore we didn't consider them here.

All experiments were carried out on a PC with Intel Xeon X3220 @ 2.40GHz CPU and 4GB RAM. The operating system is Debian 4.1.1-21. All algorithms were implemented in C++ and 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. The running time does not include loading, tokenizing, or signature generating time of datasets, but includes the time for computing prefixes and building indexes. We report preprocessing time of the algorithms in Section 8.1.7.

Our experiments covered the following similarity measures: **Jaccard** similarity, **cosine** similarity, and **weighted cosine** similarity. Tokens are weighted using *idf* weight, i.e., $weight(w) = \log \frac{|R|}{|\{r:w \in r\}|}$.

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

— **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 [Arasu et al. 2006; Bayardo et al. 2007; Xiao et al. 2008].

— **ENRON**: This dataset is from the Enron email collection[4]. 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.

— **DBLP-5GRAM**: This is the same DBLP dataset, but further tokenized into 5-grams. Specifically, tokens in a record are concatenated with a single whitespace, and then every 5 consecutive letters is extracted as a 5-gram.[5]

— **TREC-8GRAM**: This dataset is from TREC-9 Filtering Track Collections.[6] It contains 0.35M references from the MEDLINE database. We extracted author, title, and abstract fields to from records. Every 8 consecutive letters is considered as an 8-gram.

— **TREC-Shingle**: We applied Broder's shingling method [Broder 1997] on TREC-8GRAM to generate 32 shingles of 4 bytes per record, using min-wise independent permutations. TREC-8GRAM and TREC-Shingle are dedicated to experiment on near duplicate Web page detection (Section 8.1.8).

Exact duplicates in the datasets are removed after tokenizing. The records are sorted into increasing length, and the tokens within each record are sorted into increasing document frequency. Some important statistics about the datasets are listed in Table II.

8.1.2. Jaccard Similarity.

Candidate Size   Figures 3(a), 3(c), and 3(e) show the sizes of candidate pairs generated by the algorithms and the size of the join result on the DBLP, Enron, and DBLP-5GRAM 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 among the three exact algorithms 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 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 `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 *is* join results. At the threshold of 0.8, the ratio of sizes of query result over candidate size by ppjoin algorithm is 15.5%, 2.7%, and 0.02% for Enron, DBLP, and DBLP-5GRAM, 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.

— LSH generates more candidates than All-Pairs and ppjoin under large threshold settings, but fewer candidates than All-Pairs and ppjoin when small thresholds are applied. This

---

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

[5]According to [Xiao et al. 2008], long q-grams yield better performance than short q-grams on joining long English text strings. Hence we use 5-grams on DBLP and 8-grams on TREC instead of 3-grams and 4-grams, as were used in [Xiao et al. 2008].

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

(a) Jaccard, DBLP, Candidate Size

(b) Jaccard, DBLP, Time

(c) Jaccard, Enron, Candidate Size

(d) Jaccard, Enron, Time

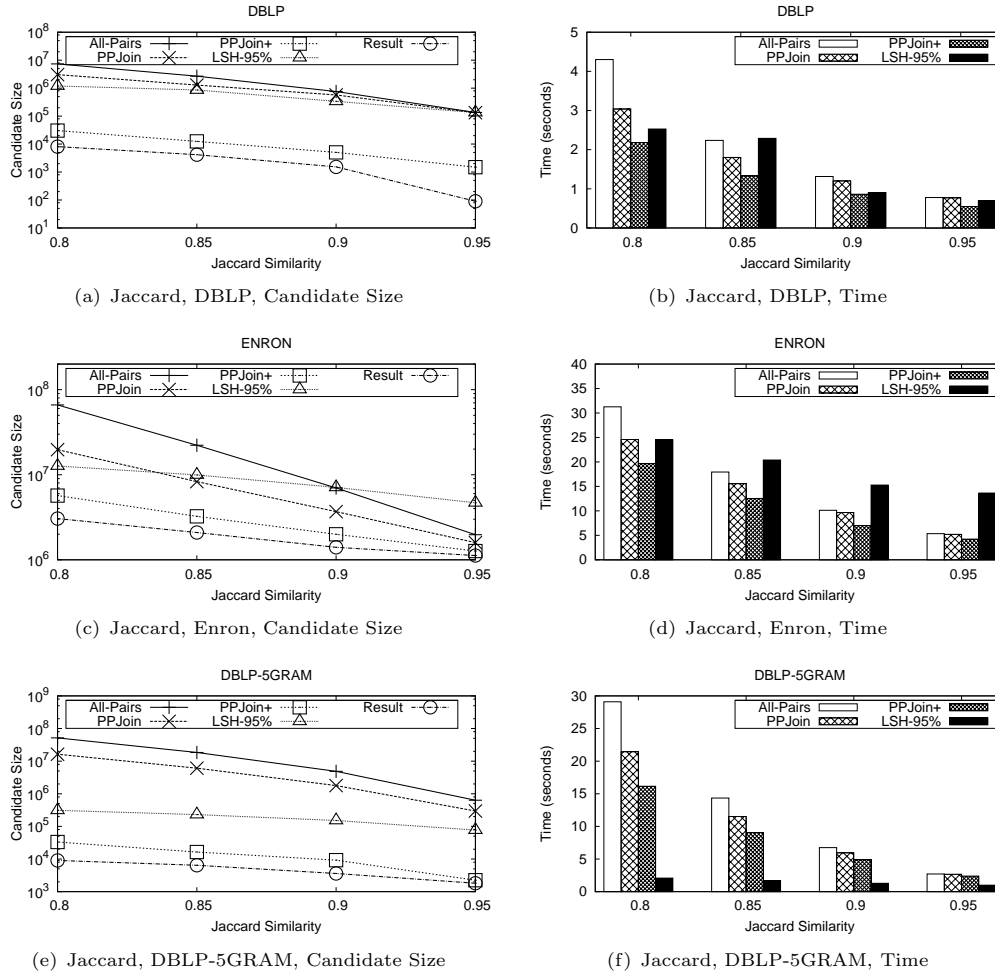(e) Jaccard, DBLP-5GRAM, Candidate Size

(f) Jaccard, DBLP-5GRAM, Time

Fig. 3. Experimental Results - Stand-alone Implementation (Jaccard)

is because when the threshold decreases, the tokens in the prefixes become more frequent, and thus the candidate sizes rapidly increase for algorithms based on prefix filtering. For LSH, smaller thresholds only introduce more signatures while the selectivity remains almost the same, and thus the candidate sizes do not increase as fast as All-Pairs and ppjoin. On DBLP-5GRAM, LSH produces much fewer candidates than ppjoin. The main reason is that the q-grams are less selective then English words and even the rarest q-gram of a record tends to be fairly frequent.

Running Time Figures 3(b), 3(d), and 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 efficient *exact* algorithm[7], 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 (a) index construction,

---

[7]Note that LSH is an approximate algorithm.

(a) Cosine, DBLP, Candidate Size

(b) Cosine, DBLP, Time

(c) Cosine, Enron, Candidate Size

(d) Cosine, Enron, Time

(e) Cosine, DBLP-5GRAM, Candidate Size
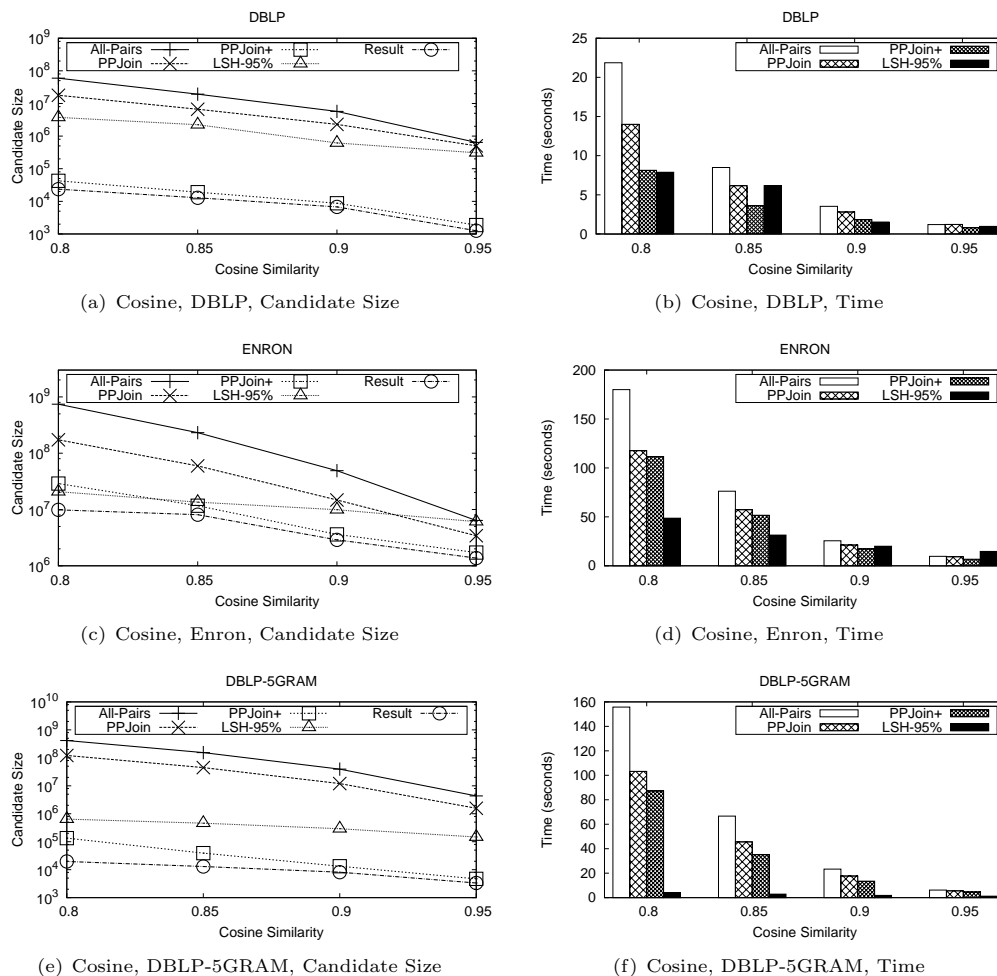
(f) Cosine, DBLP-5GRAM, Time

Fig. 4.   Experimental Results - Stand-alone Implementation (Cosine)

probing, and other overheads are more noticeable with a high similarity threshold, as the result is small and easy to compute. (b) inverted lists in the indexes 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 1.7x speed-up against All-Pairs on Enron, 1.4x on DBLP and DBLP-5GRAM. At the same threshold, ppjoin+ can achieve 1.8x speed-up on DBLP-5GRAM, 2x on DBLP, and 1.6x on Enron.

The performance between ppjoin and ppjoin+ is most substantial on DBLP-5GRAM, 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 16% 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.

Another important observation is that the improvement of positional filtering and suffix filtering in overall running time is not so significant as the improvement in candidate size. There are two main factors: (a) The running time includes constructing inverted index and accessing inverted lists to find candidates. ppjoin and ppjoin+ have no improvement in these two procedures. (b) ppjoin+ prunes candidates with binary searches, and this overhead will increase with the level of recursion, though the candidate size will be reduced.

When compared with the approximate algorithm, ppjoin+ is 1.7x faster than LSH on DBLP, and 3.2x faster on ENRON. LSH performs better on DBLP-5GRAM, and the speed-up can be up to 7.8x, as expected from the candidate sizes. This is again due to the poor selectivity of the q-grams.

8.1.3. Cosine Similarity. We ran all the algorithms on the DBLP, ENRON, and DBLP-5GRAM datasets using the cosine similarity function, and plot the candidate sizes in Figures 4(a), 4(c), and 4(e) and running times in Figures 4(b), 4(d), and 4(f). 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. The speed-ups of the ppjoin and ppjoin+ algorithm can be up to 1.6x and 2.7x on DBLP, respectively; on Enron, the speed-ups are 1.5x and 1.6x, respectively; on DBLP-5GRAM, the speed-ups are 1.5x and 1.8x, respectively. On DBLP and ENRON, ppjoin+ is faster than LSH under high similarity thresholds, but slower when threshold is as low as 0.8. On DBLP-5GRAM, LSH is always the most efficient algorithm, and gap is more significant than that on Jaccard similarity due to a looser constraint. This result demonstrates that LSH is a good choice under low threshold settings if not all the join results need to be reported.

8.1.4. Weighted Cosine Similarity. We run the four algorithms with weighted cosine similarity function on DBLP, ENRON, and DBLP-5RAM datasets. The candidate sizes are shown in Figures 5(a), 5(c), and 5(e); and the running times are shown in Figures 5(b), 5(d), and 5(f). Similar trends can be observed as those with Jaccard and cosine similarity functions. All the algorithms now run faster for the same similarity threshold than unweighted cosine similarity, since the tokens in the prefixes have more weights than those in the suffixes. Another important observation is that the speed-ups over All-Pairs algorithm are not as remarkable as with Jaccard or cosine similarity. This is because the tokens in prefixes are usually rare and assigned more weights; a candidate pair is then more likely to be a real result, hence the problem itself becomes easier. For example, at the threshold of 0.8, the ratio of real result size over candidate size by the All-Pairs algorithm increases by 4 times on all the three datasets when compared with unweighted case. With respect to the running time, ppjoin and ppjoin+ perform better than All-Pairs by a small margin, with a speed-up of 1.2x and 1.3x on the three datasets, respectively. LSH is still slower than ppjoin+ on DBLP, but faster on DBLP-5GRAM. Like unweighted case, LSH is faster than ppjoin+ under low thresholds on ENRON, but slower under high thresholds.

8.1.5. Varying Data Sizes. We performed the similarity join using Jaccard similarity on subsets of the DBLP dataset and measured running times.[8] 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 6(a) and 6(b) (both thresholds are fixed at 0.8).

---

[8]We also measured the candidate sizes (e.g., see Figure 2).

(a) Weighted Cosine, DBLP, Candidate Size

(b) Weighted Cosine, DBLP, Time

(c) Weighted Cosine, Enron, Candidate Size

(d) Weighted Cosine, Enron, Time

(e) Weighted Cosine, DBLP-5GRAM, Candidate Size

(f) Weighted Cosine, DBLP-5GRAM, Time
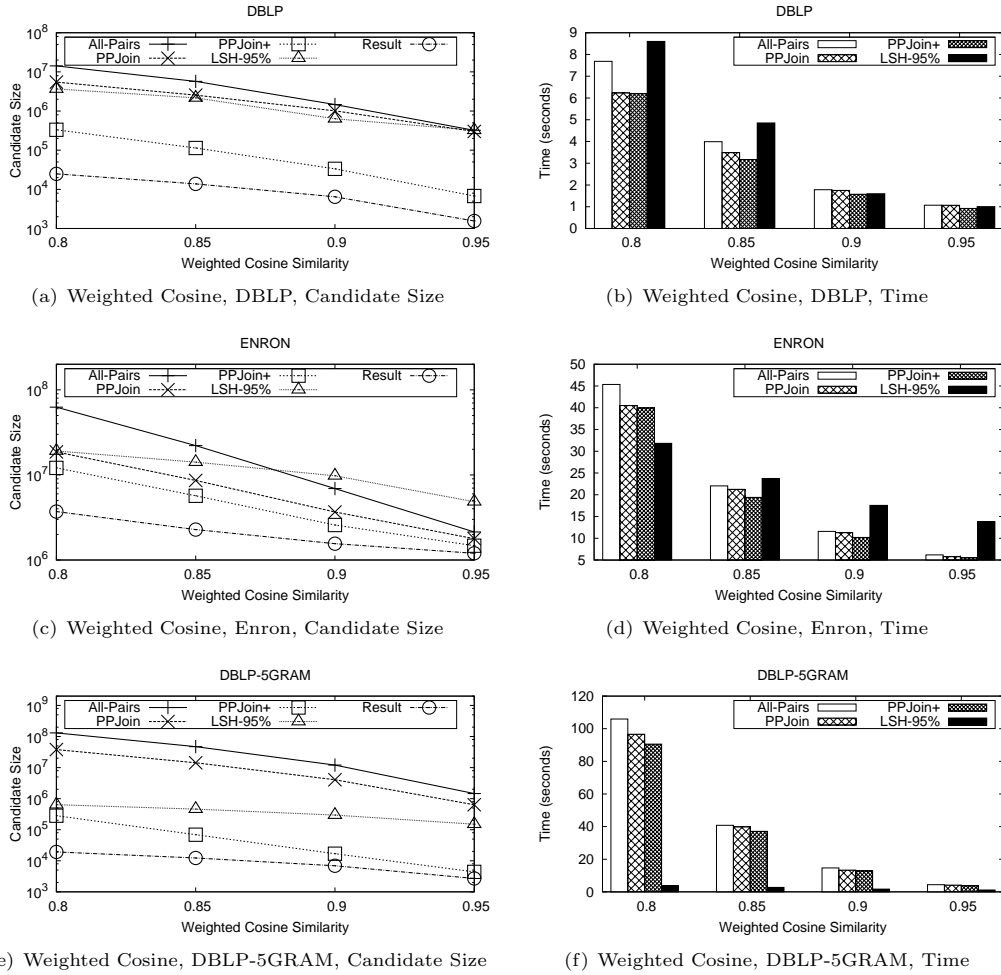
Fig. 5. Experimental Results - Stand-alone Implementation (Weighted Cosine)

(a) Jaccard, DBLP, Time
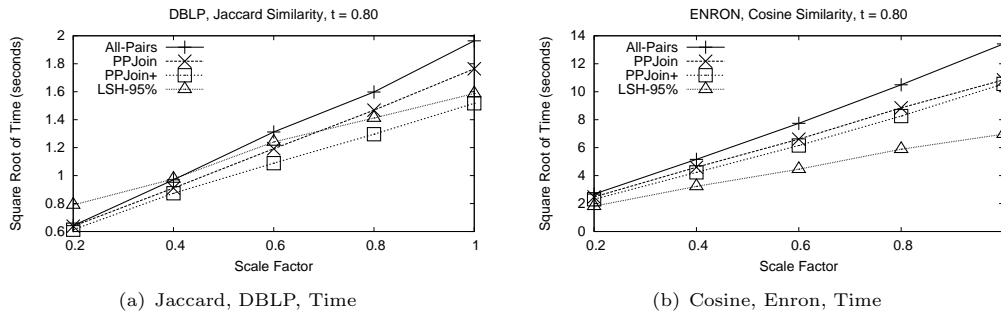
(b) Cosine, Enron, Time

Fig. 6. Experimental Results - Stand-alone Implementation (Scalability)

A:30

It is clear that the running time of all the four algorithms grow quadratically. This is not surprising given the fact that the actual result sizes already grows quadratically (e.g., See Figure 2). Our proposed algorithms have demonstrated a slower growth rate than the All-Pairs algorithm for both similarity functions and datasets.

8.1.6. Impacts of `MAXDEPTH`. An important tuning parameter of the ppjoin+ algorithm is `MAXDEPTH`, the maximum allowable recursion level while performing the suffix filtering. We found the following settings on `MAXDEPTH` work well empirically.

Dataset	$sim()$	MAXDEPTH
ANY	Jaccard	2
ANY	Cosine	3
DBLP-5GRAM	ANY	3 or 4

A possible explanation is that with a looser similarity function (e.g., cosine similarity) and/or longer records (e.g., DBLP-5GRAM), more probings are needed to achieve good candidate pruning effects, hence a larger `MAXDEPTH` works better.

The reason why we do not use a large `MAXDEPTH` value is because the suffix filtering is most effective in the initial several rounds of recursions. For example, we measured the candidate sizes and running times of ppjoin + and list the result for DBLP-5GRAM using Jaccard similarity of 0.80 in Table IV.

Table IV. Impact of `MAXDEPTH` (DBLP-5GRAM, Jaccard, $t = 0.8$, Join Result Size = 9,041)

MAXDEPTH	Candidate Size	Time (secs)
0	16,268,451	21.429
1	4,198,756	18.011
2	557,258	16.699
3	45,871	16.157
4	16,404	16.219
5	11,306	16.307
6	9,675	16.419
7	9,091	16.520

8.1.7. Preprocessing Time. We measured preprocessing cost for the algorithms. For various algorithms, the preprocessing time includes

— All-Pairs, ppjoin, and ppjoin+ extracting tokens and sorting by decreasing idf;
— LSH extracting tokens and generating min-hash signatures.

Note that All-Pairs and ppjoin+ have the same amount of preprocessing time as ppjoin. The preprocessing time for different algorithms is given in Table V.

We observe that the preprocessing cost of ppjoin is lower than LSH on DBLP and ENRON, but higher on the two q-gram datasets. Two factors may affect the preprocessing time of ppjoin: (a) the number of distinct tokens; (b) the average number of tokens in a record. The first factor affects the time to compute the global ordering of tokens, and the second factor affects the time to sort the array of token in each record. The preprocessing time

Table V. Preprocessing Time (Jaccard, $t = 0.8$, in secs)

Dataset	All-Pairs, ppjoin, ppjoin+	LSH
DBLP	1.41	15.02
ENRON	8.87	54.26
DBLP-5GRAM	11.43	68.96
TREC-8GRAM	61.69	72.14

Table VI. Overall Running Time (Jaccard, $t = 0.8$, in secs)

Dataset	All-Pairs	ppjoin	ppjoin+	LSH
DBLP	5.71	4.45	3.59	17.55
ENRON	40.14	33.45	28.56	78.78
DBLP-5GRAM	40.53	32.87	27.58	71.04
TREC-8GRAM	119.32	90.76	88.70	74.37

of LSH is determined by the choosing of parameters k and l. Larger k or l will bring higher preprocessing cost since there will be more min-hash signatures for a record. The preprocessing time of ppjoin is 6.0 to 10.7 times less than that of LSH on DBLP, ENRON, and DBLP-5GRAM, as they have either small token universe or small average record size (Table II). The advantage of ppjoin on TREC-8GRAM is not as substantial due to larger token universe and longer records. Nevertheless, ppjoin is still more efficient than LSH in terms of preprocessing time.

Considering both preprocessing and join, the overall running time is shown in Table VI. It can be observed that ppjoin and ppjoin+ outperform LSH on the first three datasets. Even on TREC-8GRAM, LSH is marginally faster then ppjoin and ppjoin+.

8.1.8. 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 [Broder et al. 1997]. Later work proposed further approximations mainly to gain more efficiency at the cost of result quality.

Instead, we designed and tested four algorithms that perform *exact* similarity join on q-grams or shingles. (a) qp algorithm where we use the ppjoin+ algorithm to join directly on the set of 8-grams of each record. (b) qa algorithm is similar to qp except that All-Pairs algorithm is used as the exact similarity join algorithm. (c) ql algorithm where we use LSH algorithm to join on the set of 8-grams with 95% of recall. (d) sp algorithm where we use the ppjoin+ algorithm to join on the set of shingles. We use Broder's shingling method [Broder 1997] to generate shingles with min-wise independent permutations. Each record has 32 shingles represented in 4-byte integers.

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:

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

where R_x is the set of result returned by algorithm x.

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

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

t	**Precision**	**Recall**	$time_{qa}$	$time_{qp}$	$time_{sp}$	$time_{ql}$
0.95	0.04	0.28	4.15	3.09	0.22	0.57
0.90	0.09	0.25	13.02	8.00	0.32	1.16
0.85	0.20	0.30	28.16	15.95	0.49	1.72
0.80	0.29	0.26	57.63	27.01	0.82	2.23

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 [Broder et al. 1997] 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 8 seconds. On the other hand, qa takes substantially longer time to perform the same join.

— ql is very efficient to find the approximate answers to detect near duplicate Web pages. The speed-up over exact algorithms is more substantial under low similarity threshold settings.

— 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. We also recommend users to choose LSH rather than shingling as an approximate solution for high recall purpose.

8.2. Experiments on the RDBMS-based Implementation

The main goal of this set of experiments is to compare the performance of various implementation alternative of similarity join algorithms on RDBMSs. We implemented the algorithms discussed in Section 7 using a popular RDBMS.

8.2.1. Experiment Setup. The similarity measures used in the experiments are **Jaccard** similarity and **Cosine** similarity.

Table VIII. Statistics of Datasets

Dataset	n	avg_len	$|U|$	Comment
DBLP	861,567	14.3	580,026	author, title
ENRON	517,386	142.4	1,180,186	email
UNIREF-4GRAM	54,819	164.5	302,847	protein sequence

We used the following publicly available real datasets in the experiment.

— **DBLP**: This dataset is a snapshot of the bibliography records from the DBLP Web site, as has been used in the experiment on stand-alone implementation in Section 8.1.

— **ENRON**: This dataset is from the Enron email collection, as has been used in the experiment on stand-alone implementation in Section 8.1.

— **UNIREF-4GRAM**: It is the UniRef90 protein sequence data from the UniProt project.[9] We sampled 58K protein sequences from the original dataset; each sequence is an array of amino acids coded as uppercase letters. Every 4 consecutive letters is then extracted as a 4-gram.

The records are sorted into increasing length, and the tokens within each record are sorted into increasing document frequency. Some important statistics about the datasets are listed in Table VIII.

---

[9]http://beta.uniprot.org/ (downloaded in March, 2008)

8.2.2. Using the Generic Prefix Table. We first study the effect of using the generic prefix table. We set $t_{\min}$ as 0.8 and use it for all similarity joins with threshold $t \geq t_{\min}$.

Table IX shows the running time of the ppjoin algorithm on the DBLP dataset with varying Jaccard similarity thresholds. Results for other datasets or algorithms are similar.

Table IX. Running Time (in secs) on DBLP

t	Prefix Table	Generic Prefix Table
0.95	22.793	23.876
0.90	66.025	68.773
0.85	143.657	146.148
0.80	296.450	302.544

With respect to the running time, we observe that the implementation using the generic prefix table exhibits similar performance as that using the specific prefix table, though the latter has a slight edge over the former under all the threshold settings. This suggests that the additional predicate that selects probing index prefixes from generic prefix table is not expensive to compute when it is compared with the overall algorithm.

Compared with specific prefix tables, generic prefix tables are more practical for many applications. Hence we use generic prefix tables in the rest of the experiment.

8.2.3. Comparison with All-Pairs. We compare three algorithms, All-Pairs, ppjoin and ppjoin+, on the three datasets. We use Jaccard similarity function for DBLP and ENRON dataset, and Cosine similarity function for UNIREF dataset.

Figures 7(a) – 7(c) show the running time of the three algorithms on the three datasets with varying thresholds, where AP denotes All-Pairs, PP ppjoin, and PP+ ppjoin+.

ppjoin algorithm is the most efficient of the three for all the settings. ppjoin+ is the runner-up on DBLP and UNIREF-4GRAM, but the slowest on ENRON. The three algorithms exhibits similar performance when t is high, but the gap becomes significant when t decreases. ppjoin achieves a speed-up of approximately 2x against All-Pairs on all the three datasets. The reason why ppjoin+ performs worse than ppjoin is that suffix filtering is not so efficient under the RDBMS-base implementation, as the suffix filtering test is implemented using a UDF. Another observation is that ppjoin+ is even slower than All-Pairs on ENRON dataset. This is because 38.1% of the candidate pairs are join results on ENRON, whereas the percentage on DBLP and UNIREF is 4.9% and 0.002%. All-Pairs has already removed a majority of false positive candidates, and additional filtering is not helpful.

For All-Pairs algorithm, the verification phase takes most of the running time, and the percentage grows when t is decreasing. This is due to the large number of candidates produced by All-Pairs algorithm, and the lack of any additional filtering technique to deal with the increase of candidate pairs when t decreases. For ppjoin algorithm, most running time is spent on generating candidate pairs when t is high, and the percentage reduces when t decreases. For example, generating candidate pairs takes 70% of the total running time when t is 0.95 on DBLP, while the percentage drops to 40% when t is 0.8. ppjoin+ algorithm spends most running time on generating candidate pairs, even when t is as low as 0.8. This is because (1) the number of candidates generated by ppjoin+ algorithm is close to the number of real results, and thus the verification time is not as significant as for All-Pairs or ppjoin; (2) unlike under the stand-alone implementation, suffix filtering is not performed efficiently under DBMS implementations, and this contributes to the candidate pair generation time.

On the ENRON dataset, candidate pair generating time of ppjoin+ algorithm is not as dominant as on DBLP and UNIREF-4GRAM dataset. This is because a high percentage of candidate pairs are final results and the total number is quite large, and therefore verification phase takes more running time.
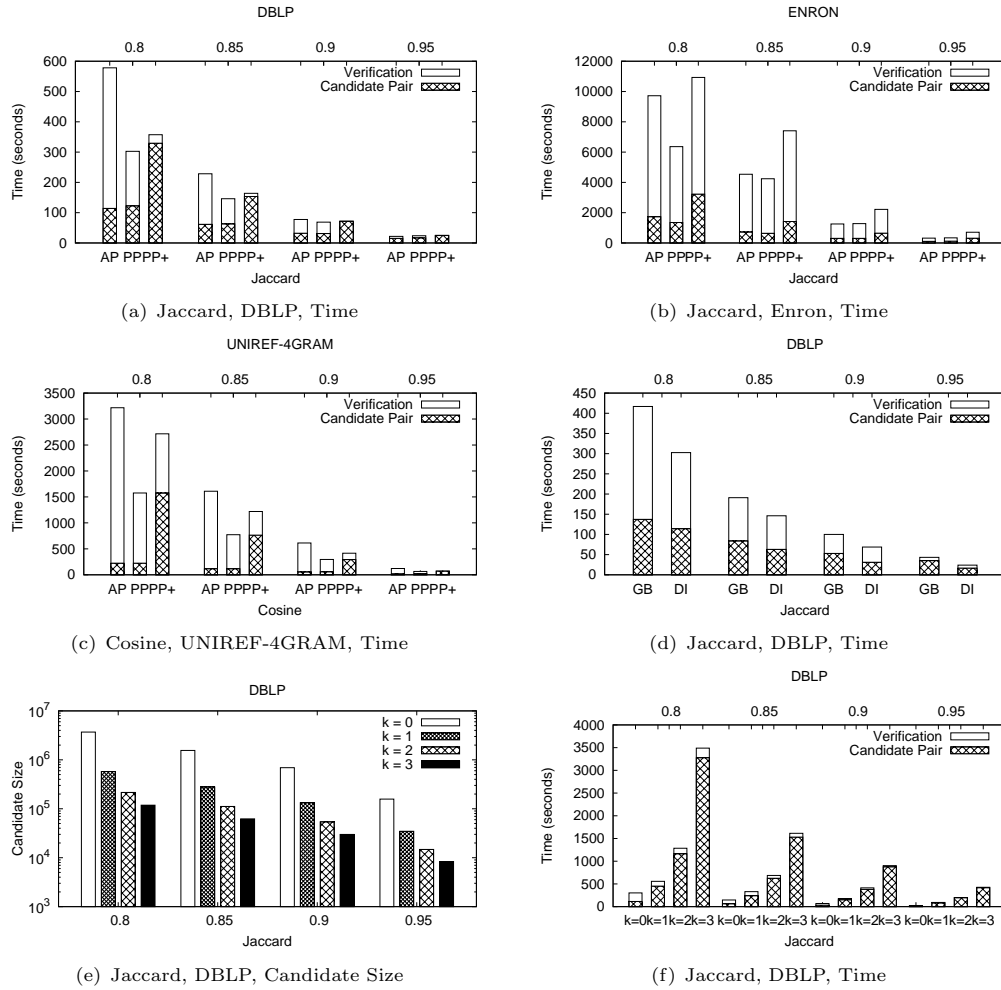
(a) Jaccard, DBLP, Time

(b) Jaccard, Enron, Time

(c) Cosine, UNIREF-4GRAM, Time

(d) Jaccard, DBLP, Time

(e) Jaccard, DBLP, Candidate Size

(f) Jaccard, DBLP, Time

Fig. 7. Experimental Results - DBMS Implementation (I)

8.2.4. Comparison with the Implementation Using GROUP BYs. We compare two versions of
ppjoin algorithm implementations using DISTINCT and GROUP BY operator, respectively.
In addition, we also consider the implementation with longer prefixes. We report the results
on the DBLP dataset with the Jaccard similarity function.

Figure 7(d) shows the running time of the algorithm implemented using distinct and
using group-by, where GB denotes GROUP-BY, and DI denotes DISTINCT.

For the implementation using distinct, we use the generic prefix table, while the normalized table is used for the implementation using group-by. The former is always faster than
the latter, and the time ratio remains a steady value of 1.4 times.

The implementation using group-by employs a normalized table rather than a generic
prefix table, and therefore the table for the join operator to generate candidate pairs has
much more number of rows than a dedicated prefix table. This increases the overhead of the
candidate pair generation, and hence affects the total running time. The implementation
using distinct also benefits from faster verification because the VERIFY function would return
early as soon as the unseen part of the two records is unable to contribute enough overlap.

In contrast, the implementation using group-by has to perform an expensive group by operation, with no early stop technique to facilitate the verification.

To study the effect of longer prefixes, we perform experiment on the DBLP dataset using the Jaccard similarity function and report the candidate size and running time in Figures 7(e) and 7(f).

The candidate size is significantly reduced when we extend the standard prefix length by one more token, but the reduction is less remarkable for further extending. This indicates that most candidate pairs that share only one token within the prefixes are not final results, but those sharing at least two common tokens within the extended prefixes have much more chance to become final results and are not easily to pruned by even longer prefixes.

With respect to the running time, longer prefixes give worse overall performance. Extending prefix by one token is 1.8 times slower than using standard prefix, and the slow-down increases to 4x and 11x when $k = 2$ and 3, respectively. The main reason is that the index size, i.e., the number of rows within the prefix table, increases with longer prefixes. Table X shows the number of rows within the extended prefix table. For longer prefixes, we need to perform the join on a large table in order to generate candidate pairs, and hence more time during the candidate generation phase. Moreover, longer prefixes admit more frequent tokens to the join operation, as the tokens within each record are sorted in the increasing document frequency order. This results in more pairs satisfying the join predicates before calling GROUP BY to filter them. The effect is more obvious when we further increase k, as can be seen from the drastically increase of the running time when we extend the prefix length.

Table X. Using Longer Prefixes

k	Number of Rows
0	2,956,679
1	3,811,945
2	4,667,211
3	5,521,234

8.2.5. Comparison with PartEnum and LSH. We perform experiments to compare three algorithms, ppjoin, PartEnum, and LSH, and report the result on the DBLP dataset using Jaccard similarity. For PartEnum and LSH we use signature tables with respect to specific similarity thresholds, and choose the optimal parameter settings under various t. The number of index entries[10], candidate size and running time of the two algorithms are shown in Figures 8(a) – 8(c), where PartEnum is denoted as PE, and ppjoin as PP.

ppjoin outperforms PartEnum and LSH under all the threshold settings, and the speed-up is increasing with the decrease of similarity threshold. When $t = 0.8$, ppjoin is 29 times faster than PartEnum algorithm, and 2.4 times faster than LSH. There are two main factors contributing to this result. First, ppjoin is smaller than the other two in index size. When t is 0.8, ppjoin's generic prefix table contains 3M rows, whereas LSH's and PartEnum's signature tables have 4.3M and 15.6M rows, respectively. In spite of the fact that ppjoin has an additional attribute of position included in the prefix to perform positional filtering, ppjoin still has the lowest cost in self-joining the prefix table. This explains why ppjoin is more efficient than PartEnum and LSH in candidate generation. Second, the candidate size of PartEnum increases rapidly with the decrease of t, while ppjoin can achieve a moderate increase, hence resulting in faster verification time. For example, the ratio of candidate size between the two algorithms is 11.2 when t is 0.8. LSH also exhibits a moderate increase in

[10]As for the number of index entries for ppjoin algorithm, we mean the number of entries within the probing prefix with respect to specific t.

(a) Jaccard, DBLP, Index Entries

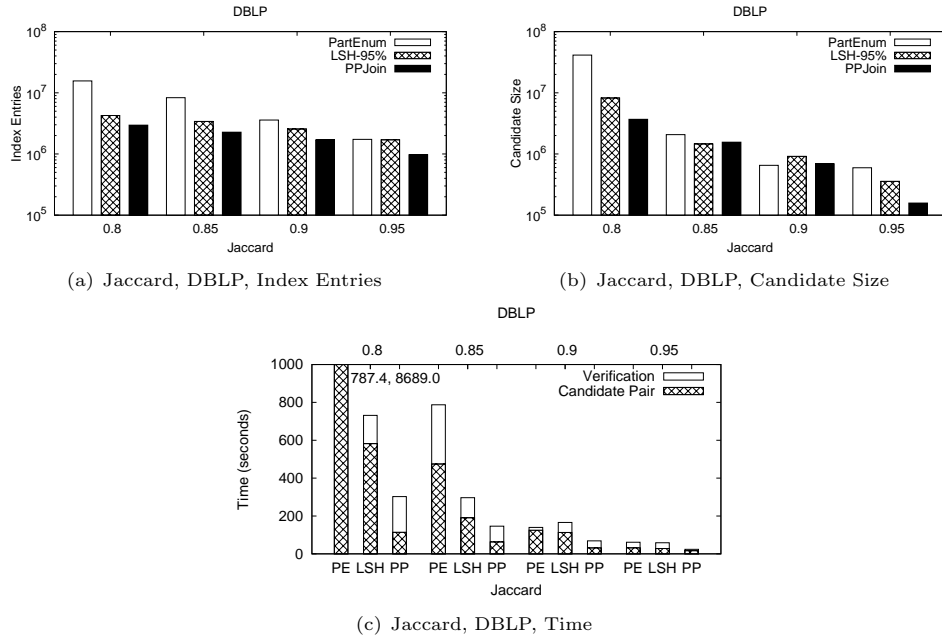(b) Jaccard, DBLP, Candidate Size

(c) Jaccard, DBLP, Time

Fig. 8.   Experimental Results - DBMS Implementation (II)

the candidate size when t decreases, but still produces more candidates than ppjoin, and hence is less efficient.

To sum up, we have the following findings after experimenting on DBMS:

— The implementations using generic prefix table can achieve a comparable performance with the implementations using specific prefix table. It is applicable and more practical to choose generic prefix table for most applications.

— ppjoin+ is inferior to ppjoin over DBMS implementations. Suffix filtering is rendered less efficient by the expensive computation of UDF.

— We suggest user choose DISTINCT operator rather than GROUP BY. In addition, using longer prefix is not promising on DBMS. The prefix table with large size inhibits the efficiency of the algorithm in spite of a smaller candidate size.

9. RELATED WORK

9.1. Near Duplicate Object Detection

Near duplicate object detection has been studied under different names in several areas, including record linkage [Winkler 1999], merge-purge [Hernández and Stolfo 1998], data deduplication [Sarawagi and Bhamidipaty 2002], name matching [Bilenko et al. 2003], just to name a few. [Elmagarmid et al. 2007] is a recent survey on this topic.

Similarity functions are the key to the near duplicate detection task. For text documents, edit distance [Ukkonen 1983] and Jaccard similarity on q-grams [Gravano et al. 2001] 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 [Broder 1997; Chowdhury et al. 2002]. Soundex is a commonly used phonetic similarity measures for names [Russell 1918].

9.2. Exact Similarity Join and 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. [Gravano et al. 2001] 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 [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006; Xiao et al. 2008]. Constraints on overlap, dice and Jaccard similarity measures can be converted to constraints on cosine similarity [Bayardo et al. 2007]. [Arasu et al. 2006] 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 [Bayardo et al. 2007; Chaudhuri et al. 2006; Sarawagi and Kirpal 2004], as discussed above. The second category of work [Arasu et al. 2006] 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. [Arasu et al. 2008] designs a novel framework to identify similar records with some token transformations. In [Lieberman et al. 2008], LSS algorithm is proposed to perform similarity join using Graphics Processing Unit (GPU).

Compared with [Xiao et al. 2008], we have made substantial improvements:

— We introduced an index reduction technique by illustrating the idea of using indexing prefixes.
— We analyzed the prefix filtering technique.
— We generalized ppjoin and ppjoin+ algorithms to several common similarity measures.
— We proposed an optimized global ordering over the token universe to replace the original increasing document frequency ordering.
— We discussed and compared several alternatives to implement the similarity join algorithms over relational database systems.

9.3. Approximate Near Duplicate Object Detection

Several previous work [Broder et al. 1997; Charikar 2002; Chowdhury et al. 2002; Gionis et al. 1999] has concentrated on the problem of retrieving approximate answers to similarity functions. LSH (Locality Sensitive Hashing) [Gionis et al. 1999] is a well-known approximate algorithm for the problem. Its basic idea is to hash the records so that similar records are mapped to the same buckets with high probability. Broder et al. [Broder et al. 1997] 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 [Charikar 2002; Fagin et al. 2003; Gionis et al. 1999; Indyk and Motwani 1998].

9.4. Similarity Join on Strings

The problem of similarity join on strings has been studied by several work [Gravano et al. 2001; Xiao et al. 2008; 2008; Li et al. 2007; Yang et al. 2008].

q-grams are widely used for approximate string match [Gravano et al. 2001]. It is especially useful for edit distance constraints due to its ability to prune candidates with the count filtering on q-grams. Together with prefix-filtering [Chaudhuri et al. 2006], the count filtering can also be implemented efficiently. Filters based on mismatching q-grams are proposed to further speed up the query processing [Xiao et al. 2008].

Gapped q-gram is shown to have better filtering powers than standard q-gram, but is only suitable for edit distance threshold of 1 [Burkhardt and Kärkkäinen 2002]. A variable

length q-gram was proposed in [Li et al. 2007; Yang et al. 2008] and was shown to speed up many computation tasks originally based on q-gram.

Similarity join on strings is also closely related to approximate string matching, an extensively studied topic in algorithm and pattern matching communities. We refer readers to [Navarro 2001] and [Gusfield 1997].

9.5. Top-k Similarity Joins

The problem of top-k query processing has been studied by Fagin *et al* [Fagin 1999; Fagin et al. 2003]. Much work build upon Fagin's work for different application scenarios, e.g., ranking query results from structured databases [Agrawal et al. 2003], processing distributed preference queries [Chang and won Hwang 2002] and keyword queries [Luo et al. 2007].

[Xiao et al. 2009] studies the top-k similarity join problem, which retrieves pairs of objects that have the highest similarity score among the data collection. Several optimizing techniques are proposed by exploiting the monotonicity of similarity function and the order by which data are sorted. The indexing prefix was proposed to reduce both index and candidate sizes.

9.6. Similarity Search

Several existing work studies the similarity search problem [Chaudhuri et al. 2003; Gravano et al. 2001; Li et al. 2008; Hadjieleftheriou et al. 2008; Behm et al. 2009], which returns the records in a collection whose similarity with the query exceeds a given threshold. Based on the inverted list framework, [Li et al. 2008] proposes an efficient principle to skip records when accessing inverted lists. For information retrieval(IR) purpose, [Hadjieleftheriou et al. 2008] designs efficient techniques for indexing and processing similarity queries under IR-style similarity functions. [Behm et al. 2009] proposes a method to omitting some of the frequent tokens while ensuring no true results are missed.

9.7. Document Fingerprinting

Another body of related work is document fingerprinting methods, mostly studied in the area of document retrieval and World Wide Web.

Shingling is a well-known document fingerprinting method [Broder 1997]. Shingles are nothing but fixed length q-grams. All the shingles of a document are generated and only k shingles with the smallest hash values are kept. This process is repeated several times using min-wise independent hash functions. An alternative method is to use every l-th shingle or shingles that satisfy certain properties [Brin et al. 1995].

Manber considered finding similar files in a file system [Manber 1994]. The scheme was improved by Winnowing [Schleimer et al. 2003], which selects the q-gram whose hash value is the minimum within a sliding window of q-grams. The Hailstorm method was proposed in [Hamid et al. 2009] which features the total coverage property, i.e., each token in the document is covered by at least one shingle. Another scheme based on DCT (Discrete Cosine Transformation) was proposed in [Seo and Croft 2008]. [Hamid et al. 2009] performed a comprehensive experimental comparison of some above-mentioned schemes.

Charikar's simhash [Charikar 2002] has been employed to detect near duplicates for Web crawling [Manku et al. 2007]. After converting Web pages to high-dimensional vectors, it maps the vectors to small-sized fingerprints. Near duplicates are identified by collecting the fingerprints that differ by only a few bits.

There are also non-q-gram-based document fingerprinting methods. For example, I-Match [Chowdhury et al. 2002] uses medium-document-frequency tokens as signatures. SpotSigs [Theobald et al. 2008] selects tokens around stopwords as signatures.

10. 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 size of data grows. 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. Experiments are also taken on relational databases to study several alternatives of algorithm implementation. 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

AGRAWAL, S., CHAUDHURI, S., DAS, G., AND GIONIS, A. 2003. Automated ranking of database query results. In *CIDR*.

ARASU, A., CHAUDHURI, S., AND KAUSHIK, R. 2008. Transformation-based framework for record matching. In *ICDE*. 40–49.

ARASU, A., GANTI, V., AND KAUSHIK, R. 2006. Efficient exact set-similarity joins. In *VLDB*.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval* 1st Edition Ed. Addison Wesley.

BAYARDO, R. J., MA, Y., AND SRIKANT, R. 2007. Scaling up all pairs similarity search. In *WWW*.

BEHM, A., JI, S., LI, C., AND LU, J. 2009. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*. 604–615.

BILENKO, M., MOONEY, R. J., COHEN, W. W., RAVIKUMAR, P., AND FIENBERG, S. E. 2003. Adaptive name matching in information integration. *IEEE Intelligent Sys. 18,* 5, 16–23.

BRIN, S., DAVIS, J., AND GARCIA-MOLINA, H. 1995. Copy detection mechanisms for digital documents. In *SIGMOD Conference*. 398–409.

BRODER, A. Z. 1997. On the resemblance and containment of documents. In *SEQS*.

BRODER, A. Z., GLASSMAN, S. C., MANASSE, M. S., AND ZWEIG, G. 1997. Syntactic clustering of the web. *Computer Networks 29,* 8-13, 1157–1166.

BURKHARDT, S. AND KÄRKKÄINEN, J. 2002. One-gapped q-gram filtersfor levenshtein distance. In *CPM*. 225–234.

CHANG, K. C.-C. AND WON HWANG, S. 2002. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*. 346–357.

CHARIKAR, M. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*.

CHAUDHURI, S., GANJAM, K., GANTI, V., AND MOTWANI, R. 2003. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*. 313–324.

CHAUDHURI, S., GANTI, V., AND KAUSHIK, R. 2006. A primitive operator for similarity joins in data cleaning. In *ICDE*.

CHO, J., SHIVAKUMAR, N., AND GARCIA-MOLINA, H. 2000. Finding replicated web collections. In *SIGMOD*.

CHOWDHURY, A., FRIEDER, O., GROSSMAN, D. A., AND MCCABE, M. C. 2002. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst. 20,* 2, 171–191.

CONRAD, J. G., GUO, X. S., AND SCHRIBER, C. P. 2003. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *CIKM*.

ELMAGARMID, A. K., IPEIROTIS, P. G., AND VERYKIOS, V. S. 2007. Duplicate record detection: A survey. *TKDE 19,* 1, 1–16.

FAGIN, R. 1999. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci. 58,* 1, 83–99.

FAGIN, R., KUMAR, R., AND SIVAKUMAR, D. 2003. Efficient similarity search and classification via rank aggregation. In *SIGMOD*.

FAGIN, R., LOTEM, A., AND NAOR, M. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 66,* 4, 614–656.

FETTERLY, D., MANASSE, M., AND NAJORK, M. 2003. On the evolution of clusters of near-duplicate web pages. In *LA-WEB*.

GIBSON, D., KUMAR, R., AND TOMKINS, A. 2005. Discovering large dense subgraphs in massive graphs. In *VLDB*.

GIONIS, A., INDYK, P., AND MOTWANI, R. 1999. Similarity search in high dimensions via hashing. In *VLDB*.

GRAVANO, L., IPEIROTIS, P. G., JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2001. Approximate string joins in a database (almost) for free. In *VLDB*.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press.

HADJIELEFTHERIOU, M., CHANDEL, A., KOUDAS, N., AND SRIVASTAVA, D. 2008. Fast indexes and algorithms for set similarity selection queries. In *ICDE*. 267–276.

HAMID, O. A., BEHZADI, B., CHRISTOPH, S., AND HENZINGER, M. R. 2009. Detecting the origin of text segments efficiently. In *WWW*. 61–70.

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

HERNÁNDEZ, M. A. AND STOLFO, S. J. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery 2,* 1, 9–37.

HOAD, T. C. AND ZOBEL, J. 2003. Methods for identifying versioned and plagiarized documents. *JASIST 54,* 3, 203–215.

INDYK, P. AND MOTWANI, R. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*.

LI, C., LU, J., AND LU, Y. 2008. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*. 257–266.

LI, C., WANG, B., AND YANG, X. 2007. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*.

LIEBERMAN, M. D., SANKARANARAYANAN, J., AND SAMET, H. 2008. A fast similarity join algorithm using graphics processing units. In *ICDE*. 1111–1120.

LUO, Y., LIN, X., WANG, W., AND ZHOU, X. 2007. SPARK: top-k keyword query in relational databases. In *SIGMOD Conference*. 115–126.

MANBER, U. 1994. Finding similar files in a large file system. In *USENIX Winter*. 1–10.

MANKU, G. S., JAIN, A., AND SARMA, A. D. 2007. Detecting near-duplicates for web crawling. In *WWW*. 141–150.

NAVARRO, G. 2001. A guided tour to approximate string matching. *ACM Comput. Surv. 33,* 1, 31–88.

RUSSELL, R. C. 1918. Index, U.S. patent 1,261,167.

SARAWAGI, S. AND BHAMIDIPATY, A. 2002. Interactive deduplication using active learning. In *KDD*.

SARAWAGI, S. AND KIRPAL, A. 2004. Efficient set joins on similarity predicates. In *SIGMOD*.

SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. 2003. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD Conference*. 76–85.

SEO, J. AND CROFT, W. B. 2008. Local text reuse detection. In *SIGIR*. 571–578.

SPERTUS, E., SAHAMI, M., AND BUYUKKOKTEN, O. 2005. Evaluating similarity measures: a large-scale study in the orkut social network. In *KDD*.

THEOBALD, M., SIDDHARTH, J., AND PAEPCKE, A. 2008. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*. 563–570.

UKKONEN, E. 1983. On approximate string matching. In *FCT*.

WINKLER, W. E. 1999. The state of record linkage and current research problems. Tech. rep., U.S. Bureau of the Census.

XIAO, C., WANG, W., AND LIN, X. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB 1,* 1, 933–944.

XIAO, C., WANG, W., LIN, X., AND SHANG, H. 2009. Top-k set similarity joins. In *ICDE*. 916–927.

XIAO, C., WANG, W., LIN, X., AND YU, J. X. 2008. Efficient similarity joins for near duplicate detection. In *WWW*.

YANG, X., WANG, B., AND LI, C. 2008. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*. 353–364.