An Efficient Similarity Join Algorithm with Cosine Similarity Predicate

Dongjoo Lee[1], Jaehui Park[1], Junho Shim[2], and Sang-goo Lee[1]

[1] School of Computer Science & Engineering,
Seoul National University, Seoul 151-742, Korea,
{therocks,jaehui,sglee}@europa.snu.ac.kr
[2] Dept of Computer Science, Sookmyung Women's University,
Seoul 140-742, Korea
jshim@sookmyung.ac.kr

Abstract. Given a large collection of objects, finding all pairs of similar objects, namely *similarity join*, is widely used to solve various problems in many application domains.Computation time of similarity join is critical issue, since similarity join requires computing similarity values for all possible pairs of objects. Several existing algorithms adopt *prefix filtering* to avoid unnecessary similarity computation; however, existing algorithms implementing the prefix filtering have inefficiency in filtering out object pairs, in particular, when aggregate weighted similarity function, such as *cosine similarity*, is used to quantify similarity values between objects. This is mostly caused by large prefixes the algorithms select. In this paper, we propose an alternative method to select small prefixes by exploiting the relationship between arithmetic mean and geometric mean of elements' weights. A new algorithm, *MMJoin*, implementing the proposed methods dramatically reduces the average size of prefixes without much overhead. Finally, it saves much computation time. We demonstrate that our algorithm outperforms a state-of-the-art one with empirical evaluation on large-scale real world datasets.

1 Introduction

Similarity join is an operation that finds all pairs of similar objects from given datasets. It is widely used to solve various problems in many application domains, such as data integration and cleansing [1,2], duplicate Web documents detection [3,4] and information retrieval [5].

More formally, similarity join can be defined as an operation that finds all pairs of objects whose similarity value quantified by the given *similarity function* is above the given *threshold* from the dataset. One issue of similarity join is how to quantify similarity values between objects. Various similarity functions, such as Jaccard-coefficient, cosine similarity, and edit similarity, are used to quantify similarity values between objects. In general, what similarity function to use depends on application domains and there is no best similarity function that works better than any other functions in all application domains. In [6], Chandel et al. grouped similarity functions into five classes, and showed accuracy and performance of similarity functions with various experimental analyses on textual data. From the results, aggregate weighted similarity function, such as cosine similarity showed comparatively good accuracy and performance in detecting errors from textual data. Also it was shown that cosine similarity produce high quality results across several domains [5,7,8,9]. In this paper, we focus on similarity join that uses cosine similarity to quantify similarity values between objects, especially when weights of elements need to be considered.

Another issue of similarity join is the computation time, since similarity join requires computing similarity values for all possible pairs of objects. Many of past researches used approximation techniques to reduce the running time of the operation, while undertaking some loss of expected answers; however, recent trend is to find all pairs of similar objects without any *false drop*. Many of recent works [4,10,11,12] in this trend adopt filtering techniques, such as *prefix filtering* and *positional filtering*, to avoid unnecessary similarity computation; however, positional filtering is not available when we should consider weights of elements. In addition, existing algorithms implementing the prefix filtering have inefficiency in filtering out object pairs, when weights of elements should be considered.

To our best knowledge, previously proposed *All-Pairs* algorithm, one of prefix filtering based methods, showed the best performance among algorithms applicable to our case [4]. From re-implementing the algorithm and analyzing experimental results with several datasets, we found out that prefix size strongly affects the running time of similarity join and All-Pairs has inefficiency in selecting prefixes. Therefore, we focused on how to select small prefixes, and finally, contrived an alternative prefix selection method by exploiting the relationship between arithmetic mean and geometric mean of elements' weights. A new algorithm, *MMJoin*, implementing the proposed prefix selection method reduces average prefix size without further overhead. Reduction of prefix size brings much more reduction of candidates, and finally saves much computation time. We demonstrate that MMJoin outperforms All-Pairs with empirical evaluation on large-scale real-world datasets.

The rest of the paper is organized as follows: Section 2 presents the problem definition with formal notations. Section 3 reviews an existing filtering-based approach. Section 4 describes our prefix selection method and similarity join algorithm. In Sect. 5, we demonstrate experimental results on large-scale real world datasets and give analyses about the results. Related work is covered in Sect. 6 and Sect. 7 concludes the paper.

2 Problem Statement

Given a set of objects $\mathcal{D}$, a similarity function $\mathrm{sim}(x, y)$, and a similarity thresholds t, similarity join is defined as an operation that finds all pairs (x, y) such that $x, y \in \mathcal{D}$ and $\mathrm{sim}(x, y) \geq t$, which is *similarity predicate*. We assume the similarity function is commutative. Thus, if the pair (x, y) satisfies the predicate, so does (y, x), and we need to include only one of them in the result.

According to the similarity function used to quantify similarities between objects, each object needs to be represented in a proper form. For example, *set* for overlap similarity, Jaccard coefficient and Dice coefficient, *vector* for cosine similarity and Tanimoto coefficient, and *sequence* for edit similarity. In this paper, we focus on aggregate weighted similarity function, particularly *cosine similarity.* Therefore, all objects are assumed to be weight vectors on pre-defined dimensions and denoted without right-pointing arrow in the rest of the paper. In addition, by default, $\mathrm{sim}(x, y)$ denotes cosine similarity between vectors x and y, unless otherwise stated. For simplicity, we assume all vectors have unit length. Then, given two vectors $x = \langle x[1], \ldots, x[m] \rangle$ and $y = \langle y[1], \ldots, y[m] \rangle$, cosine similarity is a dot product of two vectors as:

$$\mathrm{sim}(x, y) = \frac{\sum_{i=1}^{m} x[i] \cdot y[i]}{\|x\| \|y\|} = \mathrm{dot}(x, y) = \sum_{i=1}^{m} x[i] \cdot y[i], \tag{1}$$

where $x[i]$ and $y[i]$ are x's and y's weights on ith dimension respectively, and m is the total number of dimensions.

For many problem domains, especially those involving textual data, objects are *sparse* vectors where a vast majority of vector weights are 0. A sparse vector representation for a vector x is the set of all pairs $(i, x[i])$ such that $x[i] > 0$ over all $i = 1, \ldots, m$. Such pairs are called *features* of vector x. If there is a global ordering scheme $\mathcal{O}$ on dimensions $\mathcal{U}$, the sorted list of features is another representation of sparse vector. The *size* of a vector x, denoted by $|x|$, is the number of x's features. Vector size should not be confused with vector *length*, or *magnitude*, which is denoted by $\|x\|$.

For a given vector x, we denote the maximum value $x[i]$ over all i as $\mathrm{maxw}(x)$. For a given dimension i, we denote the maximum value $x[i]$ over all vectors x in the dataset $\mathcal{D}$ as $\mathrm{maxw}_i(\mathcal{D})$. Let us consider an example dataset $\mathcal{X}$ shown in Fig. 1. $\mathcal{X}$ contains five weight vectors on dimensions $\mathcal{U} = \{A, \ldots, O\}$. In Fig. 1, we can see weights of five vectors (blanks mean zero weight), but also additional column and row for $\mathrm{maxw}_i(\mathcal{X})$ and $\mathrm{maxw}(x)$ respectively.

Given a set of sparse vectors $\mathcal{D}$, an *inverted index* for the set consists of m lists $I_1, I_2, \ldots, I_m$ (one for each dimension), where list I_i, we simply refer *inverted list* for dimension i, includes pairs $(x, x[i])$ such that $x \in \mathcal{D}$ and $x[i] > 0$. For example, $I_A = \{(o_1, 0.46), (o_2, 0.46), (o_5, 0.15)\}$.

$\mathcal{U}$	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	maxw(x)
o_1	0.46	0.31			0.31			0.31		0.62	0.31				0.15	0.62
o_2	0.46	0.31		0.31	0.31			0.31		0.62					0.15	0.62
o_3		0.33		0.17		0.33	0.33		0.33	0.50	0.50		0.17			0.50
o_4		0.55	0.18	0.18	0.37	0.18		0.37	0.18		0.37		0.18		0.37	0.55
o_5	0.15		0.30		0.30			0.61				0.46		0.46		0.61
maxw$_i$($\mathcal{X}$)	0.46	0.55	0.30	0.31	0.37	0.33	0.33	0.61	0.33	0.62	0.50	0.46	0.18	0.46	0.37	

Fig. 1. Example dataset $\mathcal{X}$

For a given vector x, let $\dim(x)$ denote a set of all dimensions i such that $i \in \mathcal{U}$ and $x[i] > 0$. For example, $\dim(o_3) = \{B, D, F, G, I, J, K, M\}$. For given two vectors x and y, if $\dim(x) \cap \dim(y) \neq \emptyset$, then there exists at least one dimension i such that $x[i] > 0$ and $y[i] > 0$, which is equivalent to the predicate $\text{dot}(x, y) > 0$. For example, $\dim(o_2) \cap \dim(o_3) = \{B, D, J\} \neq \emptyset$ and $\text{dot}(o_2, o_3) = 0.465 > 0$, and $\dim(o_3) \cap \dim(o_5) = \emptyset$ and $\text{dot}(o_3, o_5) = 0$.

For a given vector x, let the *prefix* of the vector be the first several features of x and denote it as x', and the *suffix* of the vector be the remaining features and denote it as x''. Accordingly, it is obvious that x' and x'' are also vectors, as $x' = \langle x[1], \ldots, x[p], 0, \ldots, 0\rangle$ and $x'' = \langle 0, \ldots, 0, x[p+1], \ldots, x[m]\rangle$, where p is the last dimension on which x''s weight is nonzero. Prefix and suffix satisfy the followings;

- $x' + x'' = x$,
- $|x'| + |x''| = |x|$,
- $\|x'\|^2 + \|x''\|^2 = \|x\|^2$, and
- $\text{dot}(x', y) + \text{dot}(x'', y) = \text{dot}(x, y) = \text{sim}(x, y)$.

3 Filtering-Based Methods

A naïve approach to obtain similarity join result is to enumerate all possible pairs of vectors using *nested loops*, compute similarities of generated pairs, which we call *candidates*, and discard those whose similarity value is below the threshold. This approach generates total $\frac{n(n-1)}{2}$ candidates. Obviously, this approach is not feasible for large datasets due to the huge amount of comparisons.

An alternative approach may improve the performance of the similarity join by using *inverted index* used in IR community. We call this `InvertedIndexJoin` and its pseudo code is shown in Algorithm 1. While scanning each vector, InvertedIndexJoin dynamically constructs inverted index and accumulate similarity values in hash-based map by scanning the inverted index. This brings two benefits: this 1) guarantees that only one pair of (x, y) and (y, x) is considered, since each input vector is compared with vectors that had already been indexed in inverted lists and 2) reduces the overhead of scanning inverted lists, since size of inverted lists remains small in the early stage of the operation; however, still this approach is not feasible for large datasets, because this approach requires huge memory to keep the hash-based map for accumulating similarity values of candidates and yields much overhead to scan all inverted lists for each vector. Several existing algorithms improved the performance of InvertedIndexJoin by exploiting the threshold during matching and indexing.

For an input vector x, InvertedIndexJoin incrementally scans inverted lists from 1 to m such that $x[i] > 0$ (see line 6 - 8 of Algorithm 1.) Suppose that the operation is on dimension p and let x' and x'' denote the corresponding prefix and suffix for x, then $\text{sim}(x, y) = \text{dot}(x', y) + \text{dot}(x'', y)$. [1] If $\text{sim}(x, y) \geq t$ and $\text{dot}(x'', y) < t$, then $\text{dot}(x', y) > 0$, that is x' and y share at least one

[1] $x' = \langle x[1], \ldots, x[p], 0, \ldots, 0\rangle$ and $x'' = \langle 0, \ldots, 0, x[p+1], \ldots, x[m]\rangle$.

Algorithm 1. `InvertedIndexJoin`($\mathcal{D}$, t)

```
Input: a set of vectors $\mathcal{D}$, similarity threshold $t$
Output: $\{(x, y) | x, y \in \mathcal{D} \wedge \mathrm{sim}(x, y) \geq t\}$
 1: $O \leftarrow \emptyset$
 2: $I_1, \ldots, I_m \leftarrow \emptyset$
 3: for each $x \in \mathcal{D}$ do
 4:    $C \leftarrow$ empty map from id to weight
 5:    for $i = 1$ to $m$ such that $x[i] > 0$ do
 6:       for each $(y, y[i]) \in I_i$ do
 7:          $C[y] \leftarrow C[y] + x[i] \cdot y[i]$
 8:       end for
 9:       $I_i \leftarrow I_i \cup \{(x, x[i])\}$
10:    end for
11:    for each $y \in C$ do
12:       if $C[y] \geq t$ then
13:          $O \leftarrow O \cup \{(x, y)\}$
14:       end if
15:    end for
16: end for
17: return $O$
```

dimension. This is similar to the *prefix filtering principle* proposed in [2], which is based on the intuition that if two *canonicalized* objects are similar, some fragments of them should overlap with each other, otherwise the two objects cannot have enough overlap; however, the overlap-based prefix filtering principle does not cover aggregate weighted similarity functions, such as cosine similarity. Therefore, it needs to be extended to cover cosine similarity. Although Bayardo et al.[4] did not note explicitly that they used prefix filtering principle, their approach is on the similar intuition as the prefix filtering principle and we can extend the prefix filtering principle based on the notion used in [4]. An extended version of prefix filtering principle for aggregate weighted similarity functions is formalized in Lemma 1.

Lemma 1. (Aggregate Weighted Prefix Filtering Principle)
Consider two objects x and y, each of which is weight vector on dimension $\mathcal{U}$, which follows an ordering scheme $\mathcal{O}$. If $\mathrm{dot}(x, y) \geq t$*, then any x' and y', such that* $\mathrm{dot}(x'', y) < t$ *and* $\mathrm{dot}(x, y'') < t$*, share at least one dimension.*

3.1 All-Pairs Algorithm

For a vector x, if we can determine the prefix x' such that $\mathrm{dot}(x'', y) < t$ for all y in the dataset, we do not need to condier ys not observed until probing inverted lists I_i such that $i \in \mathrm{dim}(x')$. Also we only need to index x in inverted lists I_i such that $i \in \mathrm{dim}(x')$. Bayardo et al. used $\mathrm{maxw}_i(\mathcal{D})$ to calculate the upper-bound of $\mathrm{dot}(x'', y)$ for all y in the dataset as shown in (2).

$$\mathrm{dot}(x'', y) = \sum_{i=p+1}^{m} x[i] \cdot y[i] \leq \sum_{i=p+1}^{m} x[i] \cdot \mathrm{maxw}_i(\mathcal{D}) \qquad (2)$$

Based on the `InvertedIndexJoin`, they devised an algorithm, `All-Pairs`, not only employing prefix filtering but also exploiting other factors affecting the performance. We rewrite the final version of All-Pairs as shown in Algorithm 2 and 3. Let us see briefly how they improved the performance of similarity join in addition to prefix filtering.

Algorithm 2. `All-Pairs`($\mathcal{D}$, t)

```
Input: a set of vectors $\mathcal{D}$, similarity threshold $t$
Output: $\{(x, y) | x, y \in \mathcal{D} \wedge \mathrm{sim}(x, y) \geq t\}$
 1: Reorder the dimension $1 \ldots m$ such that dimension with the least non-zero entries
    in $\mathcal{D}$ appear first.
 2: Denote the max. of $x[i]$ over all $x \in \mathcal{D}$ as $\mathrm{maxw}_i(\mathcal{D})$.
 3: Denote the max. of $x[i]$ for $1 \ldots m$ as $\mathrm{maxw}(x)$.
 4: $O \leftarrow \emptyset$
 5: $I_1, I_2, \ldots, I_m \leftarrow \emptyset$
 6: for each $x \in \mathcal{D}$ in decreasing order of $\mathrm{maxw}(x)$ do
 7:    $O \leftarrow O \cup$ Find-Matches$(x, I_1, I_2, \ldots, I_m, t)$
 8:    $b \leftarrow 0$
 9:    for $i = m$ to 1 such that $x[i] > 0$ do
10:       $b \leftarrow b + x[i] \cdot \min(\mathrm{maxw}(x), \mathrm{maxw}_i(\mathcal{D}))$
11:       if $b \geq t$ then
12:          $I_i \leftarrow I_i \cup \{(x, x[i])\}$
13:       end if
14:    end for
15: end for
16: return $O$
```

Exploiting Specific Sort Order. Vectors are sequentially accessed in the algorithm. Suppose that vectors are sorted in decreasing order of $\mathrm{maxw}(x)$ and accessed in that order. For a vector x, x is compared with indexed vectors before indexing it. After indexing x, vectors that are not accessed and indexed yet will be compared with x. Such vectors have smaller maximum weight $\mathrm{maxw}(y)$ than x. Therefore, we can tighten the upper-bound of $\mathrm{dot}(x'', y)$ for such ys that are not indexed yet as $\sum_{i=p+1}^{m} x[i] \cdot \min(\mathrm{maxw}(x), \mathrm{maxw}_i(\mathcal{D}))$ (line 11 of Algorithm 2.) We can determine prefix for x based on such upper-bound and the indexed amount for x can be reduced (line 9 - 15 of Algorithm 2.)

Size Filtering in Matching Phase. For two vectors x and y, if we know $|x|$, $|y|$, $\mathrm{maxw}(x)$ and $\mathrm{maxw}(y)$, we can obtain the upper-bound of $\mathrm{dot}(x, y)$ as $\mathrm{dot}(x, y) \leq \min(|x|, |y|) \cdot \mathrm{maxw}(x) \cdot \mathrm{maxw}(y) \leq |y| \cdot \mathrm{maxw}(x)$. From this, we can obtain $|y| \cdot \mathrm{maxw}(x) < t \leftrightarrow |y| < \frac{t}{\mathrm{maxw}(x)} \rightarrow \mathrm{dot}(x, y) < t$. Finally, when we probe inverted lists, we can remove y such that $|y| < \frac{t}{\mathrm{maxw}(x)}$ from the inverted lists, since all remaining vectors to be compared with y have smaller or equal maximum weight over all dimensions than x (line 6 of Algorithm 3.)

Algorithm 3. `Find-Matches`$(x, I_1, I_2, \ldots, I_m, t)$

```
Input: a vector $x$, inverted lists $I_1, I_2, \ldots, I_m$, similarity threshold $t$
Output: $\{(x, y) | (y, y[i]) \in I_i \wedge \mathrm{sim}(x, y) \geq t\}$
 1: $O \leftarrow \emptyset$
 2: $C \leftarrow$ empty map from id to weight
 3: $b \leftarrow \sum_{i=1}^{m} x[i] \cdot \mathrm{maxw}_i(\mathcal{D})$
 4: $minsize \leftarrow \frac{t}{\mathrm{maxw}(x)}$
 5: for $i = 1$ to $m$ such that $x[i] > 0$ do
 6:    Remove $(y, y[i])$ from $I_i$ s.t. $|y| < minsize$
 7:    for each $(y, y[i]) \in I_i$ do
 8:       if $b \geq t$ or $C[y] > 0$ then
 9:          $C[y] \leftarrow C[y] + x[i] \cdot y[i]$
10:       end if
11:    end for
12:    $b \leftarrow b - x[i] \cdot \mathrm{maxw}_i(\mathcal{D})$
13: end for
14: for each $y \in C$ do
15:    if $C[y] + \min(|x|, |y''|) \cdot \mathrm{maxw}(x) \cdot \mathrm{maxw}(y'') \geq t$ then
16:       if $C[y] + \mathrm{dot}(x, y'') \geq t$ then
17:          $O \leftarrow O \cup \{(x, y)\}$
18:       end if
19:    end if
20: end for
21: return $O$
```

Size Filtering in Verification Phase. Let x be input vector, y be one of indexed vectors and y' be indexed part, that is, prefix for y, then $\mathrm{dot}(x, y')$ is obtained in $C[y]$ after probing inverted lists (line 9 of Algorithm 3.) Then we can obtain $\mathrm{sim}(x, y)$ by adding $\mathrm{dot}(x, y'')$ to $C[y]$, where y'' is un-indexed part, that is, suffix for y. Before calculating $\mathrm{dot}(x, y'')$, we may avoid unnecessary computation for $\mathrm{dot}(x, y'')$ by calculating the upper-bound of $\mathrm{dot}(x, y'')$ as $\min(|x|, |y''|) \cdot \mathrm{maxw}(x) \cdot \mathrm{maxw}(y'')$ (line 15 of Algorithm 3.)

4 MMJoin

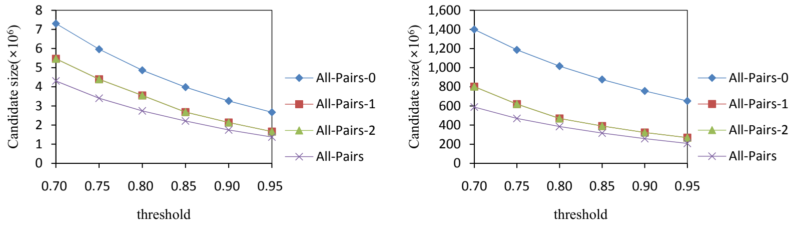
In this section, we describe what mostly affects the performance of All-Pairs and how we improved the performance of All-Pairs.

4.1 Effects of Filtering Techniques on Candidates Size

To see how each technique used in All-Pairs affects the performance of the algorithm, we implemented four versions of All-Pairs:

- **All-Pairs-0** implements basic prefix filtering method.
- **All-Pairs-1** exploits sort order of vectors based on All-Pairs-0.
- **All-Pairs-2** adopts size filtering in matching phase based on All-Pairs-1.
- **All-Pairs** adopts size filtering in verification phase based on All-Pairs-2.

We ran each version of All-Pairs over two datasets with varying the threshold (Details about the datasets are described in Sect. 5) and measured the total number of candidates each algorithm generates to see how much reduction of candidates is made by applying each technique.

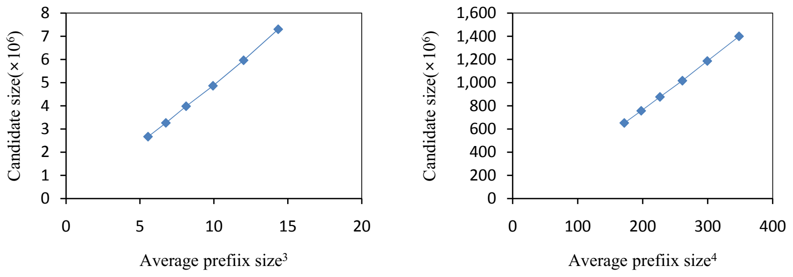


(a) Last.fm **(b)** DBLP

Fig. 2. Effect of each technique on the number of candidates

In general, as shown in Fig. 2, exploiting the specific sort order of vectors brings more performance gain than others. Size filtering in matching phase has almost no effect on reducing candidate size. Although size filtering in verification phase reduces the number of candidates, we cannot except it brings as much reduction of time as candidates, since it needs overhead to compute upper-bound for all vectors remains until verification phases.

Besides, we measured average prefix size of All-Pairs-0 and could make an interesting observation about the relationship between average prefix size and the number of candidates All-Pairs-0 generates. In general, cube (or forth power) of average prefix size is almost proportional to the total number of candidates as shown in Fig. 3. This means that if we reduce average prefix size even *a little*, we will obtain *much* performance improvements. Therefore, we focused on reducing the size of prefixes.



(a) Last.fm **(b)** DBLP

Fig. 3. Effects of average prefix size on total number of candidates

4.2 Tightening Similarity Upper-Bound

If we do not consider weights of elements for quantifying similarity values between objects, prefix size is determined by only the given threshold based on the *overlap-based prefix filtering principle* [2]; however, when we consider weights of elements, prefix size varies according to how we calculate the upper-bound of $\mathrm{dot}(x'', y)$ for all y following the Lemma 1.

In All-Pairs, for a vector x, $\mathrm{maxw}(x)$ and $\mathrm{maxw}_i(\mathcal{D})$ is used to calculate the upper-bound of $\mathrm{dot}(x'', y)$ for all y to be compared with x. Let $M(x)$ be a vector whose ith weight is $\min(\mathrm{maxw}(x), \mathrm{maxw}_i(\mathcal{D}))$. Accordingly, $\mathrm{dot}(x'', M(x)) = \mathrm{dot}(x, M(x)) - \mathrm{dot}(x', M(x))$, and it can be thought that prefix for a vector is determined by adding feature one by one until $\mathrm{dot}(x, M(x)) - \mathrm{dot}(x', M(x)) < t$. From this, we can suppose that if $\mathrm{dot}(x, M(x))$ is big, many features need to be included in prefix to fufill the predicate. As a result, prefix becomes large. This situation is easy to happen when the size of a vector is large. To overcome the weakness of All-Pairs's prefix selection, we contrived an alternative method to calculate the upper-bound of $\mathrm{dot}(x'', y)$ by exploiting the arithmetic mean and geometric mean of elements' weights.

Once again, recall that $\mathrm{sim}(x, y) = \mathrm{dot}(x', y) + \mathrm{dot}(x'', y)$. Let y' and y'' be the prefix and the suffix of y, each of which corresponds to x' and x'' respectively. Then, obviously, $\mathrm{dot}(x'', y) = \mathrm{dot}(x'', y'')$. $\mathrm{dot}(x'', y'')$ can be rewritten as:

$$\mathrm{dot}(x'', y'') = x[p+1] \cdot y[p+1] + \ldots + x[m] \cdot y[m]. \tag{3}$$

By using the relationship between arithmetic mean and geometric mean of elements' weights, we can obtain the upper-bound of $\mathrm{dot}(x'', y'')$ as shown in (4).

$$\begin{aligned} & x[p+1] \cdot y[p+1] + \ldots + x[m] \cdot y[m] \\ \le\ & \frac{x[p+1]^2 + y[p+1]^2}{2} + \ldots + \frac{x[m]^2 + y[m]^2}{2} \\ =\ & \frac{x[p+1]^2 + \ldots + x[m]^2}{2} + \frac{y[p+1]^2 + \ldots + y[m]^2}{2} \\ =\ & \frac{\|x''\|^2 + \|y''\|^2}{2} \end{aligned} \tag{4}$$

With $\|x\|^2 = \|x'\|^2 + \|x''\|^2 = 1$ and $\|x\|^2 \ge 0$ for all vectors, the upper-bound of $\mathrm{dot}(x'', y'')$ can be calculated as (5).

$$\begin{aligned} \mathrm{dot}(x'', y'') &\le \frac{\|x''\|^2 + \|y''\|^2}{2} = 1 - \frac{1}{2}\|x'\|^2 - \frac{1}{2}\|y'\|^2 \\ &\le 1 - \frac{1}{2}\|x'\|^2 \end{aligned} \tag{5}$$

Let $\mathrm{ubdot}(x'')$ denote upper-bound of $\mathrm{dot}(x'', y)$ for all y to be compared with x. Then $\mathrm{ubdot}(x'') = \min(\mathrm{dot}(x, M(x)) - \mathrm{dot}(x', M(x)), 1 - \frac{1}{2}\|x'\|^2)$.

4.3 MMJoin Algorithm

Our algorithm is almost same with All-Pairs and still exploits its merits, because we only changed the way of selecting prefixes as shown in line 8-14 of Algorithm 4. Code for selecting prefix in matching phase of `MMJoin-Find-Matches` is almost same except that $\sum_{i=p+1}^{m} x[i] \cdot \text{maxw}_i(\mathcal{D})$ is used as $M(x)$ instead of $\sum_{i=p+1}^{m} x[i] \cdot \min(\text{maxw}(x), \text{maxw}_i(\mathcal{D}))$. Therefore we omit `MMJoin-Find-Matches` in this paper.

Algorithm 4. `MMJoin`($\mathcal{D}$, t)

```
Input: $\mathcal{D} = \{o_1, o_2, \ldots, o_n\}$, similarity threshold $t$
Output: $\{(x, y) | x, y \in \mathcal{D} \wedge \text{sim}(x, y) \geq t\}$
 1: Reorder the dimension $1 \ldots m$ such that dimension with the least non-zero entries
    in $\mathcal{D}$ appear first
 2: Denote the max. of $x[i]$ over all $x \in \mathcal{D}$ as $\text{maxw}_i(\mathcal{D})$
 3: Denote the max. of $x[i]$ for $1 \ldots m$ as $\text{maxw}(x)$
 4: $O \leftarrow \emptyset$
 5: $I_1, I_2, \ldots, I_m \leftarrow \emptyset$
 6: for each $x \in \mathcal{D}$ in decreasing order of $\text{maxw}(x)$ do
 7:    $O \leftarrow O \cup$ MMJoin-Find-Matches$(x, I_1, I_2, \ldots, I_m, t)$
 8:    $b_1 \leftarrow \sum_{i=1}^{m} x[i] \cdot \min(\text{maxw}_i(\mathcal{D}), \text{maxw}(x))$
 9:    $b_2 \leftarrow 1$
10:    for $i = 1$ to $m$ s.t. $x[i] > 0$ while $\min(b_1, b_2) \geq t$ do
11:       $b_1 \leftarrow b_1 - x[i] \cdot \min(\text{maxw}(x), \text{maxw}_i(\mathcal{D}))$
12:       $b_2 \leftarrow b_2 - \frac{1}{2} x[i]^2$
13:       $I_i \leftarrow I_i \cup \{(x, x[i])\}$
14:    end for
15: end for
16: return $O$
```

5 Experimental Evaluation

In this section, we compare the performance of MMJoin with All-Pairs. We do not compare with other algorithms, since All-Pairs shows the best performance among previous algorithms applicable to our cases[4,8].

5.1 Experimental Setup

We implemented all algorithms in Java 1.6 and used the standard java libraries to implement several data structures used in algorithms. All experiments were performed on a server with 2.83 GHz Intel Core2 Quad, 8 Gbytes of RAM and two 7200 RPM SATA II-IDE hard drives. The operating system is Windows Server 2003.

We ran two algorithms on five real world datasets to cover a wide spectrum of different characteristics. Some important statistics of datasets are summarized in Table 1.

Table 1. Statistics of Datasets

Dataset	n	avg_len	$\lvert\mathcal{U}\rvert$	avg_DF
DBLP	1,298,016	8.6	381,450	29.3
DBLP 4GRAM		23.9	135,204	224.5
LAST.FM	134,949	4.8	47,295	13.8
LAST.FM 4GRAM		11.2	44,272	34.3
TREC	348,566	77.1	298,302	90.1

DBLP is a snapshot of the bibliography records from the DBLP Web site[2]. It contains almost 1.3M 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 [11,4,12,10,13].

TREC is from TREC-9 Filtering Track Collections[3]. It contains 0.35M references from the MEDLINE database. We extracted author, title, and abstract fields from records. Records are subsequently tokenized as in DBLP.

LAST.FM was gathered from last.fm web site[4]. It contains 0.13M randomly selected music tracks including artists and title. Each track is subsequently tokenized as in DBLP.

We made two additional datasets **DBLP 4GRAM** and **LAST.FM 4GRAM**, which are tokenized into 4-grams from DBLP and LAST.FM respectively. In particular, we extracted each 4-gram from tokens that had already been extracted with spaces and punctuations. After extracting tokens, we assigned weights on tokens based on tf-idf weighting scheme[14].

5.2 Experimental Results and Analysis

We ran All-Pairs and MMJoin over five datasets with varying thresholds from 0.70 to 0.95 by 0.05. We ran in-memory algorithm over LAST.FM, LAST.FM 4GRAM, and DBLP datasets; however, we could not use in-memory algorithm over DBLP 4GRAM and TREC in spite of we ran algorithms with excessive memory. Therefore, we ran disk-resident algorithm over DBLP 4GRAM and TREC. Disk resident version of All-Pairs and MMJoin were implemented in the same manner proposed in [4].

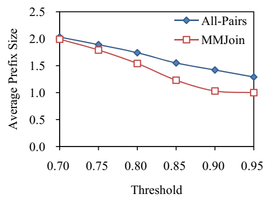
As discussed in Sect. 4.1, average prefix size affects the most on the candidate size. Also most time for similarity join is spent for calculating similarity values of candidates. Therefore, we focused on seeing these sequential effects by measuring

[2] Available at http://dblp.uni-trier.de/xml/
[3] Available at http://trec.nist.gov/data/t9_filtering.html
[4] http://www.last.fm/

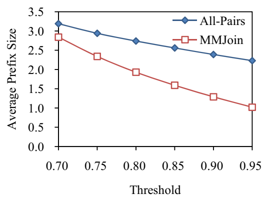
average prefix size, total number of candidates, and total running time (see Fig. 4.) We can observe expected results in all datasets and parameters. Simple analyses about the experimental results is presented in following sub-sections.

Prefix Size. It is observed that the average prefix size increases when the threshold decreases(See Fig. 4a to 4c.) The average prefix size of MMJoin grows faster than that of All-Pairs when threshold decreases; however, the starting point of the MMJoin's average prefix size is much smaller than All-Pairs, especially when the average vector size is larer. In addition, MMJoin never generate bigger prefixes than All-Pairs as proved in Sect. 4.2.

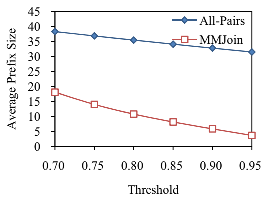
Candidate Size and Time. Figure 4d to 4i shows the number of candidates generated by the algorithms and the time to complete the similarity join with varying the thresholds. We can make identical observations that had been made in previous work; the size of the join result grows modestly when the similarity threshold decreases, and all algorithms generate more candidate pairs with the decrease of the similarity threshold. Besides, time to complete the similarity



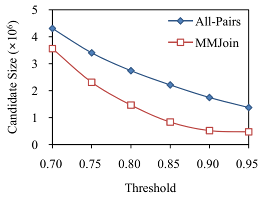
(a) LAST.FM, Prefix size



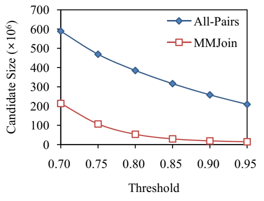
(b) DBLP, Prefix size



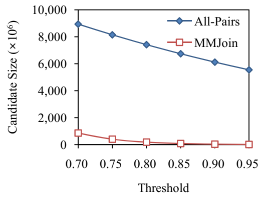
(c) TREC, Prefix size



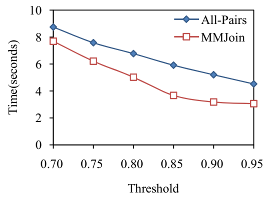
(d) LAST.FM, Candidate size



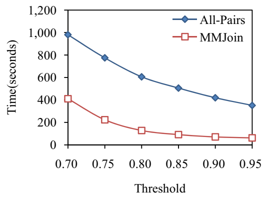
(e) DBLP, Candidate size



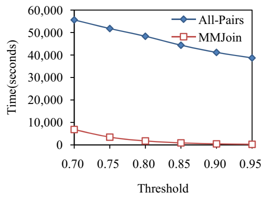
(f) TREC, Candidate size



(g) LAST.FM, Time



(h) DBLP, Time

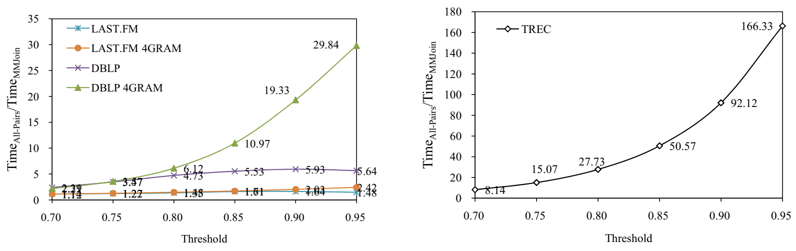


(i) TREC, Time

Fig. 4. Experimental Results

join mostly depends on the candidate size. This means that time to generate candidates do not occupy much portion of total running time in both algorithms. In all situations, MMJoin generates much smaller candidates than All-Pairs.

Performance Differences. MMJoin shows better performance than All-Pairs. It is much strongly observed when the average length of vectors is larger and the threshold is greater as shown in Fig. 5. Even in the case of TREC with threshold 0.95, speed-up of MMJoin is about 166x. This is exactly what we expected in Sect. 4.2.



(a) All datasets except TREC **(b)** TREC

Fig. 5. Performance differences between All-Pairs and MMJoin with varying threholds

6 Related Work

Early studies in similarity join were limited to binary similarity functions for sets including strict containment [15,16,17], equality [16], and non-zero overlap joins [16]. There are found several recent work that covers various partial overlap predicates using a variety of similarity functions including Jaccard coefficient, cosine similarity, edit distance, Hamming distance and their variants [4,12,10,13]. Similarity join in multi-dimensional non-binary space has also been studied[18,1]. [6] compared a large number of similarity functions experimentally with an evaluation on their performance and accuracy.

There is extensive related work in the IR community on designing efficient methods for indexing and compressing textual data [19] viewed as a set. Recent studies showed that this approach is effective to design efficient algorithms realizing similarity join [11,12,10,13]. Most of them use small part of object to reduce the number of objects that have to be fully compared with input object. Sarawagi et al. proposed a simple prefix filtering based algorithms with fully constructed inverted index[12]. Bayardo et al. improved the prefix filtering by dynamically constructing inverted index as well as considering other factors affecting the performance [4]. Recently proposed positional filtering shows remarkable performance improvements on similarity join with set-based similarity functions including Jaccard coefficient, overlap distance, and edit distance [10,13]; however positional filtering is not applicable to weighted cases, since they directly change positional information to measure similarities. To the best of our knowledge, All-Pairs is the best algorithm that is applicable to weighted similarity measures. We extended and improved the All-Pairs adapting a novel prefix selection method.

Our algorithm solves the similarity join problem with assuring that all similar pairs satisfying the given constraints are detected. Another line of work is to solve the similarity join problem with approximation. Locality Sensitive Hashing (LSH) [20] that is widely used in nearest neighbor search can be adapted to similarity join [11,4]. In [4], shingle-based technique was used to detect near duplicated web pages. Several alternatives are proposed to improve hash-based approaches [6].

7 Conclusion

Similarity join has been used in a wide range of application domains such as data integration and cleaning, pattern recognition, and information retrieval. Various similarity functions are used to define join conditions of similarity join. In this paper, we focused on an efficient algorithm for similarity join with cosine similarity predicate. We analyzed the previous algorithms and found out that the most critical process that affects the performance of similarity join is prefix selection. We contrived a novel prefix selection method that efficiently reduces the amount of indexed prefix size by exploiting the relationship between arithmetic mean and geometric mean of elements' weights. We proposed an algorithm, MMJoin, that implements our prefix selection method. Although we refined small part of previous algorithm, we obtained much performance gain through it. We demonstrated that the proposed algorithm outperforms state-of-the-art algorithm with empirical evaluation on large-scale real-world datasets.

Acknowledgments

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency). (grant number NIPA-2010-C1090-1031-0002).

References

1. Chaudhuri, S., Chen, B.C., Ganti, V., Kaushik, R.: Example-driven design of efficient record matching queries. In: VLDB (2007)
2. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE (2006)
3. Henzinger, M.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: SIGIR (2006)
4. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW (2007)

5. Chien, S., Immorlica, N.: Semantic similarity between search engine queries using temporal correlation. In: WWW (2005)
6. Chandel, A., Hassanzadeh, O., Koudas, N., Sadoghi, M., Srivastava, D.: Benchmarking declarative approximate selection predicates. In: SIGMOD (2007)
7. Chuang, S.L., Chien, L.F.: Taxonomy generation for text segments: A practical web-based approach. ACM Trans. Inf. Syst. 23(4), 363–396 (2005)
8. Sahami, M., Heilman, T.D.: A web-based kernel function for measuring the similarity of short text snippets. In: WWW (2006)
9. Spertus, E., Sahami, M., Buyukkokten, O.: Evaluating similarity measures: a large-scale study in the orkut social network. In: KDD (2005)
10. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW (2008)
11. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB (2006)
12. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD (2004)
13. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In: VLDB (2008)
14. Jones, K.S.: A statistical interpretation of term specificity and its application in retrieval. Taylor Graham Series in Foundations of Information Science, pp. 132–142 (1988)
15. Helmer, S., Moerkotte, G.: Evaluation of main memory join algorithms for joins with set comparison join predicates. In: VLDB (1997)
16. Mamoulis, N.: Efficient processing of joins on set-valued attributes. In: SIGMOD (2003)
17. Ramasamy, K., Patel, J.M., Naughton, J.F., Kaushik, R.: Set containment joins: The good, the bad and the ugly. In: VLDB (2000)
18. Böhm, C., Braunmüller, B., Krebs, F., Kriegel, H.P.: Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. SIGMOD Rec. 30(2), 379–388 (2001)
19. Hersh, W.: Managing gigabytes—compressing and indexing documents and images (second edition). Inf. Retr. 4(1), 79–80 (2001)
20. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB (1999)