Efficient Set Similarity Joins Using Min-prefixes

Leonardo A. Ribeiro and Theo Härder

AG DBIS, Department of Computer Science,
University of Kaiserslautern, Germany
{ribeiro,haerder}@informatik.uni-kl.de

Abstract. Identification of all objects in a dataset whose similarity is not less than a specified threshold is of major importance for management, search, and analysis of data. Set similarity joins are commonly used to implement this operation; they scale to large datasets and are versatile to represent a variety of similarity notions. Most set similarity join methods proposed so far present two main phases at a high level of abstraction: candidate generation producing a set of candidate pairs and verification applying the actual similarity measure to the candidates and returning the correct answer. Previous work has primarily focused on the reduction of candidates, where candidate generation presented the major effort to obtain better pruning results. Here, we propose an opposite approach. We drastically decrease the computational cost of candidate generation by dynamically reducing the number of indexed objects at the expense of increasing the workload of the verification phase. Our experimental findings show that this trade-off is advantageous: we consistently achieve substantial speed-ups as compared to previous algorithms.

1 Introduction

Similarity joins pair objects from a dataset whose similarity is not less than a specified threshold; the notion of similarity is mathematically approximated by a *similarity function* defined on the collection of relevant features representing two objects. This is a core operation for many important application areas including data cleaning [1, 5], text data support in relational databases [6], Web indexing [3], social networks [10], and information extraction [4].

Several issues make the realization of similarity joins challenging. First, the objects to be matched are commonly sparsely represented in very high dimensionstext data is a prominent example. It is well-known that indexing techniques based on data-space partitioning achieve only little improvement over a simple sequential scan at high dimensionality. Moreover, many domains involve very large datasets, therefore scalability is a prime requirement. Finally, the concept of similarity is intrinsically application-dependent. Thus, a general purpose similarity join realization has to support a variety of similarity functions [5].

Recently, set similarity joins gained popularity as a means to tackle the issues mentioned above [9, 5, 1, 13]. The main idea behind this special class of similarity joins is to view operands as sets of features and employ a set similarity function to assess their similarity. An important property, predicates containing set similarity functions can be expressed by set overlap [9, 5]. Several popular measures belong to the general class of set similarity functions, including Jaccard, Hamming, and Cosine. Moreover, even when not representing a similarity function on its own, set overlap constraints can still be used as an effective filter for metric distances such as the string edit distance [6, 11].

Most set similarity joins algorithms are composed of two main phases: *candidate generation*, which produces a set of candidate pairs, and *verification*, which applies the actual similarity measure to the generated candidates and returns the correct answer. Recently, Xiao et. al [13] improved the previous state-of-the-art similarity join algorithm due to Bayardo et al. [2] by pushing the overlap constraint checking into the candidate generation phase. To reduce the number of candidates even more, the authors proposed the suffix filtering technique, where a relatively expensive operation is carried out before qualifying a pair as a candidate. As a result, the number of candidates is substantially reduced, often to the same order of magnitude of the result set size.

In this paper, we propose a new index-based algorithm for set similarity joins. Our work builds upon the previous work of [2] and [13], however, we follow an opposite approach to that of [13]. Our focus is on the decrease of the computational cost of candidate generation instead of number of candidates reduction. For this, we introduce the concept of *min-prefix*, a generalization of the *prefix filtering* concept [5]. Min-prefix allows to *dynamically* maintain the length of the inverted lists reduced to a minimum, and therefore the candidate generation time is drastically decreased. We address the increasing in the workload of the verification phase, a side-effect of our approach, by interrupting the computation of candidate pairs that will not meet the overlap constraint as early as possible. Finally, we improve the overlap score accumulation by avoiding the overhead of dedicated data structures. We experimentally demonstrated that our algorithm consistently outperforms previous ones for unweighted and weighted sets.

2 Preliminaries

Given a finite universe U of features and a database D of sets, where every set consists of a number features from U[1], let $sim\left(x_1, x_2\right)$ be a set similarity function that maps a pair of sets x_1 and x_2 to a number in $[0, 1]$. We assume the similarity function is commutative, i.e., $sim\left(x_1, x_2\right) = sim\left(x_2, x_1\right)$. Given a threshold γ, $0 \leq \gamma \leq 1$, our goal is to identify all pairs $\left(x_1, x_2\right), x_1, x_2 \in D$, which satisfy the similarity predicate $sim\left(x_1, x_2\right) \geq \gamma$. We focus on a general class of set similarity functions, for which the similarity predicate can be equivalently represented as a set overlap constraint of the form $|x_1 \cap x_2| \geq minoverlap\left(x_1, x_2\right)$, where $minoverlap\left(x_1, x_2\right)$ is a function that maps the constant γ and the sizes of x_1 and x_2 to an *overlap lower bound* (overlap bound, for short). Hence, the similarity join problem is reduced to a set overlap problem, where all pairs, whose overlap is not less than $minoverlap\left(x_1, x_2\right)$, are returned.

[1] In Sect. 5, we consider weighted sets where features have associated weights.

Table 1: Set similarity functions

Function	Definition	$minoverlap\left(x_1, x_2\right)$	$\left[minsize\left(x\right), maxsize\left(x_c\right)\right]$
Jaccard	$\frac{\|x_1 \cap x_2\|}{\|x_1 \cup x_2\|}$	$\frac{\gamma}{1+\gamma}\left(\|x_1\| + \|x_2\|\right)$	$\left[\gamma\,\|x\|, \frac{\|x\|}{\gamma}\right]$
Dice	$\frac{2\,\|x_1 \cap x_2\|}{\|x_1\| + \|x_2\|}$	$\frac{\gamma\left(\|x_1\| + \|x_2\|\right)}{2}$	$\left[\frac{\gamma\,\|x\|}{2-\gamma}, \frac{\left(2-\gamma\right)\|x\|}{\gamma}\right]$
Cosine	$\frac{\|x_1 \cap x_2\|}{\sqrt{\|x_1\|\,\|x_2\|}}$	$\gamma\sqrt{\|x_1\|\,\|x_2\|}$	$\left[\gamma^2\,\|x\|, \frac{\|x\|}{\gamma^2}\right]$

This set overlap formulation gives rise to several optimizations. First, it is possible to derive *size bounds*. Intuitively, observe that $|x_1 \cap x_2| \leq |x_1|$ for $|x_2| \geq |x_1|$, i.e., set overlap and, therefore, similarity are trivially bounded by $|x_1|$. By carefully exploiting the similarity function definition, it is possible to derive tighter bounds allowing immediate pruning of candidate pairs whose sizes differ enough. Table 1 shows the overlap constraint and the size bounds of the following widely-used similarity functions [9, 1, 8, 12, 13]: Jaccard, Dice, and Cosine. An important observation is that, for all similarity functions, $minoverlap\left(x_1, x_2\right)$ increases monotonically with one or both set sizes.

Another optimization technique instigated by the set overlap abstraction is the *prefix filtering* concept [5]. The idea is to derive a new overlap constraint to be applied on subsets of the operand sets. More specifically, for any two sets x_1 and x_2 under a same total order O, if $|x_1 \cap x_2| \geq \alpha$, the subsets consisting of the first $|x_1|-\alpha+1$ elements of x_1 and the first $|x_2|-\alpha+1$ elements of x_2 must share at least one element [9, 5]. We refer to such subsets as *prefix filtering subsets*, or simply *prefixes*, when the context is clear; further, let $pref\left(x\right)$ denote the prefix of a set x, i.e., $pref\left(x\right)$ is the subset of x containing the first $|pref\left(x\right)|$ elements according to the ordering O. It is easy to see that, for $\alpha = \lceil minoverlap\left(x_1, x_2\right)\rceil$, the set of all pairs $\left(x_1, x_2\right)$ sharing a common prefix element is a superset of the correct result. Thus, one can identify matching candidates by examining only a fraction of the original sets.

The exact prefix size is determined by $minoverlap\left(x_1, x_2\right)$, which varies according to each matching pair. Given a set x_1, a question is how to determine $|pref\left(x_1\right)|$ such that it suffices to identify all x_2, such that $|x_1 \cap x_2| \geq minoverlap\left(x_1, x_2\right)$. Clearly, we have to take the largest prefix in relation to all x_2. Because the prefix size varies inversely with $minoverlap\left(x_1, x_2\right)$, $|pref\left(x_1\right)|$ is largest when $|x_2|$ is smallest (recall that $minoverlap\left(x_1, x_2\right)$ increases monotonically with $|x_2|$). The smallest possible size of x_2, such that the overlap constraint can be satisfied, is $minsize\left(x_1\right)$. Let $maxpref\left(x\right)$ denote the largest prefix of x; thus, $|maxpref\left(x\right)| = |x| - \lceil minsize\left(x\right)\rceil + 1$.

A specific feature ordering can be exploited to improve performance in two ways. First, we rearrange the sets in D according to a *feature frequency ordering*, O_f, to obtain sets ordered by increasing frequencies. The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs

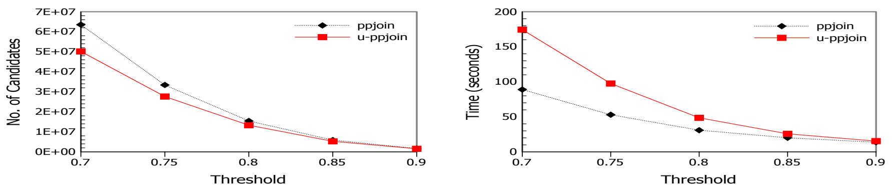
Algorithm 1: The ppjoin algorithm

```
Input: A set collection $D$ sorted in increasing order of the set size; each set is
        sorted according to the total order $O_f$; a threshold $\gamma$
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim(x_p, x_c) \geq \gamma$
1  $I_1, I_2, \ldots, I_{|U|} \leftarrow \varnothing, S \leftarrow \varnothing$
2  foreach $x_p \in D$ do
3      $M \leftarrow$ empty map from set id to $(os, i, j)$      // os = overlap score
4      foreach $f_i \in maxpref(x_p)$ do                // candidate generation phase
5          Remove all $(x_c, j)$ from $I_f$ s.t. $|x_c| < minsize(x_p)$
6          foreach $(x_c, j) \in I_f$ do
7              $M(x_c) \leftarrow (M(x_c).os + 1, i, j)$
8              if $M(x_c).os + min(rem(x_p, i), rem(x_c, j)) < minoverlap(x_p, x_c)$
9                  $M(x_c).os \leftarrow -\infty$      // do not consider $x_c$ anymore
10     $S \leftarrow S \cup Verify(x_p, M, \gamma)$      // verification phase
11     foreach $f_i \in midpref(x_p)$ do
12         $I_f \leftarrow I_f \cup \{(x_p, i)\}$
13 return $S$
```

by shifting lower frequency features to the prefix positions. Second, because O_f imposes an ordering on the elements of a set x, we can use the *positional information* of a common feature between two sets to quickly verify whether or not there are enough remaining features in both sets to meet a given threshold (see [13], Lemma 1). Given a set $x = \{f_1, \ldots, f_{|x|}\}$, let $rem(x, i)$ denote the number of features following the feature f_i in x; thus, $rem(x, i) = |x| - i$.

A further optimization consists of sorting the database D in *increasing* order of the set sizes. By exploiting this ordering, one can ensure that x_1 is only matched against x_2, such that $|x_1| \leq |x_2|$. As a result, the prefix size of x can be reduced: instead of $maxpref(x)$, we obtain a shorter prefix by using $minoverlap(x, x)$ to calculate the prefix size. Let $midpref(x)$ denote the prefix of x for sorted input; therefore $|midpref(x)| = |x| - \lceil minoverlap(x, x) \rceil + 1$.

We are now ready to present a "baseline" algorithm for set similarity joins. Algorithm 1 shows *ppjoin* [13], a state-of-the-art, index-based algorithm that comprises all optimizations previously described. The top-level loop of *ppjoin* scans the dataset D, where, for each set x_p, a *candidate generation phase* delivers a set of candidates by probing the index with the feature elements of $maxpref(x_p)$ (lines 4–9). We call the set x_p, whose features are used to probe the index, a *probing set*; any set x_c that appears in the scanned inverted lists is a *candidate set* of x_p. Besides the accumulated overlap score, the hash-based map M also stores the feature positional information of x_p and x_c (line 7). In the *verification phase*, the probing set and its candidates are checked against the similarity predicate and those pairs satisfying the predicate are added to the result set (line 10); we defer details about the *Verify* procedure to Sect. 4.1. Finally, a *pointer* to set x_p is appended to each inverted list I_f associated with the features of $midpref(x_p)$ (lines 11–12). Note that the algorithm also indexes the



(a) No. of candidates: Jaccard on DBLP (b) Runtime efficiency: Jaccard on DBLP

Fig. 1: Number of candidates vs. runtime efficiency

feature positional information, which is needed for checking the overlap bound (line 8). Additionally, the algorithm employs the lower bound of the set size to dynamically remove sets from inverted lists (line 5).

3 Min-prefix Concept

In this section, we first empirically show that the number of generated candidates can be highly misleading as a measure of runtime efficiency. Motivated by this observation, we introduce the min-prefix concept and propose a new algorithm that focuses on minimizing the computational cost of candidate generation.

3.1 Candidate Reduction vs. Runtime Efficiency

Most set similarity join algorithms operate on shorter set representations in the candidate generation phase (e.g., prefixes and signatures) followed by a potentially more expensive stage where a thorough verification is conducted on each candidate. Accordingly, previous work has primarily focused on candidates reduction where increased effort is dedicated to candidate generation to achieve stronger filtering effectiveness. In this vein, an intuitive approach consists of moving part of the verification into candidate generation. For example, we can generalize the prefix filtering concept to subsets of any size: $(|x| - \alpha + c)$-sized prefixes must share at least c features. This idea has already been used for related similarity operations, but in different algorithmic frameworks [8, 4]. Lets examine this approach applied to *ppjoin*. We can easily swap part of the workload between verification and candidate generation by increasing feature indexing from *midpref* (x) to *maxpref* (x) (Alg. 1, line 11). We call this version *u-ppjoin*, because it exactly corresponds to a variant of *ppjoin* for unordered datasets. Although *u-ppjoin* considers more sets for candidate generation, a larger number of candidate sets are pruned by the overlap constraint (Alg. 1, lines 8–9). Figure 1 shows the results of both algorithms w.r.t. number of candidates and runtime for varying Jaccard thresholds on a 100K sample taken from the DBLP dataset (details about the datasets are given in Sect. 6). As we see in Fig. 1a, *u-ppjoin* indeed reduces the amount of candidates, especially for lower similarity thresholds, thereby reducing the verification workload[2]. However, the run time results showed in Fig. 1b are reversed: *u-ppjoin* is considerably slower than *ppjoin*. Similar results were reported by Bayardo et al. [2] for the unordered version of their *All-pairs* algorithm. We also observed identical trends on several other real world datasets as well as for different growth pattern of feature indexing. These results reveal that, at least for inverted-list-based algorithms, candidate set reduction alone is a poor measure of the overall efficiency. Moreover, they suggest that the trade-off of workload shift between candidate generation and verification can be exploited in an opposite way.

3.2 Min-prefix Concept

A set x_c is indexed by appending a pointer to the inverted lists associated with features $f_j \in \mathit{midpref}\,(x_c)$ which results in an indexed set, denoted by $I\,(x_c)$; accordingly, let $I\,(x_c, f_j)$ denote a feature $f_j \in x_c$ whose associated list has a pointer to x_c. A list holds a reference to x_c until being accessed by a probing set with size $|x_p| > \mathit{maxsize}\,(x_c)$, when this reference is eventually removed by size bound checking (Alg. 1, line 5). We call the interval between the processing of the set x_c following in DB sort order and the last set with size less than or equal to $\mathit{maxsize}\,(x_c)$ the *validity window* of $I\,(x_c)$. Within its validity window, any appearance of $I\,(x_c)$ in lists accessed by a probing set either elects $I\,(x_c)$ as a new candidate, if the first appearance thereof, or accumulates its overlap score.

As previously mentioned, the exact (and minimal) size of $\mathit{pref}\,(x_c)$ is determined by the lower bound of pairwise overlaps between x_c and a reference set x_p. As our key observation, the minimal size of $\mathit{pref}\,(x_c)$ monotonically decreases along the validity window of $I\,(x_c)$ due to dataset pre-sorting. Hence, as the validity window of x_c is processed, an increasing number of the indexed features in $\mathit{midpref}\,(x_c)$ no longer alone suffices to elect x_c as a candidate. More specifically, we introduce the concept of *min-prefix*, formally stated as follows.

Definition 1 (Min-prefix). *Let x_c be a set and let $\mathit{pref}\,(x_c)$ be a prefix of x_c. Let x_p be a reference set. Then $\mathit{pref}\,(x_c)$ is a min-prefix of x_c relative to x_p, denoted as $\mathit{minpref}\,(x_c, x_p)$, iff $1 + \mathit{rem}\,(x_c, j) \geq \mathit{minoverlap}\,(x_p, x_c)$ holds for all $f_j \in \mathit{pref}\,(x_c)$.*

When processing a probing set x_p, the following fact is obvious: if x_c first appears in an inverted list associated with a feature $f_j \notin \mathit{minpref}\,(x_c, x_p)$, then (x_c, x_p) cannot meet the overlap bound. We call a feature $I\,(x_c, f_j)$, which is not an element of $\mathit{minpref}\,(x_c, x_p)$, a *stale feature* relative to x_p.

Example 1. Fig. 2a shows an example with an indexed set $I\,(x_1)$ of size 10 and two probing sets x_2 and x_3 of size 10 and 16, respectively. Given Jaccard as similarity function and a threshold of 0.6, we have $\mathit{midpref}\,(x_1) = 3$,

[2] Actually, the verification workload is even more reduced than suggested by number of candidates. Due to the increased overlap score accumulation in the candidate generation, many more candidates are discarded at the very beginning of *Verify*.

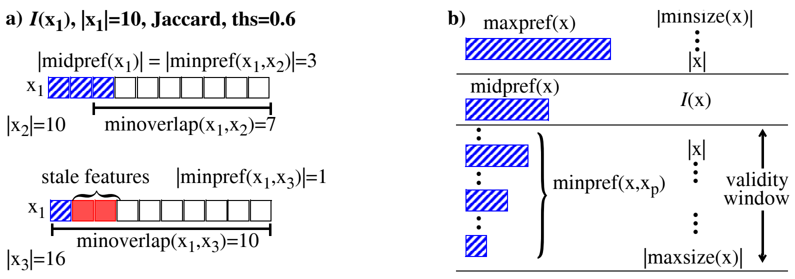


Fig. 2: Min-prefix example

which corresponds to the number of indexed features of $I(x_1)$. For x_2, we have *minpref* $(x_1, x_2) = 3$; thus, no stale features are present. On the other hand, for x_3 as reference set, we have *minpref* $(x_1, x_3) = 1$. Hence, $I(x_1, f_2)$ and $I(x_1, f_3)$ are stale features.

The relationship between the prefix types is shown in Fig. 2b. The three prefixes are minimal in different stages of an index-based set similarity join by exploiting different kinds of information. In the candidate generation phase, the size lower bound of a probing set x defines *maxpref* (x), which is used to find candidates among (shorter) sets indexed. To index x, the database sort order allows reducing the prefix to *midpref* (x). Finally, min-prefix determines the minimum amount of information that needs to *remain* indexed to identify x as a candidate. Differently from the previous prefixes, *minpref* (x, x_p) is defined in terms of a reference set x_p, which corresponds to the current probing set within the validity window of x. The following lemma states important properties of stale features according to the database and the feature ordering.

Lemma 1. *Let D be a database of sets of features; each set is sorted according to a total order O_f. Let $I(x_c)$ be an indexed set and x_p be a probing set. If a feature $I(x_c, f_j)$ is stale in relation to x_p, then $I(x_c, f_j)$ is stale for any $x_{p'}$ such that $|x_{p'}| \geq |x_p|$. Moreover, if $I(x_c, f_j)$ is stale, then any $I(x_c, f_{j'})$, such that $j' > j$, is also stale.*

3.3 The mpjoin Algorithm

Algorithm *ppjoin* only uses stale features for score accumulation. Candidate pairs whose first common element is a stale feature are pruned by the overlap constraint. Because set references are only removed from lists due to size bound checking, repeated processing of stale features are likely to occur very often along the validity window of indexed sets. As strongly suggested by the results reported in Sect. 3.1, such overhead in candidate generation can have a negative impact on the overall runtime efficiency.

Listed in Alg. 2, we now present algorithm *mpjoin* which builds upon the previous algorithms *All-pairs* and *ppjoin*. However, it adopts a novel strategy in

Algorithm 2: The mpjoin algorithm

```
Input: A set collection $D$ sorted in increasing order of the set size; each set is
       sorted according to the total order $O_f$; a threshold $\gamma$
Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim(x_p, x_c) \geq \gamma$
1  $I_1, I_2, \ldots I_{|U|} \leftarrow \varnothing, S \leftarrow \varnothing$
2  foreach $x_p \in D$ do
3      $M \leftarrow$ empty map from set id to $(os, i, j)$      // os = overlap score
4      foreach $f_i \in maxpref(x_p)$ do                 // candidate generation phase
5          Remove all $(x_c, j)$ from $I_f$ s.t. $|x_c| < minsize(x_p)$
6          foreach $(x_c, j) \in I_f$ do
7              if $x_c.prefsize < m$
8                  Remove $(x_c, j)$ from $I_f$      // $I(x_c, j)$ is stale
9                  continue
10             $M(x_c) \leftarrow (M(x_c).os + 1, i, j)$
11             if $M(x_c).os + min(rem(x_p, i), rem(x_c, j)) < minoverlap(x_p, x_c)$
12                 $M(x_c).os \leftarrow -\infty$      // do not consider $x_c$ anymore
13                 if $M(x_c).os + rem(x_c, j) < minoverlap(x_p, x_c)$
14                     Remove $(x_c, j)$ from $I_f$      // $I(x_c, j)$ is stale
15         $x_c.prefsize \leftarrow |x_c| - minoverlap(x_p, x_c) + 1$// update prefix size
16     $S \leftarrow S \cup Verify(x_p, M, \gamma)$      // verification phase
17     $x_p.prefsize \leftarrow |midpref(x)|$// set initial prefix size information
18     foreach $f_i \in midpref(x_p)$ do
19         $I_f \leftarrow I_f \cup \{(x_p, i)\}$
20 return $S$
```

the candidate generation phase. The main idea behind *mpjoin* is to exploit the concept of min-prefixes to *dynamically reduce* the lengths of the inverted lists to a minimum. As a result, a larger number of irrelevant candidate sets are never accessed and processing costs for inverted lists are drastically reduced.

To employ min-prefixes in an index-based similarity join, we need to keep track of the min-prefix size of each indexed set in relation to the current probing set. For this reason, we define min-prefix size information as an attribute of indexed sets, which is named as *prefsize* in the algorithm. At indexing time, *prefsize* is initialized with the value of *midprefix* (line 17). Further, whenever a particular inverted list is scanned during candidate generation, *prefsize* of all related indexed sets is updated using the overlap bound relative to the current probing set (line 15). Stale features can be easily identified by verifying if the *prefsize* attribute is smaller than the feature positional information in a given indexed set. This verification is done for each set as soon as it is encountered in a list; set references in lists associated with stale features are promptly removed and the algorithm moves to the next list element (lines 07–09). Additionally, for a given indexed set, stale features may be probed, before its *prefsize* is updated. Because features of an indexed set are accessed as per the feature order by a probing set (they can be accessed in any order by different probing sets though),

Algorithm 3: The *Verify* algorithm

```
   Input: A probing set $x_p$; a map of candidate sets $M$; a threshold $\gamma$
   Output: A set $S$ containing all pairs $(x_p, x_c)$ such that $sim(x_p, x_c) \geq \gamma$
1  $S \leftarrow \varnothing$
2  foreach $x_c \in M s.t. (overlap \leftarrow M(x_c).os) \neq -\infty$ do
3      if($f_c \leftarrow featureAt(x_c, x_c.prefpos)) < (f_p \leftarrow featureAt(x_p, maxpref(x)))$
4          $f_p \leftarrow featureAt(x_p, M(x_c).i + 1), f_c$++
5      else
6          $f_c \leftarrow featureAt(x_c) M(x_c).j + 1, f_p$++
7      while $f_p \neq end$ and $f_c \neq end$ do    // merge-join-based overlap calc.
8          if  $f_p = f_c$ then $overlap$++, $f_p$++, $f_c$++
9          else
10             if $rem(min(f_p, f_c)) + overlap < minoverlap(x_p, x_c)$ then break
11             $min(f_p, f_c)$++     // advance cursor of lesser feature
12     if $overlap \geq minoverlap(x_p, x_c)$
13         $S \leftarrow S \cup \{(x_p, x_c)\}$
14 return $S$
```

stale feature can only appear as a first common element. In this case, it follows from Definition 1 that the overlap constraint cannot be met and the set reference can be removed from the list (lines 13–14).

The correctness of *mpjoin* partially follows from Lemma 1: it can be trivially shown that the inverted-list reduction strategy of *mpjoin* does not lead to missing any valid result. Another important property of *mpjoin* is that score accumulation is done exclusively on min-prefix elements. This property ensures the correctness of the *Verify* procedure, which is described in the next section.

4 Further Optimizations

4.1 Verification Phase

A side-effect of the index-minimization strategy is the growth of candidate sets. Besides that, as overlap score accumulation is performed only on min-prefixes, larger subsets have to be examined to calculate the complete overlap score. Thus, high performance is a crucial demand for the verification phase. In [13], feature positional information is used to leverage prior overlap accumulation during the candidate generation. We can further optimize the overlap calculation by exploiting the feature order to design a merge-join-based algorithm and the overlap bound to define break conditions.

In Alg. 3, we present the algorithm corresponding to the *Verify* procedure of *mpjoin*, which applies the optimizations mentioned above. (Note that we have switched to a slightly simplified notation.) The algorithm iterates over each candidate set x_c evaluating its overlap with the probing set x_p. First, the starting point for scanning both sets is located (lines 03–06). The approach used here is similar to *ppjoin* (see [13] for more details). Note for both sets, the algorithm starts scanning from the feature following either the last match of candidate generation, i.e., $i+1$ or $j+1$, or the prefixes. No common feature between x_p and x_c is missed, because only min-prefix elements were used for score accumulation during candidate generation. Otherwise, we could miss a match on a stale feature at position j, $x_c.\mathit{prefpos} < j$, whose reference to x_c in the associate inverted list had been previously removed.

The merge-join-based overlap takes place thereafter (lines 7–11). Feature matches increment the overlap accordingly; for each mismatch, the break condition is tested, which consists in verifying if there are enough remaining features in the set relative to the currently tested feature (line 10). Finally, the overlap constraint is checked and the candidate pair is added to the result if there is enough overlap (lines 12–13).

4.2 Optimizing Overlap Score Accumulation

Reference [2] argues that hash-based score accumulators and sequential list processing provide superior performance compared to the heap-based merging approach of other algorithms (e.g., [9]). We now propose a simpler approach by eliminating dedicated data structures and corresponding operations for score accumulation altogether: overlap scores (and the matching positional information) can be stored in the indexed set itself as attributes in the same way as the min-prefix size information. Therefore, overlap score can be directly updated as indexed sets are encountered in inverted lists. We just have to maintain an (re-sizeable) array to store the candidate sets, which will be passed to the *Verify* procedure. Finally, after verifying each candidate, we clear its overlap score and matching positional information.

5 The Weighted Case

We now consider the weighted version of the set similarity join problem. In this version, sets are drawn from a universe of features U_w, where each feature f is associated with a weight $w(f)$. All concepts presented in Sect. 2 can be easily modified to accord with weighted sets. The weighted size of a set x, denoted as $w(x)$, is given by the summation of the weight of its elements, i.e., $w(x) = \sum_{f \in x} w(f)$. Correspondingly, the weighted Jaccard similarity (WJS), for example, is defined as $\mathit{WJS}(x_1, x_2) = w(x_1 \cap x_2)/w(x_1 \cup x_2)$. The prefix definition has to be slightly modified as well. Given an overlap bound α, the weighted prefix of a set x, denoted as $\mathit{pref}(x)$, is the *shortest* subset of x such that $w(\mathit{pref}(x)) > w(x) - \alpha$.

We now present the weighted version of *mpjoin*, called *w-mpjoin*. The most relevant modifications are listed in Alg. 4. As main difference to *mpjoin*, *w-mpjoin* uses the sum of all feature weights up to a given feature instead of feature positional information. For this reason, we define the *cumulative weight* of a feature $f_i \in x$ as $c(f_i) = \sum w(f_j)$, where $1 \leq j \leq i$. We then index $c(f_i)$,

Algorithm 4: The w-mpjoin algorithm

```
   ...
   foreach $f_i \in maxpref(x_p)$ do                // candidate generation phase
5      Remove all $(x_c, c(f_j), j) \in I_f$ from $I_f$ s.t. $w(x_c) < minsize(x_p)$
6      foreach $(x_c, c(f_j), j) \in I_f$ do
7          if $x_c.prefsize + w(f_j) < c(f_j)$
8              Remove $(x_c, c(f_j), j)$ from $I_f$      // $I(x_c, c(f_j), j)$ is stale
9              continue
10         $M(x_c) \leftarrow (M(x_c).os + w(f_j), i, j)$
11         if $M(x_c).os + min(crem(x_p, i), crem(x_c, j)) < minoverlap(x_p, x_c)$
12             $M(x_c).os \leftarrow -\infty$     // do not consider $x_i$ anymore
13             if $M(x_c).os + crem(x_c, j) < minoverlap(x_p, x_c)$
14                 Remove $(x_c, c(f_j), j)$ from $I_f$      // $I(x_c, c(f_j), j)$ is stale
15         $x_c.prefsize \leftarrow w(x_c) - minoverlap(x_p, x_c)$     // update prefix size
16     $S \leftarrow S \cup Verify(x_p, M, \gamma)$      // verification phase
17     $cweight \leftarrow 0$
18     foreach $f_i \in midpref(x)$ do
19         $cweight \leftarrow cweight + w(f_i)$
20         $I_f \leftarrow I_f \cup \{(x_p, cweight, i)\}$
21     $x_p.prefsize \leftarrow cweight$
22 ...
```

for each $f_i \in midpref(x)$ and set the *prefsize* to the cumulative weight of the last feature in $midpref(x)$ (lines 17–21). Note that feature positional information is still necessary to find the starting point of scanning in the *Verify* procedure.

The utility of the cumulative weight in the candidate generation is twofold. First, it is used for overlap bound checking. Given $c(f_i)$, the cumulative weight of the features *following* f_i in x is $crem(x, i) = w(x) - c(f_i)$. Hence, *crem* can be used to verify whether or not there are enough remaining cumulative weights to reach the overlap bound (lines 11 and 13). Second, the cumulative weight is used to identify stale features by comparing it with *prefsize* (line 07). Note that the cumulative weight of the last feature in $minpref(x_c, x_p)$ is always greater than $w(x_c) - \alpha$, for $\alpha = minoverlap(x_p, x_c)$. Hence, to be sure that a given feature is stale, we have to add the weight of the current feature to *prefsize* before comparing it to the cumulative weight.

Due to space constraints, we do not discuss the weighted version of the Verify procedure, but the modifications needed are straightforward.

6 Experiments

6.1 Experimental Setup

The main goal of our experiments is to measure the runtime performance of our algorithms, *mpjoin* and *w-mpjoin*, and compare them against previous, state-of-the-art set similarity join algorithms. All tests were run on an Intel Xeon Quad

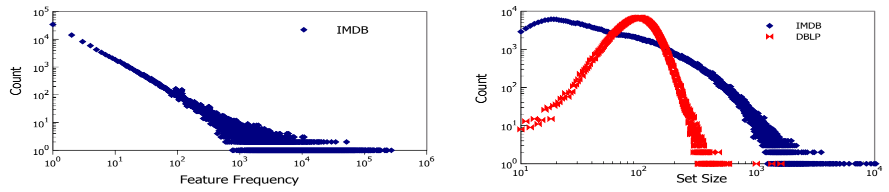
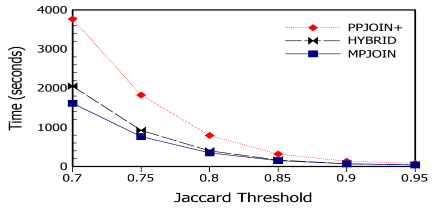


Fig. 3: Feature frequency and set size distributions

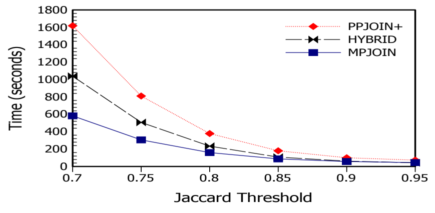
Core 3350 2,66GHz Intel Pentium IV computer (two 3.2 GHz CPUs, about 2.5 GB main memory, Java Sun JDK 1.6.0).

Algorithms: We focused on index-based algorithms, because they consistently outperform competitor signature-based algorithms [2] (see discussion in Sect. 7) and implemented the best known index-based algorithms due to Xiao et. al [13]. For unweighted sets, we used *ppjoin+*, an improved version of *ppjoin*, which applies a suffix filtering technique in the candidate generation phase to substantially reduce the number of candidates. This algorithm constitutes an interesting counterpoint to *mpjoin*. We also investigated a *hybrid* version, which combines *mpjoin* and *ppjoin+* by adding the suffix filtering procedure in *mpjoin* (Alg. 2, inside the loop of line 6 and after line 15). As recommended by the authors, we performed suffix filtering only once for each candidate pair and limited the recursion level to 2. For weighted sets, however, it is not clear how to adapt the suffix filtering technique, because the underlying algorithm largely employs set partitioning based on subset size. In contrast, when working with weighted sets, cumulative weights have to be used, which requires subset scanning to calculate them also for unseen elements. For this reason, this approach is likely to result in poor performance. Therefore, we refrained from using *ppjoin+* and instead employed our adaptation of *ppjoin* for weighted sets, denoted *w-ppjoin*. We only considered the in-memory version of all algorithms. Reference [2] presented a simple nested-loop algorithm fetching entire blocks of disk-resident data, which could be easily adapted for the algorithms evaluated here. Because the same in-memory algorithm is used in each outer-loop iteration and IO overhead is similar in all algorithms, the relative difference in the results reported for the in-memory algorithms should also hold for their external-memory version. For evaluation of weighted sets, we used the well-known IDF weighting scheme. Finally, due to space constraints, we only report results for the Jaccard similarity. The corresponding results for other similarity functions follow identical trends.

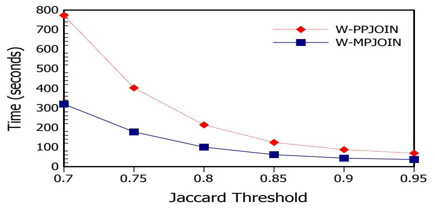
Datasets: We used two well-known real datasets: *DBLP* (dblp.uni-trier.de/xml) containing computer science publications and *IMDB* (www.imdb.com) storing information about movies. We extracted 0.5M strings from each dataset; each string is a concatenation of authors and title, for *DBLP*, and movie title together with actor and actress names, for *IMDB*. We converted all strings to upper case letters and eliminated repeated white spaces. We then generated 4 additional "dirty" copies of each string, i.e., duplicates to which we injected textual modifications consisting of 1–5 character-level modifications (insertions, deletions, and substitutions). Finally, we tokenized all strings into sets of 3-grams, ordered



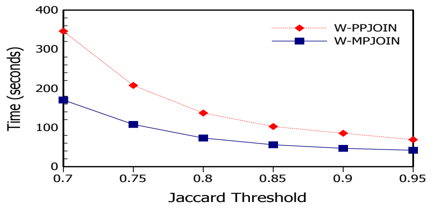
(a) DBLP using unweighted sets



(b) IMDB using unweighted sets



(c) DBLP using weighted sets



(d) IMDB using weighted sets

Fig. 4: Runtime experiments

the tokens as described in Sect. 2, and stored the sets in ascending size order. With this procedure, we simulated typical duplicate elimination scenarios [1, 5]. Figure 3 shows the feature frequency and set size distributions. The feature frequency distribution of both data sets follows a similar power-law distribution and therefore only the distribution of *IMDB* is shown. In contrast, the set size distributions are quite different: *DBLP* has a clear average around 100, whereas *IMDB* does not seem to cluster around any particular value.

6.2 Results

Figure 4a and 4b illustrate the performance results for the unweighted version of the algorithms with varying Jaccard similarity threshold. In all settings, *mpjoin* clearly exhibits the best performance. For the *DBLP* dataset, *mpjoin* achieves more than twofold speed-ups over *ppjoin+* for thresholds lower than 0.85, whereas the performance gains are up to a factor of 3 for the IMDB dataset. Note, the performance advantage of *mpjoin* is more prominent at lower thresholds. In such cases, more stale features are present in the inverted lists, for which a larger number of unqualified candidate sets is generated. Hence, as a result, the performance of *ppjoin+* degrades by a substantially stronger degree.

Note, even the *hybrid* version is slower than *mpjoin*. The candidate reduction does not pay-off the extra-effort of suffix filtering. To highlight this observation, Fig. 5 shows the filtering behavior of the algorithms. In the charts, we show the number of candidates eliminated by suffix filtering (SUFF) and overlap bounds (O_BOUND) during cadidate generation (see Alg. 1, line 8), and the number of candidate pairs considered in the verification phase (CAND). Note, *ppjoin+* eliminates more candidates using O_BOUND than *mpjoin* and *hybrid*. But, a large part of them are candidates related to stale features, i.e., irrelevant candidates that are repeatedly considered along their validity window.

Finally, we observe that all algorithms are about two times faster on the *IMDB* dataset. This is due to the wider set size distribution in the *IMDB* dataset, which results in shorter validity windows for indexed sets. The results for weighted sets are shown in Fig. 4c and 4d. Again, *w-mpjoin* is the most efficient algorithm: it consistently achieves about twofold speed-ups compared to *w-ppjoin*. In general, the results for weighted sets show the same trends as those of the unweighted sets. As expected, the results of all algorithms are considerably faster than those for the unweighted case, because the weighting scheme results in shorter prefixes.

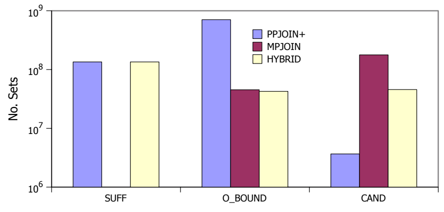


Fig. 5: Filtering behavior, IMDB, 0.8 ths

7 Related Work

A rich variety of techniques have been proposed to improve time efficiency of set similarity joins. Some examples of such techniques are probabilistic dimension reduction methods [3], signature schemes [1, 5], derivation of bounds (e.g., size bound [9, 1, 2, 13, 7]), and exploitation of a specific data set order [9, 2, 13]. Additionally, there are two main query processing models. The first uses an *unnested* representation of sets in which each set element is represented together with the corresponding object identifier. Here, query processing is based on signature schemes and commonly relies on relational database machinery: equi-joins suported by clustered indexes are used to identify all pairs sharing signatures, whereas grouping and aggregation operators together with UDFs are used for verification [5, 1]. In the second model, an index is built for mapping features to the list of objects containing that feature [9, 2, 13]—for self-joins, which can be dynamically performed as the query is processed. The index is then probed for each object to generate the set of candidates which will be later evaluated against the overlap constraint. Previous work has shown that approaches based on indexes consistently outperform signature-based approaches [2] (see also [7] for selection queries). As primary reason, a query processing model based on indexes provides superior optimization opportunities. A major method for that uses an index reduction technique [2, 13], which minimizes the number of features to be indexed. Furthermore, most signature schemes are *binary*, i.e., a single shared signature suffices to elect a pair of sets as candidates. Also, signatures are solely used to find candidates; matching signatures are not leveraged in the verification phase. As a result, each set in a candidate pair must be scanned from the beginning to compute their similarity. In contrast, approaches based on indexes accumulate overlap scores already during candidate generation. Hence, the set elements accessed in this phase can be ignored in the verification.

8 Conclusion

In this paper, we proposed a new index-based algorithm for set similarity joins. Following a completely different approach from previous work, we focused on a reduction of the computational cost for candidate generation as opposed to a lower number of candidates. For this reason, we introduced the concept of *min-prefix*, a generalization of the *prefix filtering* concept, which allows to *dynamically* and *safely* minimize the length of the inverted lists; hence, a larger number of irrelevant candidate pairs are never considered and, in turn, a drastic decrease of the candidate generation time is achieved. As a side-effect of our approach, the workload of the verification phase is increased. Therefore, we optimized this phase by stopping as early as possible the computation of candidate pairs that do not meet the overlap constraint. Finally, we improved the overlap score accumulation by storing scores and auxiliary information within the indexed set itself instead of using a hash-based map. Our experimental results on real datasets confirm that the proposed algorithm consistently outperforms previous ones for both unweighted and weighted sets.

Acknowledgment: Work supported by CAPES/Brazil; grant BEX1129/04-0.

References

1. Arasu, A., Ganti, V., and Kaushik, R.: Efficient Exact Set-Similarity Joins. In: Proc. VLDB, pp. 918–929 (2006)
2. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling Up All Pairs Similarity Search. In: Proc. WWW, pp. 131–140 (2007)
3. Broder, A.Z.: On the Resemblance and Containment of Documents. In: Proc. Compression and Complexity of Sequences, p. 21 (1997)
4. Chakrabarti, K., Chaudhuri, S., Ganti, V., Xin, D.: An Efficient Filter for Approximate Membership Checking. In: Proc. SIGMOD, pp. 805–818 (2008)
5. Chaudhuri, S., Ganjam, K., Kaushik, R.: A Primitive Operator for Similarity Joins in Data Cleaning. In: Proc. ICDE, p. 5 (2006)
6. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., et. al: Approximate String Joins in a Database (Almost) for Free. In: Proc. VLDB, pp. 491–500 (2001)
7. Hadjieleftheriou M., Chandel, A., Koudas, N., Srivastava, D.: Fast Indexes and Algorithms for Set Selection Queries. In: Proc. ICDE, pp. 267–276 (2008)
8. Li, C., Lu, J., Lu, Y.: Efficient Merging and Filtering Algorithms for Approximate String Searches. In: Proc. ICDE, pp. 257–266 (2008)
9. Sarawagi, S., Kirpal, A.: Efficient Set Joins on Similarity Predicates. In: Proc. SIGMOD, pp. 743–754 (2004)
10. Spertus, E., Sahami, M., Buyukkokten, O.: Evaluating Similarity Measures: A Large Scale Study in the Orkut Social Network. In: Proc. KDD, pp. 678–684 (2005)
11. Xiao, C., Wang, W., Lin, X.: Ed-Join: An Efficient Algorithm for Similarity Joins with Edit Distance Constraints. In: PVLDB, vol. 1, no. 1, pp. 933–944 (2008)
12. Xiao, C., Wang, W., Lin, X. Shang, H.: Top-k Set Similarity Joins. In: Proc. ICDE, pp. 916–927 (2009)
13. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient Similarity Joins for Near Duplicate Detection. In: Proc. WWW, pp. 131–140 (2008)