Information Systems 36 (2011) 62–78

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Generalizing prefix filtering to improve set similarity joins ☆

Leonardo Andrade Ribeiro [1,*], Theo Härder

AG DBIS, Department of Computer Science, University of Kaiserslautern, Germany

ARTICLE INFO

Keywords:
Advanced query processing
Set similarity join

ABSTRACT

Identification of all pairs of 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 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 known 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 [2,3], text data support in relational databases [4,5], collaborative filtering [6], Web indexing [7,8], social networks [6], and information extraction [9].

Several issues make the realization of similarity joins challenging. First, the objects to be matched are often sparsely represented in very high dimensions—text data are a prominent example. It is well known that indexing techniques based on data-space partitioning are often outperformed by simple sequential scans at high dimensionality [10]. 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 [3].

Recently, set similarity joins gained popularity as a means to tackle the issues mentioned above [2,3,8,11–13]. 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 the set overlap abstraction [3,11]. Several popular measures belong to the general class of set similarity functions, including Jaccard, Dice, 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 [5,14].

As a concrete example, consider the data cleaning domain. A fundamental data cleaning activity is the identification of the so-called "fuzzy duplicates", i.e., multiple and non-identical representations of a real-world entity. Fuzzy duplicates appear in a dataset often owing to data entry errors like typos and misspellings. In such cases, fuzzy duplicates exhibit slight textual

☆ This paper is a significantly extended and revised version of [1].

* Corresponding author.
E-mail addresses: ribeiro@cs.uni-kl.de, leo.arib@googlemail.com (L.A. Ribeiro), haerder@cs.uni-kl.de (T. Härder).

[1] Work partially supported by CAPES/Brazil; grant BEX1129/04-0.

0306-4379/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2010.07.003

deviations and can be identified by applying a (self) similarity join over the dataset. A widely used notion of string similarity is based on the concept of q-grams. Informally, a q-gram is a substring of size q, obtained by "sliding" a window of size q over the characters of a given string. We can view q-grams as features representing a string. Employing similarity joins based on multidimensional data structures is problematic due to high-dimensionality of the underlying space: up to $|\Sigma|^q$, where Σ is the alphabet from which strings are built (see further discussion in Section 8). In this context, set similarity joins have been the method of choice to realize similarity matching based on q-grams [2,3]. Besides efficiency, the corresponding set similarity functions have been shown to provide competitive quality results compared to other (more complex) similarity functions [15].

Example 1. Let s_1=*Kaiserslautern* and s_2=*Kaisersautern* be strings; their respective set of 2-grams are

$q(s_1) = \{Ka,ai,is,se,er,rs,sl,la,au,ut,te,er,rn\}$,

$q(s_2) = \{Ka,ai,is,se,er,rs,sa,au,ut,te,er,rn\}$.

Consider the Jaccard similarity (JS), which is defined as

$$JS(x_1,x_2) = \frac{|x_1 \cap x_2|}{|x_1 \cup x_2|},$$

where x_1 and x_2 are set operands. Applying Jaccard on the q-gram sets of s_1 and s_2, we obtain: $JS(q(s_1),q(s_2)) = 11/(13+12-11) \cong 0.785$.

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. [12] 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. For that purpose, the overlap constraint is converted into an equivalent Hamming distance and subsets are verified in a coordinated way using a divide-and-conquer algorithm. 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 [12,13], 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 reduction of the number of candidates. For this, we introduce the concept of *min-prefix*, a generalization of the *prefix filtering* concept [3,11] applied to indexed sets. 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. We also improve the overlap score accumulation by avoiding the overhead of dedicated data structures. Furthermore, we consider disk-based and parallel versions of the algorithm. Finally, we conduct a thorough experimental evaluation using synthetic and real datasets. Our results demonstrate that our algorithm consistently outperforms the so-far known ones for unweighted and weighted sets and reveal important trends of set similarity join algorithms in general.

The rest of this paper is organized as follows. Section 2 defines our terminology and reviews important optimization techniques for set similarity joins. In Section 3, we introduce the min-prefix concept and show how it can be exploited to improve the runtime of set similarity joins. In Section 4, we present further optimizations in the candidate generation and verification phase. Section 5 considers disk-based and parallel versions of *mpjoin* and Section 6 describes the version for weighted sets. Experimental results are presented in Section 7. We discuss related work in Section 8, before we wrap up with the conclusions in Section 9.

2. Preliminaries

In this section, we first provide background material on set similarity join concepts and techniques. Then, we describe the baseline algorithm for set similarity joins that we use in this work.

2.1. Background

Given a finite universe U of features and a set collection C, where every set consists of a number features from U,[2] let $Sim(x_1, x_2)$ 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., $Sim(x_1, x_2)$=Sim (x_2, x_1). Given a threshold γ, $0 \le \gamma \le 1$, our goal is to identify all pairs $(x_1,x_2),x_1,x_2 \in C$, which satisfy the similarity predicate $Sim(x_1,x_2) \ge \gamma$.

We focus on a general class of set similarity functions, for which the similarity predicate can be equivalently represented as a set overlap constraint. Specifically, we express the original similarity predicate in terms of an *overlap lower bound* (*overlap bound*, for short) [3].

Definition 1 (*Overlap bound*). Let x_1 and x_2 be sets of features, *Sim* be a set similarity function, and γ be a similarity threshold. The overlap bound between x_1 and x_2 relative to *Sim*, denoted by $minoverlap(x_1, x_2)$, is a function that maps γ and the sizes of x_1 and x_2 to a real value, s.t. $Sim(x_1,x_2) \ge \gamma$ iff $|x_1 \cap x_2| \ge minoverlap(x_1,x_2)$.

Hence, the similarity join problem is reduced to a set overlap problem, where all pairs whose overlap is not less than $minoverlap(x_1, x_2)$ are returned. Table 1 shows the overlap bound of the following widely used similarity functions [2,11,13,16,17]: Jaccard, Dice, and Cosine. An important observation is that, for all similarity functions,

[2] In Section 6, we consider weighted sets where features have associated weights.

Table 1
Set similarity functions.

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

$minoverlap(x_1, x_2)$ increases monotonically with one or both set sizes.

The set overlap formulation enables the derivation of *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.

Definition 2 (*Set size bounds*). Let x_1 be a set of features, *Sim* be a set similarity function, and γ be a similarity threshold. The size bounds of x_1 relative to *Sim* are functions, denoted by $minsize(x_1)$ and $maxsize(x_1)$, that map γ and the size of x_1 to a real value, s.t. $\forall x_2$, if $Sim(x_1,x_2) \geq \gamma$ then $minsize(x_1) \leq |x_2| \leq maxsize(x_1)$.

Therefore, given a set x, we can safely ignore all sets whose size do not fall within the interval $[minsize(x), maxsize(x)]$. Table 1 shows the set size bounds of several similarity functions.

Overlap bound and set size bounds give raise to several other optimizations. We can prune a the comparison space by exploiting the *prefix filtering* concept [3]. 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 [3,11]. We refer to such subsets as *prefix filtering subsets*, or simply *prefixes*, when the context is clear; further, let $pref(x)$ denote the prefix of a set x, i.e., $pref(x)$ is the subset of x containing the first $|pref(x)|$ elements according to the ordering O. It is easy to see that, for $\alpha = \lceil minoverlap(x_1,x_2) \rceil$, the set of all pairs (x_1,x_2) 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 $minoverlap(x_1, x_2)$, which varies according to each matching pair. Given a set x_1, a question is how to determine $|pref(x_1)|$ such that it suffices to identify all matchings of x_1 (no false negatives). Clearly, we have to take the largest prefix in relation to all x_2. Because the prefix size varies inversely with $minoverlap(x_1, x_2)$, $|pref(x_1)|$ is largest when $|x_2|$ is smallest (recall that $minoverlap(x_1, x_2)$ increases monotonically with $|x_2|$). The smallest possible size of x_2, such that the overlap constraint can be satisfied, is $minsize(x_1)$.

Definition 3 (*Max-prefix*). Let x_1 be a set of features. The max-prefix of x_1, denoted by $maxpref(x_1)$, is its smallest prefix needed for identifying $\forall x_2$ s.t. $|x_1 \cap x_2| \geq minoverlap(x_1,x_2)$. The size of max-prefix is given by: $|maxpref(x_1)| = |x_1| - \lceil minsize(x_1) \rceil + 1$.

Another optimization consists of sorting C 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 $|x_2| \geq |x_1|$. 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 [12,13,17].

Definition 4 (*Mid-prefix*). Let x_1 be a set of features. The mid-prefix of x_1, denoted by $midpref(x_1)$, is its smallest prefix needed for identifying $\forall x_2 \geq x_1$ s.t. $|x_1 \cap x_2| \geq minoverlap(x_1,x_2)$. The size of mid-prefix is given by: $|midpref(x_1)| = |x_1| - \lceil minoverlap(x_1,x_1) \rceil + 1$.

Example 2. Recall the sets of q-grams from Example 1, i.e., $q(s_1)$ and $q(s_2)$. Consider JS as similarity function and $\gamma = 0.75$. We have $|q(s_1)| = 13$ and $|q(s_2)| = 12$. Then, we have $\lceil minoverlap(q(s_1),q(s_2)) \rceil = 11$. For $q(s_1)$, we have $[\lfloor minsize(q(s_1)) \rfloor, \lceil maxsize(q(s_1)) \rceil] = [10,18]$. Further, we have $|maxpref(q(s_1))| = 4$ and $|midpref(q(s_1))| = 3$. Assuming, for simplicity, that the features of $q(s_1)$ are already sorted according to some order O as depicted in Example 1, we have $maxpref(q(s_1))$ = {*Ka*, *ai*, *is*, *se*} and $midpref(q(s_1))$ ={*Ka*,*ai*, *is*}.

Feature ordering can be further exploited to improve performance. Because O 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 $x = \{f_1, \ldots, f_{|x|}\}$, where the subscripts represent the feature position in the set, let $rem(x, i)$ denote the number of features following the feature f_i in x; thus, $rem(x,i) = |x| - i$. We can also rearrange the sets in C according to a specific order, namely the *feature frequency ordering*, O_f, to obtain sets ordered by increasing feature frequencies. The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by shifting lower frequency features to the prefix positions [3].

2.2. The ppjoin algorithm

Algorithm 1. The ppjoin algorithm

```
   Input: A set collection C, a threshold γ
   Output: A set S containing all pairs (x_p,x_c) such that Sim(x_p,x_c) ≥ γ
1  I_1,I_2,...,I_|U| ← ∅, S ← ∅
2  foreach x_p ∈ C do
3      M ← empty map from set id to (os,i,j)   // os = overlap score
4      foreach f_i ∈ 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) ∈ I_f do
7              M(x_c) ← (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 ← −∞   // do not consider x_c anymore
10     S ← S ∪ Verify(x_p,M,γ)   // verification phase
11     foreach f_i ∈ midpref (x_p) do // index x_p
12         I_f ← I_f ∪ {(x_p,i)}
13 return S
```

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. Henceforth, we assume that the set collection C is sorted in increasing order of the set sizes as well as each set is sorted according to the total order O_f.

The top-level loop of *ppjoin* scans the dataset C, 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 Section 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 and 12). Note that the algorithm also indexes the 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).

Note that Algorithm 1 has a different notation from that of Xiao et al. [13] and also present some minor modifications. First, we use the notation introduced in the previous section for overlap bound, size bounds, and prefixes. In the original paper, the authors presented an instantiation of *ppjoin* for the Jaccard; for example, given a set x, they used $\gamma|x|$ for the lower bound of the set size, whereas, here, we generally use $minsize(x)$. Further, we store positional information in the hash-based map. As we will see shortly, this information is used in the *Verify* procedure to find the position of the last feature matched in the candidate generation phase (for both sets of each candidate pair). In their paper, Xia et al. used the accumulated overlap score to (approximately) obtain this information (see [13, Algorithm 2, lines 8 and 12]). Finally, we incorporated the mid-prefix optimization in our algorithm. Note that mid-prefix corresponds to the optimization presented in [13, Lemma 3] (again, instantiated for Jaccard).

3. Generalizing prefix filtering

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) 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 [9,16]. Let us 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 $maxpref(x)$ (Algorithm 1, line 11). We call this version *u-ppjoin*, because it 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 (Algorithm 1, lines 8–9). Fig. 1(a) shows the results of both algorithms w.r.t. the 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 Section 7). As we see in Fig. 1, *u-ppjoin* indeed reduces the amount of candidates, especially for lower similarity thresholds, thereby reducing the verification workload.[3] However, the run time results showed in Fig. 1(b) are reversed: *u-ppjoin* is considerably slower than *ppjoin*. Similar results were reported by Bayardo et al. [12] 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 the opposite way.

3.2. The min-prefix concept

A set x_c is indexed by appending a pointer to the inverted lists associated with features $f_j \in midpref(x_c)$, 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 x_p s.t. $minsize(x_p) > |x_c|$, when this reference is eventually removed by size bound checking (Algorithm 1, line 5). We call the interval between the indexing of the set x_c and the last set x_p with $minsize(x_p)$ not greater than 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 $pref(x_c)$ is determined by the lower bound of pairwise

[3] 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 the verification phase.

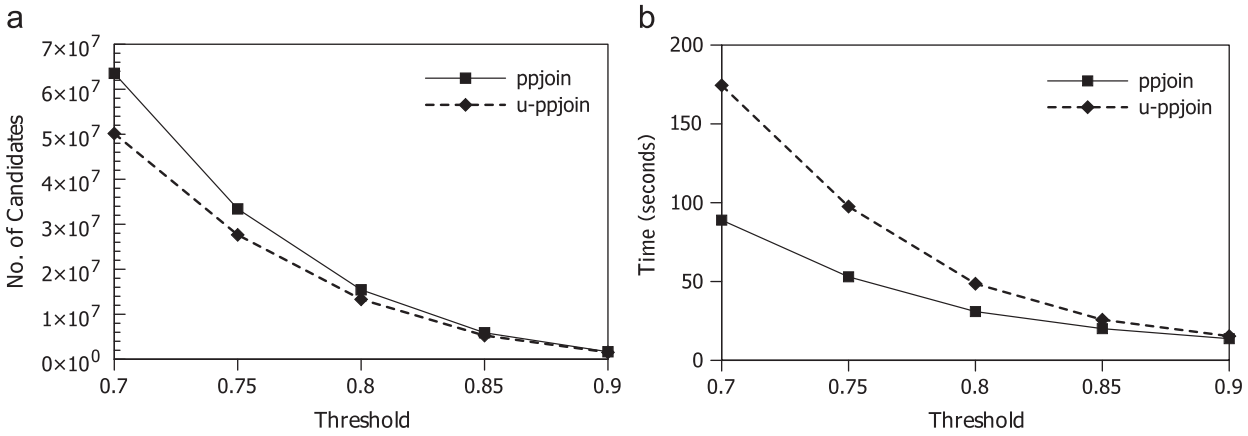

Fig. 1. Number of candidates vs. runtime efficiency. (a) No. of candidates: Jaccard on DBLP. (b) Runtime efficiency: Jaccard on DBLP.

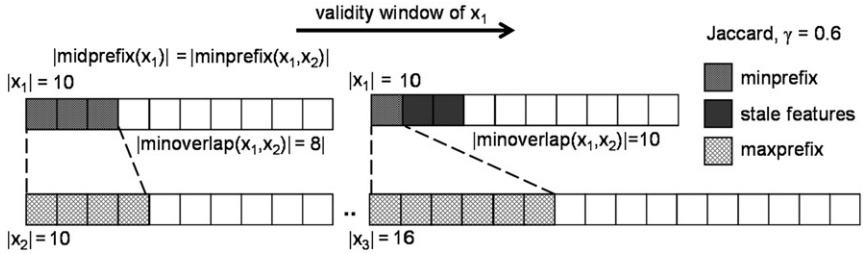

Fig. 2. Min-prefix example.

overlap between x_c and a reference set x_p. As our key observation, the minimal size of $pref(x_c)$ 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 $midpref(x_c)$ no longer suffices alone to elect x_c as a candidate. More specifically, we introduce the concept of *min-prefix*, formally stated as follows.

Definition 5 (*Min-prefix*). Let x_c be a set and let $pref(x_c)$ be a prefix of x_c. Let x_p be a reference set. Then $pref(x_c)$ is a min-prefix of x_c relative to x_p, denoted as $minpref(x_c, x_p)$, iff $1 + rem(x_c, j) \geq minoverlap(x_p, x_c)$ holds for all $f_j \in pref(x_c)$.

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 $f_j \notin minpref(x_c, x_p)$, 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 $minpref(x_c, x_p)$, a *stale feature* relative to x_p.

Example 3. Fig. 2 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 $midpref(x_1)=3$, 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. 3. 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 the (shorter) sets already indexed. To index x, the set collection sort order allows reducing the prefix to $midpref(x)$. The prefixes maxprefix and midprefix are statically defined and fixed-sized. 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; min-prefix is dynamically defined and variable-sized. The following lemma states important properties of stale features according to the set collection and the feature ordering.

Lemma 1. *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

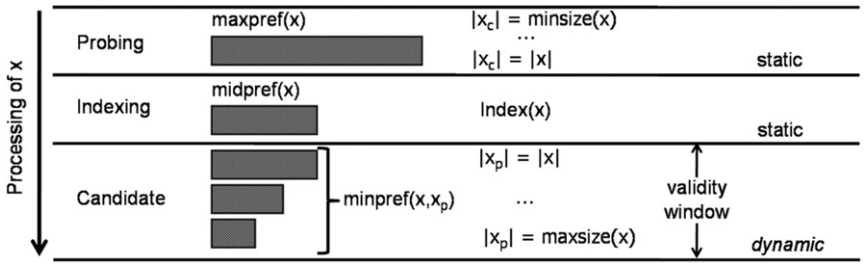

Fig. 3. Min-prefix generalization of prefix filtering.

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 Section 3.1, such overhead in candidate generation can have a negative impact on the overall runtime efficiency.

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

Algorithm 2. The mpjoin algorithm

```
   Input: A set collection C, a threshold γ
   Output: A set S containing all pairs (x_p, x_c) such that sim(x_p,x_c) ≥ γ
1  I_1,I_2,...,I_|U| ← ∅, S ← ∅
2  foreach x_p ∈ C do
3      M ← empty map from set id to (os,i,j)   // os = overlap score
4      foreach f_i ∈ 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) ∈ I_f do
7              if x_c.prefsize < j
8                  Remove (x_c,j) from I_f      // I(x_c,j) is stale
9                  continue
10             M(x_c) ← (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 ← −∞      // 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 ← |x_c| − minoverlap(x_p,x_c) + 1   // update prefsize
16     S ← S ∪ Verify(x_p,M,γ)      // verification phase
17     x_p.prefsize ← |midpref(x)|      // set prefix size information
18     foreach f_i ∈ midpref(x_p) do      // index x_p
19         I_f ← I_f ∪ {(x_p,i)}
20 return S
```

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 size 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 7–9). 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), stale feature can only appear as a first common element. In this case, it follows from Definition 5 that the overlap constraint cannot be met and the set reference can be removed from the list (lines 13 and 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

In this section, we discuss the verification phase and propose a modification to *mpjoin* concerning the optimization of overlap score accumulation.

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.

Algorithm 3. The *Verify* algorithm

```
   Input: A probing set x_p; a map of candidate sets M; a threshold γ
   Output: A set S containing all pairs (x_p,x_c) such that sim(x_p,x_c) ≥ γ
1  S ← ∅
2  foreach x_c ∈ Ms.t. (overlap ← M(x_c).os) ≠ −∞ do
3      if (f_c ← featAt(x_c,x_c.prefsize)) < (f_p ← featAt((,x)_p,|maxpref(x)|))
4          f_p ← featAt(x_p,M(x_c).i+1),f_c++
5      else
6          f_c ← featAt(x_c,M(x_c).j+1),f_p++
7      while f_p ≠ end and f_c ≠ 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 ≥ minoverlap(x_p,x_c)
13         S ← S ∪ {(x_p,x_c)}
14 return S
```

In Algorithm 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 3–6). 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 last prefix element. 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 have a last match on a stale feature, i.e., $x_c.prefsize < j$, and miss a stale feature at position $j' < j$, 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 and 13).

4.2. Optimizing overlap score accumulation

Ref. [12] 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. [11]). 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. Practical aspects

In this section, we address two important practical aspects around our *min-prefix* approach, namely: a disk-based external version of *mpjoin* to work with limited memory and data splitting for parallel execution.

5.1. Disk-based external version

So far, we have assumed that there is enough available memory to hold the index through the whole join processing. Obviously, this is an unrealistic assumption, specially when dealing with very large datasets. For this reason, we have adapted the "out-of-core" version of All-pairs [12], which conceptually resembles a block nested-loop join: the algorithm makes multiple passes over the input set collection, where a block of the input is indexed and matched as in the in-memory version at each pass. To produce all the results, the algorithm has a matching-only phase where it continues executing the candidate generation phase and verification phase after the last set in a block has been indexed until the end of the dataset. Here, we can exploit size bounds to devise a simple yet effective optimization. Instead of proceeding with the matching-only phase until the end of the dataset we can terminate the processing of the current block as soon as a set is read whose size is larger than the size upper bound (*maxsize*) of the last set indexed. From this point, we are sure that no other set will be a valid match of any indexed set.

Algorithm 4. The out-of-core variant of mpjoin

```
   Input: A set collection C, a threshold γ, a memory budget parameter
   Output: A set S containing all pairs (x_p,x_c) such that sim(x_p,x_c) ≥ γ
1  I_1,I_2,...,I_|U| ← ∅,S ← ∅,lastIndexedSet ← ∅,indexing ← true,
2  while (x_p ← read()) ≠ eof do
3      if not indexing and maxsize (lastIndexedSet) < |x_p| then
4          I_1,I_2,...,I_|U| ← ∅,indexing ← true
5          x_p ← seek(lastIndexedSet)
6          continue
7      M ← Probe(x_p,γ)        // candidate generation phase
8      S ← S ∪ Verify(x_p,M,γ)      // verification phase
9      if indexing then
10         Index(x_p)
11         if memory budget exceeded then
12             indexing ← false      // enter matching−only phase
13             lastIndexedSet ← x_p
14 return S
```

Algorithm 4 shows the disk-based external version of *mpjoin*. The algorithm has an extra parameter specifying the memory budget. Every time this budget is exceeded the algorithm enters in the matching only phase and saves the last set indexed (lines 12 and 13). The main refinement of the algorithm is the stop condition for the matching-only phase (line 3), as described above. After reading a probing set that is large enough, the algorithm

clears out the index and start a new block following the last set indexed (lines 4–6).

5.2. Parallel execution

The external version of *mpjoin* can process arbitrarily large amount of data. Nevertheless, some sort of parallelism is necessary for dealing with massive datasets. As for the out-of-core version, size bounds provide a natural way to split the input data among multiple processors and memories. This approach was adopted by Theobald et al. [8] in their parallel algorithm; the underlying technique is basically the same as that used by Arasu et al. [2] to divide a Jaccard-based set similarity join instance into a set of smaller Hamming-based ones. In the following, we briefly review this size-based data splitting strategy and discuss its use with *mpjoin*.

First, the set of integers is partitioned into P, where each $P_i \in P$ is defined by the interval $[l_i,r_i]$. Specifically, starting from $P_1=[1,1]$, define $P_i=[r_{i-1}+1,\lfloor maxsize(l_i)\rfloor$ where $maxsize(r)$ is the size upper bound value obtained from a set of size r. Next, subsets $C_1,C_2,\ldots$ of C are constructed as follows: for each set $x \in C$, if $|x| \in P$, then add x to C_i and C_{i+1}. It can be shown that if $x_1 \in C_i$ and $Sim(x_1,x_2) \geq \gamma$, then $x_2 \in C_{i-1} \cup C_i \cup C_{i+1}$ (see [2,8] for details).

We can execute instances of *mpjoin* on each subset C_i independently. Only a minor modification on *mpjoin* is needed to avoid duplicate result pairs. Given a subset C_i, the algorithm starts with an indexing-only phase, where incoming sets are directly indexed (Algorithm 2, lines 17–19) without executing the candidate generation phase and verification phase—contrast this stage with the matching-only phase of the out-of-core variant. The indexing-only phase continues until the first set is seen whose size is no shorter than l_i; afterwards, the algorithm switches to its normal operation. The reason for the indexing-only phase is that the sets shorter than l_i are processed by the mpjoin instance associated with the subset C_{i-1}.

As a consequence of the data splitting, each instance of *mpjoin* processes input data exhibiting more concentrated set size distribution. As we will empirically demonstrate in Section 7, this fact reduces the performance gains of *mpjoin*—and *ppjoin* as well. Note, however, that some subsets C_i may contain very few elements. Hence, in practice, contiguous subsets will be merged to form the input of a *mpjoin* instance. Devising a subset merging strategy that maximizes both performance of set similarity joins and parallelism is a topic for future work.

6. 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 $w(f)$. Weights are used to quantify the importance of features. In many domains, features show non-uniformity regarding some semantic properties, such as discriminating power, and therefore the definition of an appropriate weighting scheme is instrumental in obtaining reasonable results. For instance, the widely used *inverse document frequency* weighting scheme (IDF) in Information Retrieval [18] defines feature weights to be inversely proportional to their frequency, which captures the intuition that rare features are more relevant for similarity assessment.

All concepts presented in Section 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 $WJS(x_1,x_2)=w(x_1 \cap x_2)/w(x_1 \cup x_2)$. The prefix definition has to be slightly modified as well. Given an overlap bound α, the weighted prefix of a set x, denoted as $pref(x)$, is the *shortest* subset of x such that $w(pref(x)) > w(x)-\alpha$.

Algorithm 5. The w-mpjoin algorithm

```
   ...
   foreach f_i ∈ maxpref(x_p) do        // candidate generation phase
6      Remove all (x_c,c(f_j),j) ∈ I_f from I_f s.t. w(x_c) < minsize(x_p)
7      foreach (x_c,c(f_j),j) ∈ I_f do
8          if x_c.prefsize + w(f_j) < c(f_j)
9              Remove (x_c,c(f_j),j) from I_f      //I(x_c,c(f_j),j) is stale
10             continue
11         M(x_c) ← (M(x_c).os + w(f_j),i,j)
12         if M(x_c).os + min(crem(x_p,i),crem(x_c,j)) < minoverlap(x_p,x_c)
13             M(x_c).os ← −∞      //do not consider x_i anymore
14             if M(x_c).os + crem(x_c,j) < minoverlap(x_p,x_c)
15                 Remove (x_c,c(f_j),j) from I_f   //I(x_c,c(f_j),j) is stale
16         x_c.prefsize ← w(x_c) − minoverlap(x_p,x_c)
17     S ← S ∪ Verify(x_p,M,γ)      //verification phase
18     cweight ← 0
19     foreach f_i ∈ midpref(x_p) do // index x_p
20         cweight ← cweight + w(f_i)
21         I_f ← I_f ∪ {(x_p,cweight,i)}
22     x_p.prefsize ← cweight
23 ...
```

We now present the weighted version of *mpjoin*, called *w-mpjoin*. The most relevant modifications are listed in Algorithm 5. 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 $c(f_i)$, for each $f_i \in midpref(x)$ and set *prefsize* to the cumulative weight of the last feature in $midpref(x)$ (lines 18–22). 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 weight to reach the overlap bound (lines 12 and 14). Second, the cumulative weight is used to identify stale features by comparing it with *prefsize* (line 08). Note that the cumulative weight of the last feature in *minpref* (x_c, x_p) is always greater than the current *prefsize*. 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.

For brevity, we do not discuss the weighted version of the Verify procedure, but the modifications needed are straightforward.

7. Experiments

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. We also aim at identifying the most important characteristics of the input data driving the performance of the set similarity joins algorithms under study. To this end, we conduct our study under several different data distributions and configuration parameters using real and synthetic datasets. Finally, we present a scalability study where the out-of-core variant of the algorithms under study is evaluated.[4] All tests were run on an Intel Xeon Quad Core 3350 2,66 GHz, about 2.5 GB of main memory, and using Java Sun JDK 1.6.0.

7.1. Algorithms

We focused on index-based algorithms, because they consistently outperform competitor signature-based algorithms [12] (see discussion in Section 8) 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 explored a *hybrid* version, which combines *mpjoin* and *ppjoin+* by adding the suffix filtering procedure in *mpjoin* (Algorithm 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*. For evaluation of weighted sets, we used the well-known IDF weighting scheme. For brevity, we only report results for the Jaccard similarity. The corresponding results for other similarity functions follow identical trends. In the experiments, we focus on evaluating the performance gains obtained from the optimized candidate generation phase, in particular, the effectiveness of the min-prefix technique. Therefore, we used the improved verification procedure in all algorithms (see Section 4.1).

7.2. Datasets

We used two well-known real datasets: *DBLP* containing information about computer science publications (dblp.uni-trier.de/xml) and *IMDB* containing information about movies (www.imdb.com). To obtain different data distributions, we first derived three subsets from each dataset by selecting different fields from them: title (*DT*), author (*DA*), and their concatenation (*DTA*), for *DBLP*; title (*IT*), actor (*IA*), and their concatenation (*ITA*), for *IMDB*. We randomly selected strings from the corresponding fields, converted them to upper-case letters and eliminated repeated white spaces. Each string is converted into a set of features by tokenizing it into sets of q-grams and using the Karp–Rabin fingerprint function [19] to map each q-gram to a hash value (with small probability of collision). We then ordered the features within a set according to their frequency, and stored the sets in ascending size order. Figs. 4(a) and (b) plot the set size distribution. The values of the *DBLP* datasets and *IT* fit reasonably well to a log-normal model (note the log scale on both axes), whereas those of *ITA* and *IA* (only *ITA* is shown) resembles a power-law relationship. Besides the set size, we also vary the feature frequency distribution of the datasets by using differing q-gram sizes. Fig. 4(c) shows the "count-frequency plot"[5] for q ranging from 2 to 4 on the *DTA* dataset—we obtained similar results with the other datasets. The distributions seem to follow a power-law distribution with exponent α about 1.125,

[4] We do not consider the parallel version of the algorithms in our experiments. The relative performance of the algorithms under parallel execution can be nevertheless roughly estimated from the results presented in this section.

[5] For this plot, we excluded frequencies with count less than 10 to avoid fluctuation effects.

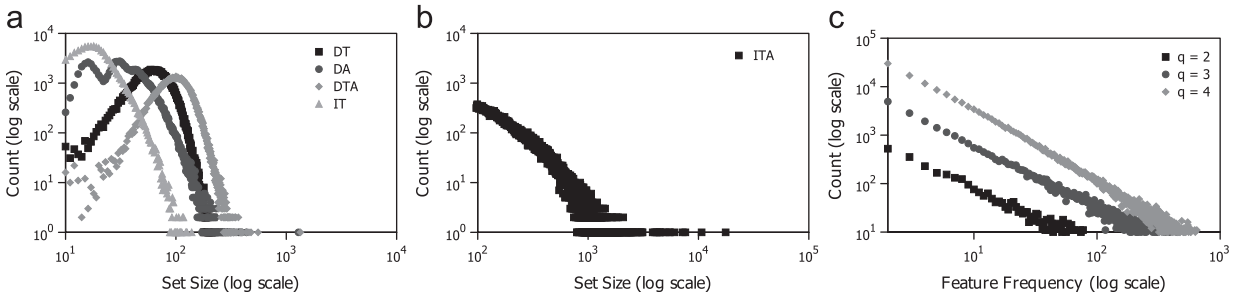

Fig. 4. Set size and feature frequency distributions. (a) Set size distribution, *DBLP* and *IT* datasets. (b) Set size distribution, *ITA* dataset. (c) Count-frequency plot of features for q=2–4, *DTA* dataset.

Table 2
Parameters used in the experiments.

Parameter	μ	σ	α	q	N	γ
Description	Normal mean	Normal std. deviation	Zipf exponent	q-gram size	No. of input sets	Threshold
Range	[50,300]	[10,50]	[0.6,2]	[2,4]	[0.1M,1M]	[0.5,0.9]
Default	100	25	1	2	0.1M	0.75

1.194, and 1.509, for q=2, 3, and 4, respectively,[6] i.e., the skewness increases with the size of q.

In addition, we generated synthetic set collections to have closer control over data distribution and to support our conclusions on real datasets. The details of data generation process are as follows: we first created N sets and inserted them into a list L. Then, we generated one unique feature value v (using sequential numbers) at a time together with its frequency f; for each generated feature v, we randomly selected f sets from L and insert a copy of v into each of them. When a set was entirely filled, we removed it from L and we continued the process until L is empty. Set sizes were drawn from a normal distribution and feature frequency from a Zipf distribution [21]. Table 2 summarizes the parameters used for data generation as well as those regarding the input of the similarity join algorithms, i.e., number of input sets and threshold. We used the default value unless stated otherwise.

7.3. Performance results on synthetic datasets

We first analyze the performance of the algorithms under controlled data distribution parameters, namely, mean and standard deviation of the set sizes, and skew of the feature frequency distribution. We start with the results for the unweighted version of the algorithms, which are shown in Figs. 5(a)–(c). In all settings, *mpjoin* clearly exhibits the best performance. In particular, *mpjoin* achieves performance gains compared to *ppjoin+* about a factor of 2.5 on average. Although the *hybrid* version outperforms *ppjoin+*, it is about 70% slower than *mpjoin* on average. Evidently, the candidate reduction does not pay-off the extra-effort of the suffix filtering (we emphasize this observation when we detail the workload of the algorithms).

The performance of all algorithms severely degrades as the mean set size increases (Fig. 5(a)). This effect is not a surprise, because larger sets translate into larger prefixes and therefore more features are used to probe the index in the candidate generation phase. Moreover, larger subsets have to be processed in the verification phase. Another crucial aspect is inverted list reduction. Because we increased the mean of the set size distribution while maintaining its standard deviation constant ($\sigma = 25$), dynamic removal of indexed sets by size bound and *min-prefix* checking turned out to be less effective. For example, consider a set x with size 300, therefore $minsize(x)$=225. For $\mu = 300$, nearly all the sets have sizes not smaller than $minsize(x)$, i.e., sizes are at most three standard deviations smaller than the mean size (recall that set sizes are normally distributed in this experiment). Likewise, we have $maxsize(x)$=400; hence, the validity window of x will last until the end of the dataset. In this connection, it is easy to see that the worst-case scenario is an equi-sized set collection: $minpref(x)$ would be equal to $midpref(x)$ along the whole validity window of x and $minsize(x)$ checking would be useless, i.e., dynamic index reduction would not be possible.

Fig. 5(b) plots the results with varying σ. All algorithms run significantly faster as the standard deviation increases, which confirms the influence of the size distribution spread on performance. Particularly, the performance gain of *mpjoin* over *ppjoin+* increases from 1.8 to 2.7 as σ increases from 10 to 50. This improvement is due to the increased number of set entries associated with stale

[6] We used a traditional and simple procedure to model the feature frequency distribution with a power-law distribution: we fit a straight line on the (log–log) "count-frequency plot" using least-square linear regression and took the absolute slope of the straight line as the exponent α. Note that accurately estimating α as well as making a strong case for a power-law distribution against competing distributions is a difficult problem. In [20] , the authors propose maximum likelihood estimators and goodness-of-fit tests based on the Kolmogorov–Smirnov measure and likelihood ratios. Here, an approximate modeling is nevertheless sufficient for the purposes of our discussion.

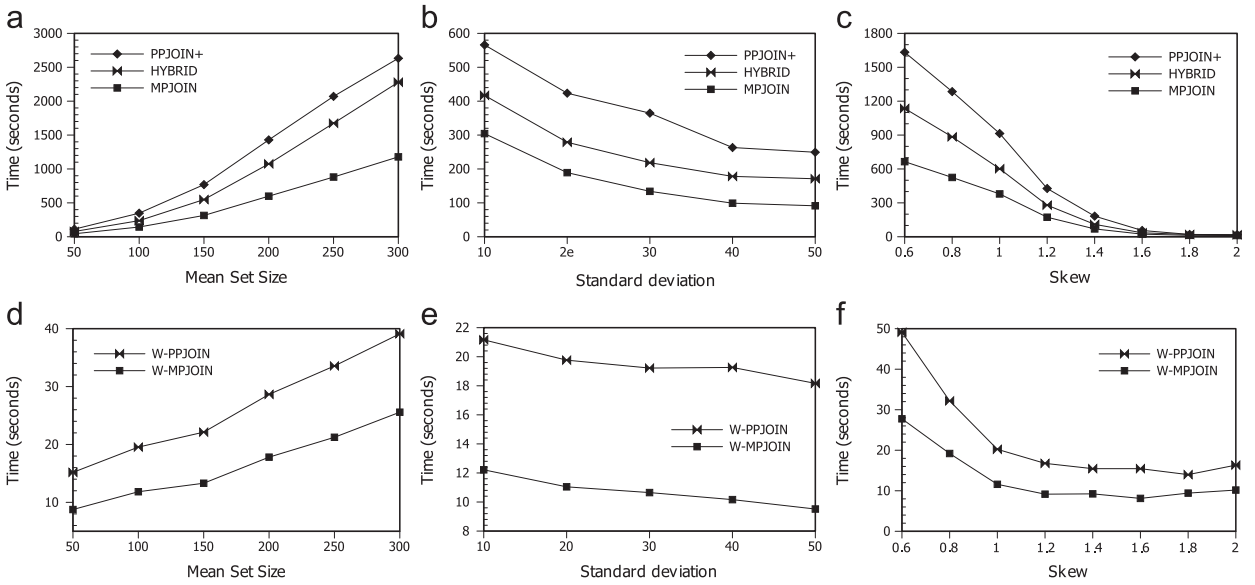

Fig. 5. Performance results on synthetic data. (a) Unwei. sets, varying μ. (b) Unwei. sets, varying σ. (c) Unwei. sets: varying α. (d) Weighted sets, varying μ. (e) Weighted sets, varying σ. (f) Weighted sets, varying α.

features in the index that degrades the performance of *ppjoin+*, but are removed by *mpjoin*.

Fig. 5(c) shows the results with varying skew. Again, *mpjoin* achieves more than twofold speed-ups on average over *ppjoin+*. Furthermore, the runtime of all algorithms is drastically reduced as the skew increases. The reason for this improvement is that there are an increased number of low-frequency features with higher skew, which are placed at the prefixes due to feature frequency ordering. As a result, the inverted lists are shorter and there is much less prefix overlap between dissimilar sets, thereby decreasing the number of generated candidates.

Figs. 5(d)–(f) plot the results for weighted sets. As for unweighted sets, *w-mpjoin* is faster than *w-ppjoin* in all settings achieving up to twofold speed-ups. Notably, the algorithms are considerably faster than those for the unweighted case, because the weighting scheme results in shorter prefixes. In general, we observe the same trends for weighted sets: performance worsens with larger sets, but improves at higher set size variance and feature frequency skew. However, data distribution variation affects the algorithms by a lesser degree owing to the reduced prefix size.

7.4. Performance results on real datasets

We now analyze the efficiency of the set similarity join algorithms using real datasets. Fig. 6(a) shows the runtime performance using unweighted sets. The trends observed are similar to those on the synthetic data: *mpjoin* is the best performing algorithm, *ppjoin+* the worst. On all datasets, *mpjoin* is more than two times faster than *ppjoin+*, achieving more than a threefold speed-up over *ppjoin+* on *IA* and *ITA* datasets.

The reasons for the above results are revealed in Fig. 6(b), which illustrates the workload on the candidate generation and verification phases, i.e., the number of sets processed in these phases. In the chart, *PROBED* corresponds to indexed sets appearing in the inverted lists during the candidate generation phase. Because of the partial overlap score accumulation when generating candidates, some sets are immediately pruned at the first overlap bound checking in the verification phase without further processing (see Algorithm 3, line 10); these sets are represented by *P-VERIFIED*. Finally, *VERIFIED* corresponds to the sets that are actually scanned in the verification phase including those that will be part of the output. Note that *mpjoin*, *hybrid*, and *ppjoin+* are abbreviated in the chart by *M*, *H*, and *P*, respectively. In the candidate generation phase, *ppjoin+* doubled the number of indexed sets needed by *mpjoin*. The extra indexed sets of *ppjoin+* are related to stale features, i.e., irrelevant candidates that are repeatedly considered along their validity window. Together with the elimination of dedicated data structures for score accumulation, the decreased number of sets processed by *mpjoin* dramatically reduces the computational cost for candidate generation. As expected, the number of sets delivered to the verification phase by *mpjoin* is larger. Moreover, because fewer features are considered, score accumulation is reduced when generating candidates. As a result, *P-VERIFIED* is negligible for *mpjoin*, i.e., nearly all sets passed on to the verification phase have to be processed. But now the optimization employed in the verification phase (see Section 4.1) comes into play and the higher workload does not translate into an overwhelming performance penalty. For instance, consider the *DTA* data set. Even though *VERIFIED* for *mpjoin* is about 18 $\times$ larger compared to *ppjoin+*, the overall execution runtime of *mpjoin* is about 2.8 $\times$ shorter. The advantage of faster

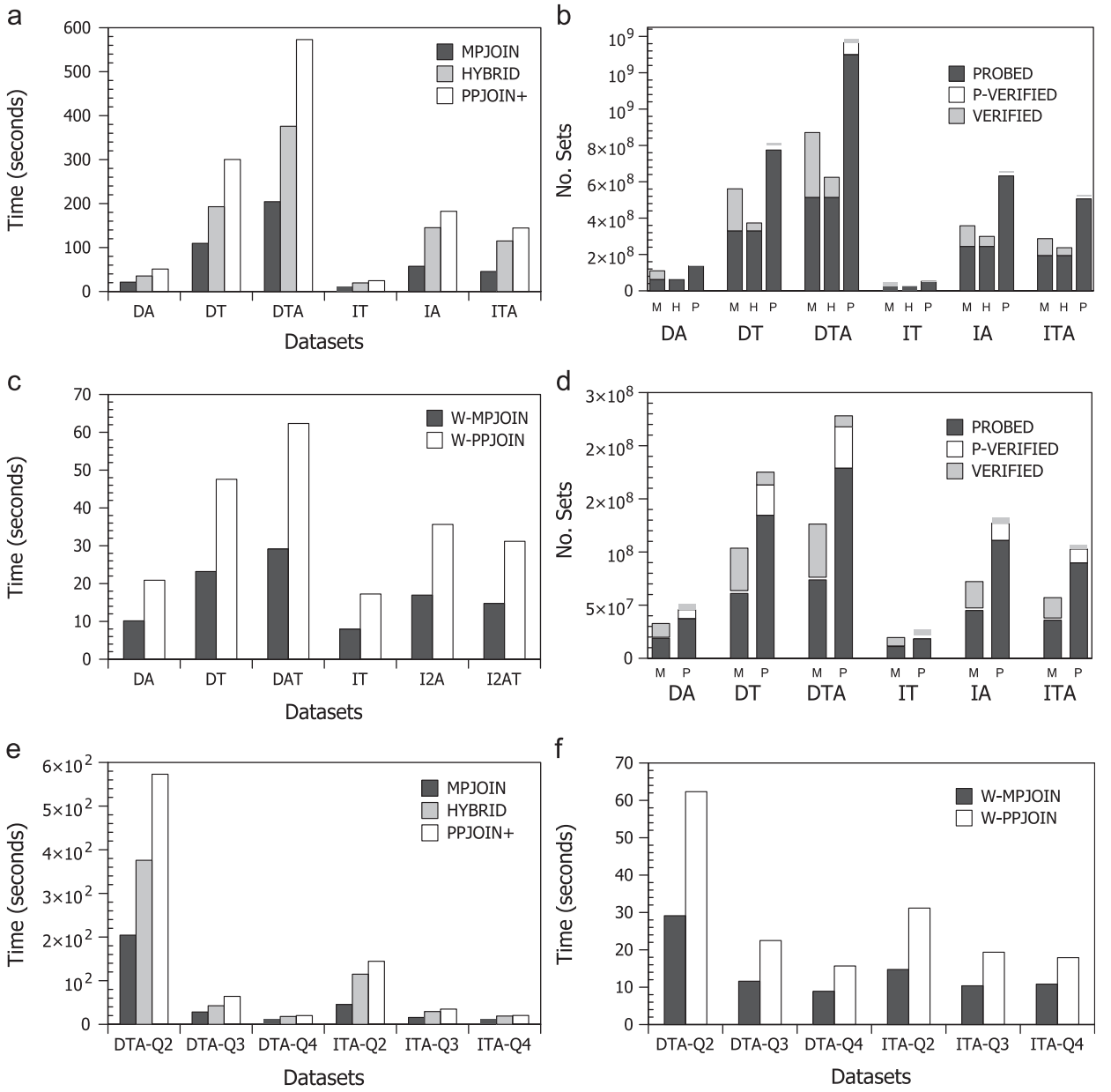

Fig. 6. Performance results on real data. (a) Runtime, unweighted sets. (b) Workload, unweighted sets. (c) Runtime, weighted sets. (d) Workload, weighted sets. (e) Varying q size, unweighted sets. (f) Varying q size, weighted sets.

candidate generation is made explicit when comparing *mpjoin* to *hybrid*. *PROBED* is the same for both algorithms, but *VERIFIED* is about 5.6× shorter for hybrid due to suffix filtering. However, this saving is ineffective: the overall runtime of *hybrid* is about the double of that of *mpjoin*.

The performance of the algorithms across the datasets is dictated by the underlying data distribution. The set size distribution of the *DBLP* datasets and *IT* have similar shape (see Fig. 4(a)) and the runtime of the algorithms closely follow the mean set size. On the other hand, all algorithms are faster on *IA* and *ITA* compared to *DT* and *DTA*. Although *IA* and *ITA* contain some very large sets, their set size distribution is more dispersed than those of *DA* and *DTA*. As a result, the validity window of indexed sets is shorter and more entries in the inverted lists are removed due size bound checking. Also, features become stale more quickly within the validity window, which favors *mpjoin*: the performance gap between *mpjoin* and *ppjoin+* is larger on *IA* and *ITA*.

Fig. 6(c) shows the results for weighted sets. The trends are similar to the ones for unweighted sets. *w-mpjoin* outperforms *w-ppjoin* by a factor larger than 2 in all measurements and the performance of the algorithms across the datasets follows the respective data distribution. As observed on the synthetic datasets,

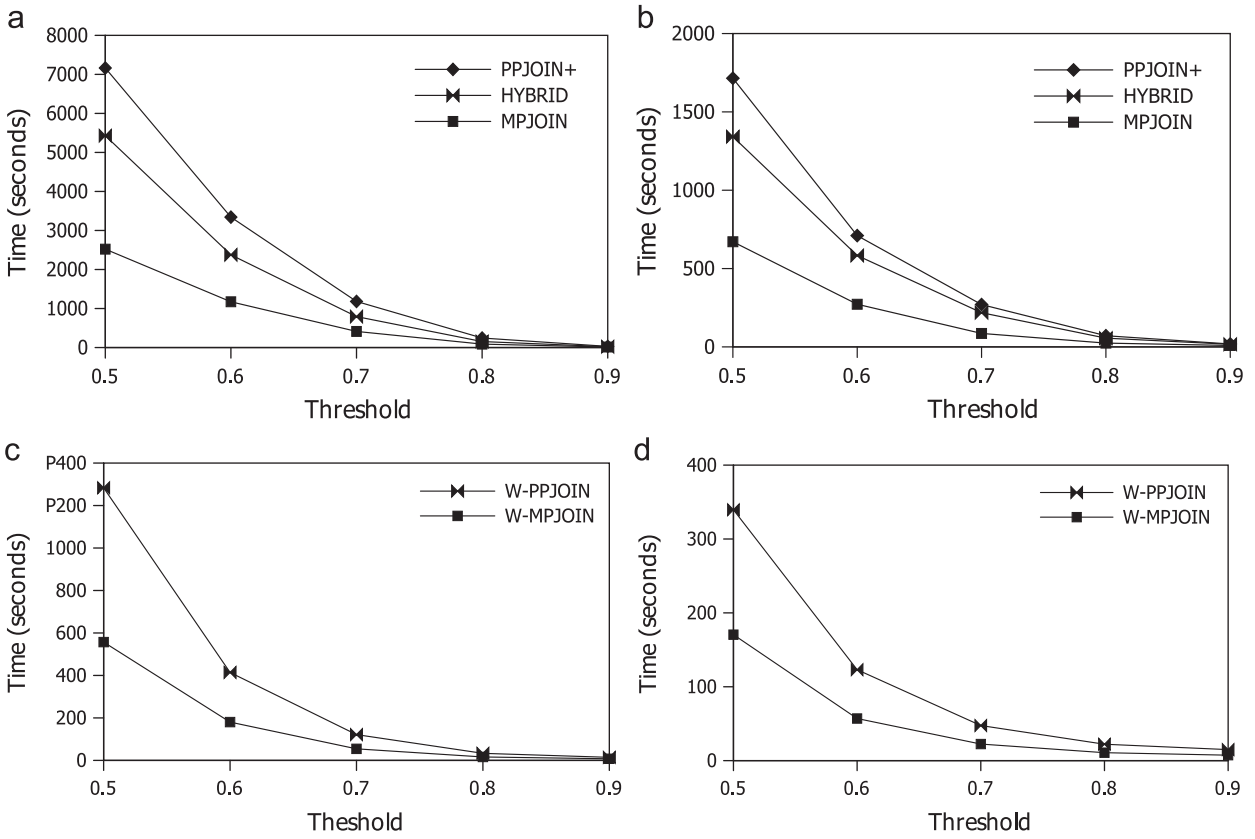

Fig. 7. Performance results with varying threshold. (a) DTA dataset, unweighted sets. (b) ITA dataset, unweighted sets. (c) DTA dataset, weighted sets. (d) ITA dataset, weighted sets.

all algorithms are much faster on weighted sets (about one order of magnitude), which is explained by the lower workload owing to shorter prefixes (see Fig. 6(d)).

We also analyzed the performance of the algorithms under different feature frequency distributions by varying the size of q. As mentioned previously, the skew increases with the size of q. Figs. 6(e) and (f) show the results on *DTA* and *ITA* datasets for unweighted and weighted sets, respectively—we obtained similar results on the other datasets. As for synthetic data, all algorithms become much faster as the data becomes more skewed. Note, the performance advantage of *mpjoin* is more prominent when the skew is lower. For instance, on the ITA dataset, the performance gains increase from about 1.7 × with q=4 to 3.1 × with q=2, for unweighted sets; for weighted sets, the increase is from 1.7 × to 2.1 ×. This observation is of particular importance, because many application domains are characterized by relatively low-skewed data. For example, [15] recommends q-grams of size 2 to obtain the best quality results in a data cleaning scenario.

Finally, we measured the runtime performance with varying threshold parameter. Figs. 7(a) and (b) show the results on the *DTA* and *ITA* for unweighted sets, respectively; Figs. 7(c) and (d) show the results on the same datasets for weighted sets. *mpjoin* and *w-mpjoin* remain faster than their competitors throughout the whole threshold range on both datasets. All algorithms considerably increased their runtime as the threshold decreases (two orders of magnitude from 0.9 to 0.5), mainly because lower thresholds imply larger prefixes.

7.5. Scalability experiments

We conducted scalability tests on datasets varying from 100K to 1000K. We also evaluated the performance of the disk-based version of the algorithms by restricting the memory budget such that only 200K input sets could be dynamically indexed and kept memory-resident; for larger numbers of input sets, the algorithms had to scan the disk-resident input data multiple times to complete the operation as described in Section 5.1. We report the results on synthetic data with default parameters (e.g., $\mu = 100$, $\sigma = 25$, $\theta = 1$) and on the *DTA* dataset. The algorithms were configured to stop indexing and enter the matching-only phase after indexing (0.35M) 1.2M features for (weighted) unweighted sets on the synthetic dataset; (1.3M) 3.1M features for (weighted) unweighted sets on the *DTA* dataset (these values roughly reflect the number of features indexed on an input containing 200K sets).

Figs. 8(a) and (b) show the results on the synthetic dataset and *DTA* for unweighted sets, respectively; Figs. 8(c) and (d) show the results on the same datasets for weighted sets. The runtime of all algorithms on both datasets exhibits

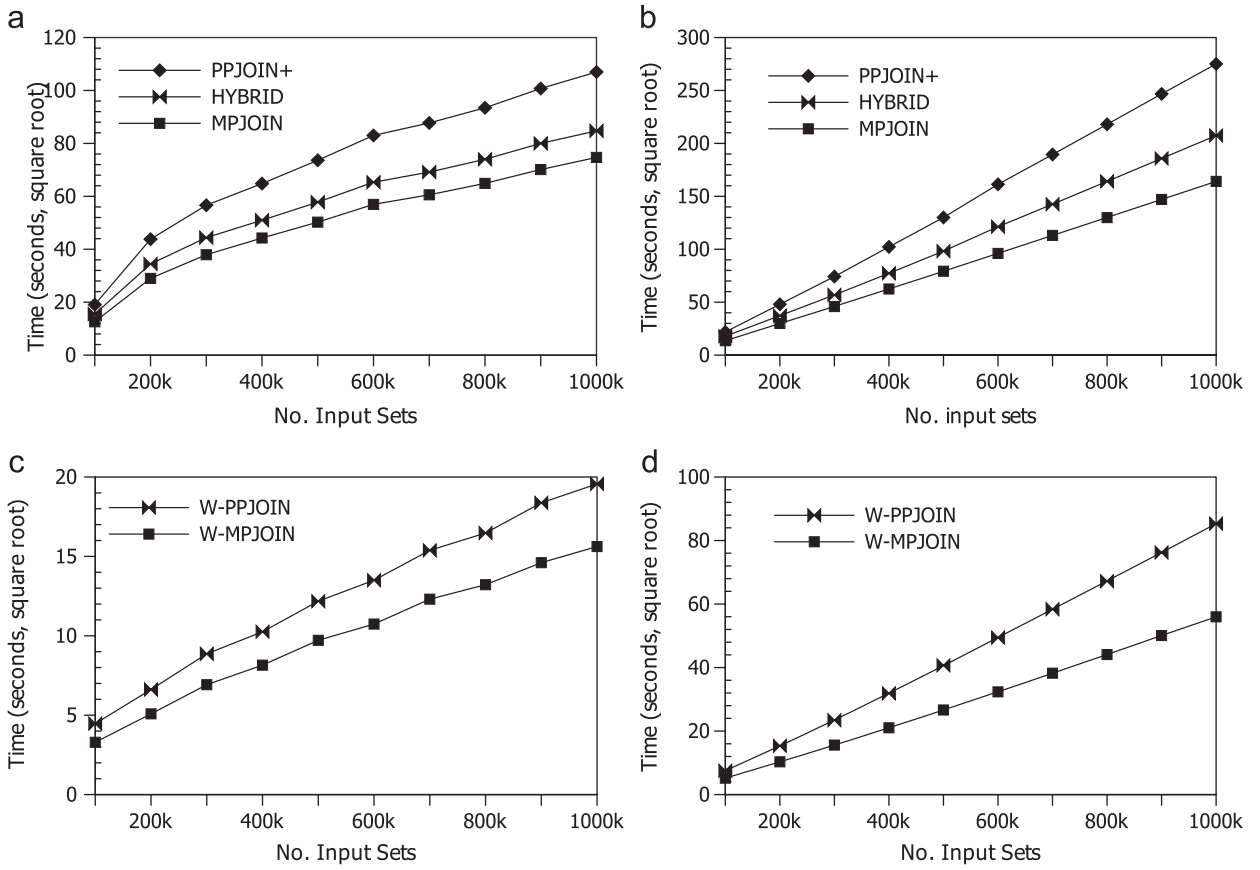

Fig. 8. Scalability results using synthetic and real datasets. (a) Synthetic dataset, default parameters, unweighted sets. (b) DTA dataset, unweighted sets. (c) Synthetic dataset, default parameters, weighted sets. (d) DTA dataset, weighted sets.

a quadratic growth (note that we show the square root of the runtime). This behavior is expected because the workload in terms of set candidates maintained and processed by set similarity join algorithms also grows quadratically with the input size (also observed in [13]). The growth rate of all algorithms is quite similar and their relative performance practically stays constant as the input size increases. Finally, the IO overhead of the disk-resident variant incurs only a little performance penalty as we do not observe any significant degradation on input data containing more than 200K sets.

7.6. Experimental summary

In all measurements performed on synthetic and real datasets, *mpjoin* and *w-mpjoin* provided a superior performance than their competitors. We have shown that a large part of the sets processed during the candidate generation phase are indeed associated with stale features, i.e., these sets only add unnecessary overhead, because they cannot be part of the final result. The optimizations in the verification phase turned out to be effective: the runtime did not blow up even with a dramatic 18 $\times$ increase of the number of candidates. As opposed, suffix filtering was ineffective: the reduction of the candidates did not compensate the increased runtime in the candidate generation phase—even when employed together with the min-prefix technique. This fact emphasizes our observation in Section 3 that reduction of the candidates should be considered with care, because it does not always translate into performance gains.

The spread of the set size distribution has more impact on the performance than the set size itself. The algorithms run faster on datasets exhibiting highly dispersed set sizes than datasets containing more uniform set sizes. In particular, the performance gain of *mpjoin* and *w-mpjoin* increases with the set size variance, because features associated with indexed sets became stale more quickly in the course of processing. The performance increases with the skew of the feature frequency distribution because inverted lists become shorter. On the other hand, the advantage of using the min-prefix technique to minimize of length of the inverted lists diminishes. The runtime of all algorithms grows quadratically as the input set increases, as expected. Finally, by varying input size as well as the threshold parameter, both *mpjoin* and *w-mpjoin* steadily revealed their performance advantage.

8. Related work

There is a vast body of literature on performing similarity joins in vector spaces; in this context, a similarity

join is a variant of the more general approach known as *spatial join*. See [22] for a recent survey. Indexing techniques for vector spaces are well-suited for implementing similarity joins in application domains where the objects can be described by low-dimension feature vectors and the notion of similarity can be expressed by a distance function of the Minkowski family, such as the Euclidean distance (L_2 norm). However, all these techniques suffer from the "curse of dimensionality", meaning that their performance degrades as the underlying dimensionality increases; they are often outperformed by sequential scans—already for as few as 10 dimensions [10].

Very often, real world datasets present strong correlations between various dimensions or contain unimportant dimensions (e.g., those not exhibiting any discriminating power) thereby effectively reducing the intrinsic dimensionality compared to the dimensionality of the underlying space. In such cases, dimension reduction methods [23,24] and embedding methods (e.g., FastMap [25]) can be employed to uncover the intrinsic dimensionality of a dataset. However, in domains such as text data applications, an effective dimensionality in the order of hundreds has to be anticipated, even after aggressive dimension reduction (about 90%) [24]. This fact motivates the use of inherently approximating approaches. To this end, locality-sensitive hashing (LSH) [26] has been widely used, which is based on hashing functions that are approximately distance-preserving.

In many applications, the objects of interest cannot be properly represented as a collection of features or distance functions of the Minkowski family are not appropriate. An alternative method for such situations is to resort to generic metric spaces where the only information available is the pairwise distance between objects given by a *distance metric* [27]. In this context, a common approach consists of using embedding methods to map object from the original metric space into a vector space; afterwards, spatial indexes can be used to save distance calculations. See [28,29] for such approaches on text and XML documents, respectively. Other techniques reduce the dependency on the parameter space by avoiding the use of pre-built indexes. For example, Ref. [30] presents the Quickjoin algorithm, which recursively partitions the data until each partition is sufficiently small such that nested-loop joins can be applied in an economical way. Quickjoin has been shown to perform better than competing methods without index support. Note that the performance of all metric-space methods heavily relies on a good pivot selection strategy. Unfortunately, this problem is not well understood and, in practical applications, pivots are randomly chosen even though it is likely to lead to suboptimal results [27].

Set similarity joins embody an important method to implement similarity joins when the objects of interest lend themselves to set representation and exhibit high dimensionality, such as text data (see other examples of application areas in Section 1). In this context, a rich variety of optimizations has been proposed—most of them were discussed in Section 2 and are used by our algorithms: derivation of bounds (e.g., size bound [2,8,11–13]), exploitation of a specific data set order [11–13] and feature positional information [13], and signature schemes [2,3]. The prefix-filtering concept was first exploited to improve set similarity joins in [11] and formally defined in [3]; in these contributions, prefix-filtering is used without any information concerning set sort order and therefore corresponds to our definition of *maxprefix*. The use of smaller prefixes for indexing, i.e. *midprefix*, was first employed in [12] and formally defined in [17].

There are two main query processing models for set similarity joins. The first one 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 supported by clustered indexes are used to identify all pairs sharing signatures, and grouping and aggregation operators together with user-defined functions (UDFs) are used for verification [2,3]. The second model is related to Information Retrieval techniques. An index is built for mapping features to the list of objects containing that feature [11–13]. 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 [12] (see also [31] 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 [12,13], 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, both sets representing 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. Inverted lists have also been shown to perform better than signature-based approaches on simpler kinds of set joins, such as strict containment and non-zero overlap joins [32].

In addition to exact optimizations, dimension reduction methods can be used to speed-up set similarity join processing at the cost of producing approximate results (some valid object pairs may be missed). LSH is the most popular technique for approximate similarity joins: it can be used to reduce the size of the input sets [7] as well as signature schemes [33]. Not all set similarity functions admit an LSH hash function family, however. A required property is that the distance formulation of a similarity function *Sim*, i.e. $1-Sim$, must satisfy the triangle inequality [34]. For example, Jaccard admits an LSH hash function family, whereas there is not such a function for Dice [34].

Text similarity joins were first proposed in the context of data integration in the seminal work of Cohen [4]. In this context, the string edit distance is typically used as

the similarity join predicate. A common approach consists of mapping strings to sets of q-grams and using set-overlap as edit distance constraint. The work in [5] implements this approach on top of a relational query engine. Xiao et al. [14] exploit the locations and contents of mismatching q-grams to improve filtering.

A related line of work focuses on set-similarity selection queries. The work in [31] exploits size bounds of the IDF similarity and feature ordering in inverted lists to design highly efficient algorithms. Li et al. [16] propose algorithms to optimize the merging of inverted lists, which is used to calculate the overlap score of candidates, and investigate different filtering configurations.

Recent trends concerning similarity joins include: exploitation of parallelism and sort and searching capabilities of GPUs [35]; compact representation of similarity join results [36]; and text similarity joins using string transformations [37]. Moreover, following the idea presented by Chaudhuri et al. [3], recent work addresses the integration of similarity joins into DBMS query engines. For example, [38] exploits Power-law distributions to estimate the sizes of set similarity join outputs and [39] introduces transformation rules for the optimization of logical query plans containing similarity join operators.

9. Conclusion

In this paper, we proposed a new index-based algorithm for set similarity joins. Following a completely different approach compared to previous work, we focused on a reduction of the computational cost for candidate generation as opposed to lowering the 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 is 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. Moreover, we improved the overlap score accumulation by storing scores and auxiliary information within the indexed set itself instead of using a hash-based map. Finally, for dealing with massive datasets, we presented variants of our algorithm for disk resident data and parallel execution. Our experimental results on synthetic and real datasets confirm that the proposed algorithm consistently outperforms the known ones for both unweighted and weighted sets.

References

[1] L.A. Ribeiro, T. Härder, Efficient set similarity joins using min-prefixes, in: Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), 2009, pp. 88–102.

[2] A. Arasu, V. Ganti, R. Kaushik, Efficient exact set-similarity joins, in: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB 2006), 2006, pp. 918–929.

[3] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), 2006, p. 5.

[4] W.W. Cohen, Integration of heterogeneous databases without common domains using queries based on textual similarity, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1998, pp. 201–212.

[5] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001), 2001, pp. 491–500.

[6] E. Spertus, M. Sahami, O. Buyukkokten, Evaluating similarity measures: a large-scale study in the orkut social network, in: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2005, pp. 678–684.

[7] A.Z. Broder, On the resemblance and containment of documents, in: Proceedings of the International Conference on Compression and Complexity of Sequences (SEQUENCES'97), 1997, pp. 21–29.

[8] M. Theobald, J. Siddharth, A. Paepcke, Spotsigs: robust and efficient near duplicate detection in large web collections, in: Proceedings of the 31st International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2008), 2008, pp. 563–570.

[9] K. Chakrabarti, S. Chaudhuri, V. Ganti, D. Xin, An efficient filter for approximate membership checking, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2008), 2008, pp. 805–818.

[10] R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98), 1998, pp. 194–205.

[11] S. Sarawagi, A. Kirpal, Efficient set joins on similarity predicates, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2004), 2004, pp. 743–754.

[12] R.J. Bayardo, Y. Ma, R. Srikant, Scaling up all pairs similarity search, in: Proceedings of the 16th International Conference on World Wide Web (WWW 2007), 2007, pp. 131–140.

[13] C. Xiao, W. Wang, X. Lin, J.X. Yu, Efficient similarity joins for near duplicate detection, in: Proceedings of the 17th International Conference on World Wide Web (WWW 2008), 2008, pp. 131–140.

[14] C. Xiao, W. Wang, X. Lin, Ed-join: an efficient algorithm for similarity joins with edit distance constraints, Proceedings of the VLDB Endowment (PVLDB) 1 (1) (2008) 933–944.

[15] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, D. Srivastava, Benchmarking declarative approximate selection predicates, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2007, pp. 353–364.

[16] C. Li, J. Lu, Y. Lu, Efficient merging and filtering algorithms for approximate string searches, in: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), 2008, pp. 257–266.

[17] C. Xiao, W. Wang, X. Lin, H. Shang, Top-k set similarity joins, in: Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), 2009, pp. 916–927.

[18] S.E. Robertson, K.S. Jones, Relevance weighting of search terms, Journal of the American Society for Information Science 27 (3) (1976) 129–146.

[19] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development 31 (2) (1987) 249–260.

[20] A. Clauset, C.R. Shalizi, M.E.J. Newman, Power-law distributions in empirical data, SIAM Review 51 (4) (2009) 661–703.

[21] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger, Quickly generating billion-record synthetic databases, 1994, pp. 243–252.

[22] E.H. Jacox, H. Samet, Spatial join techniques, ACM Transactions on Database Systems (TODS) 32 (1) (2007) 7.

[23] D. Barbará, W. DuMouchel, C. Faloutsos, P.J. Haas, J.M. Hellerstein, Y.E. Ioannidis, H.V. Jagadish, T. Johnson, R.T. Ng, V. Poosala, K.A. Ross, K.C. Sevcik, The New Jersey data reduction report, IEEE Data Engineering Bulletin 20 (4) (1997) 3–45.

[24] Y. Yang, J.O. Pedersen, A comparative study on feature selection in text categorization, in: Proceedings of the 14th International Conference on Machine Learning (ICML 1997), 1997, pp. 412–420.

[25] C. Faloutsos, K.-I. Lin, Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1995, pp. 163–174.

[26] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99), 1999, pp. 518–529.

[27] E. Chávez, G. Navarro, R.A. Baeza-Yates, J.L. Marroquín, Searching in metric spaces, ACM Computing Surveys 33 (3) (2001) 273–321.
[28] L. Jin, C. Li, S. Mehrotra, Efficient record linkage in large data sets, in: 8th International Conference on Database Systems for Advanced Applications (DASFAA'03), 2003, p. 137.
[29] S. Guha, H.V. Jagadish, N. Koudas, D. Srivastava, T. Yu, Integrating xml data sources using approximate joins, ACM Transactions on Database Systems 31 (1) (2006) 161–207.
[30] E.H. Jacox, H. Samet, Metric space similarity joins, ACM Transactions on Database Systems (TODS) 33 (2) (2008).
[31] M. Hadjieleftheriou, A. Chandel, N. Koudas, D. Srivastava, Fast indexes and algorithms for set similarity selection queries, in: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), 2008, pp. 267–276.
[32] N. Mamoulis, Efficient processing of joins on set-valued attributes, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 2003, pp. 157–168.
[33] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Mootwani, J.D. Ullman, C. Yang, Finding interesting associations without support pruning, IEEE Transactions on Knowledge and Data Engineering (TKDE) 13 (1) (2001) 64–78.
[34] M. Charikar, Similarity estimation techniques from rounding algorithms, in: Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC 2002), 2002, pp. 380–388.
[35] M.D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), 2008, pp. 1111–1120.
[36] B. Bryan, F. Eberhardt, C. Faloutsos, Compact similarity joins, in: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), 2008, pp. 346–355.
[37] A. Arasu, S. Chaudhuri, R. Kaushik, Transformation-based framework for record matching, in: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), 2008, pp. 40–49.
[38] H. Lee, R.T. Ng, K. Shim, Power-law based estimation of set similarity join size, PVLDB 2 (1) (2009) 658–669.
[39] Y.N. Silva, W.G. Aref, M.H. Ali, The similarity join database operator, in: Proceedings of the 26th International Conference on Data Engineering (ICDE 2010), 2010, pp. 892–903.