# PEL: Position-Enhanced Length Filter for Set Similarity Joins

Willi Mann
Department of Computer Sciences
Jakob-Haringer-Str. 2
Salzburg, Austria
wmann@cosy.sbg.ac.at

Nikolaus Augsten
Department of Computer Sciences
Jakob-Haringer-Str. 2
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

## ABSTRACT

Set similarity joins compute all pairs of similar sets from two collections of sets. Set similarity joins are typically implemented in a filter-verify framework: a filter generates candidate pairs, possibly including false positives, which must be verified to produce the final join result. Good filters produce a small number of false positives, while they reduce the time they spend on hopeless candidates. The best known algorithms generate candidates using the so-called prefix filter in conjunction with length- and position-based filters.

In this paper we show that the potential of length and position have only partially been leveraged. We propose a new filter, the *position-enhanced length filter*, which exploits the matching position to incrementally tighten the length filter; our filter identifies hopeless candidates and avoids processing them. The filter is very efficient, requires no change in the data structures of most prefix filter algorithms, and is particularly effective for foreign joins, i.e., joins between two different collections of sets.

## 1. INTRODUCTION

The *set similarity join* computes all pairs of similar sets from two collections of sets. The similarity is assessed using a set similarity function, e.g., set overlap, Jaccard, or Cosine similarity, together with a threshold. A pair of sets is in the join result if the similarity exceeds the threshold.

Set similarity joins have many interesting applications ranging from near duplicate detection of Web documents to community mining in social networks [9]. The set elements are called *tokens* [3] and are often used to represent complex objects, e.g., strings ($q$-grams [11]) or trees ($pq$-grams [2]).

The best algorithms for set similarity joins are based on an inverted list index and the so-called *prefix filter* [5]. The prefix filter operates on sorted sets and rejects candidate pairs that have no overlap in a (short) prefix. Only the prefix must be indexed, which leads to substantial savings in space and runtime. However, for frequent tokens, the prefix filter produces a large number of candidates. Recent developments in the field leverage the position of the matching tokens between two prefixes (positional filter) and the number of remaining tokens in the overall set (length filter) to further reduce the number of candidates.

This paper proposes a new filter, the *position-enhanced length filter* (PEL), which tightens the length filter based on the position of the current token match. In previous work, position and length information have only partially been exploited. PEL fully leverages position and length to achieve additional pruning power. As a key feature, PEL does not require changes in the prefix filter index, but is applied on top of previous algorithms at almost no cost. In our experiments we show that PEL is particularly effective for foreign joins. PEL also performs well for self joins over large collections of small sets.[1]

The remaining paper is organized as follows: Section 2 introduces the set similarity join, provides background material, and an in-depth analysis of filtering techniques based on position and length. Our novel position-enhanced length filter (PEL) is introduced in Section 3. We empirically evaluate our technique and demonstrate its effectiveness on real world data in Section 4. In Section 5 we survey related work and finally draw conclusions in Section 6.

## 2. BACKGROUND

We revisit candidate generation, candidate reduction, and efficient verification techniques discussed in literature. The main concepts of this section are summarized in Figure 1.

### 2.1 Candidate Generation

We shorty explain the prefix filter, translate normalized thresholds to overlap thresholds, revisit length- and position-based filter conditions, and discuss prefixes in index implementations of set similarity joins.

**Prefix Filter.** The fastest set similarity joins are based on the prefix filter principle [5]: A pair of sorted sets $s_0, s_1$ can only meet an overlap threshold $t_O$, i.e., $|s_0 \cap s_1| \geq t_O$, if there is a non-zero overlap in the prefixes of the sets. The prefixes are of length $|s_0| - t_O + 1$ for $s_0$ and $|s_1| - t_O + 1$ for $s_1$. The set similarity join proceeds in three steps: (a) index the prefixes of one join partner, (b) probe the prefixes of the other join partner against the index to generate candidates, (c) verify the candidates by inspecting the full sets.

**Threshold for normalized set overlap.** Normalized set similarity measures take the set sizes into account. For

In: G. Specht, H. Gamper, F. Klan (eds.): Proceedings of the 26th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 21.10.2014 - 24.10.2014, Bozen, Italy, published at http://ceur-ws.org.

[1]In a self join, both input relations are identical, which cannot be assumed in a foreign join.

**Table 1: Set similarity functions and related definitions, extending [7, Table 1] by new pmaxsize.**

| | Similarity function | $\mathbf{minoverlap}(t, s_0, s_1)$ | $\mathbf{minsize}(t, s_0)$ | $\mathbf{pmaxsize}(t, s_0, p_0)$ | $\mathbf{maxsize}(t, s_0)$ |
|---|---|---|---|---|---|
| Jaccard | $J(s_0, s_1) = \frac{\lvert s_0 \cap s_1\rvert}{\lvert s_0 \cup s_1\rvert}$ | $\frac{t}{1+t}(\lvert s_0\rvert + \lvert s_1\rvert)$ | $t\,\lvert s_0\rvert$ | $\frac{\lvert s_0\rvert-(1+t)\cdot p_0}{t}$ | $\frac{\lvert s_0\rvert}{t}$ |
| Cosine | $C(s_0, s_1) = \frac{\lvert s_0 \cap s_1\rvert}{\sqrt{\lvert s_0\rvert \cdot \lvert s_1\rvert}}$ | $t\sqrt{\lvert s_0\rvert \cdot \lvert s_1\rvert}$ | $t^2\,\lvert s_0\rvert$ | $\frac{(\lvert s_0\rvert-p_0)^2}{\lvert s_0\rvert \cdot t^2}$ | $\frac{\lvert s_0\rvert}{t^2}$ |
| Dice | $D(s_0, s_1) = \frac{2\cdot(\lvert s_0 \cap s_1\rvert)}{\lvert s_0\rvert+\lvert s_1\rvert}$ | $\frac{t(\lvert s_0\rvert+\lvert s_1\rvert)}{2}$ | $\frac{t\,\lvert s_0\rvert}{2-t}$ | $\frac{(2-t)\cdot\lvert s_0\rvert-2p_0}{t}$ | $\frac{(2-t)\lvert s_0\rvert}{t}$ |
| Overlap | $O(s_0, s_1) = \lvert s_0 \cap s_1\rvert$ | $t$ | $t$ | $\infty$ | $\infty$ |

the purpose of set similarity joins, the normalized threshold is translated into an overlap threshold (called **minoverlap**). The overlap threshold may be different for each pair of sets. For example, for the Cosine threshold $t_C$ (join predicate $|s_0 \cap s_1|/\sqrt{|s_0| \cdot |s_1|} \geq t_C$), $\mathbf{minoverlap}(t_C, s_0, s_1) := t_C \cdot \sqrt{|s_0| \cdot |s_1|}$. Table 1 lists the definitions of well-known set similarity functions with the respective overlap thresholds.

**Length filter.** For a given threshold $t$ and a reference set $s_0$, the size of the matching partners must be within the interval $[\mathbf{minsize}(t, s_0), \mathbf{maxsize}(t, s_0)]$ (see Table 1). This was first observed for the PartEnum algorithm [1] and was later called the *length filter*. Example: $|s_0| = 10$, $|s_1| = 6$, $|s_2| = 16$, Cosine threshold $t_C = 0.8$. Since $\mathbf{minoverlap}(t_C, s_0, s_1) = 6.1 > |s_1|$ we conclude (without inspecting any set element) that $s_0$ cannot reach threshold $t_C$ with $s_1$. Similarly, $\mathbf{minoverlap}(t_C, s_0, s_2) = 10.1$, thus $s_2$ is too large to meet the threshold with $s_0$. In fact, $\mathbf{minsize}(t_C, s_0) = 6.4$ and $\mathbf{maxsize}(t_C, s_0) = 15.6$.

**Prefix length.** The prefix length is $|s_0| - t_O + 1$ for a given overlap threshold $t_O$ and set $s_0$. For *normalized* thresholds $t$ the prefix length does not only depend on $s_0$, but also on the sets we compare to. If we compare to $s_1$, the minimum prefix size of $|s_0|$ is $\mathbf{minprefix}(t, s_0, s_1) = |s_0| - \mathbf{minoverlap}(t, s_0, s_1) + 1$. When we index one of the join partners, we do not know the size of the matching partners upfront and need to cover the worst case; this results in the prefix length $\mathbf{maxprefix}(t, s_0) = |s_0| - \mathbf{minsize}(t, s_0) + 1$ [7], which does not depend on $s_1$. For typical Jaccard thresholds $t \geq 0.8$, this reduces the number of tokens to be processed during the candidate generation phase by 80 % or more.

For self joins we can further reduce the prefix length [12] w.r.t. **maxprefix**: when the index is built on-the-fly in increasing order of the sets, then the indexed prefix of $s_0$ will never be compared to any set $s_1$ with $|s_1| < |s_0|$. This allows us to reduce the prefix length to $\mathbf{midprefix}(t, s_0) = |s_0| - \mathbf{minoverlap}(t, s_0, s_0) + 1$.

**Positional filter.** The minimum prefix length for a pair of sets is often smaller than the worst case length, which we use to build and probe the index. When we probe the index with a token from the prefix of $s_0$ and find a match in the prefix of set $s_1$, then the matching token may be outside the optimal prefix. If this is the first matching token between $s_0$ and $s_1$, we do not need to consider the pair. In general, a candidate pair $s_0, s_1$ must be considered only if

$$\mathbf{minoverlap}(t, s_0, s_1) \leq o + \min\{|s_0| - p_0, |s_1| - p_1\}, \quad (1)$$

where $o$ is the current overlap (i.e., number of matching tokens so far excluding the current match) and $p_0$ ($p_1$) is the position of the current match in the prefix of $s_0$ ($s_1$); positions start at 0.

Previous work:

- $\mathbf{minoverlap}(t, s_0, s_1)$: equivalent overlap threshold for Jaccard, Cosine, or Dice threshold $t$ for a pair of sets $s_0, s_1$.
- $\mathbf{minsize}(t, s_0)$, $\mathbf{maxsize}(t, s_0)$: minimum and maximum size of any set that can satisfy threshold $t$ w.r.t. set $s_0$.
- $\mathbf{maxprefix}(t, s_0) = |s_0| - \mathbf{minsize}(t, s_0) + 1$: length of probing prefix
- $\mathbf{midprefix}(t, s_0) = |s_0| - \mathbf{minoverlap}(t, s_0, s_0) + 1$: length of indexing prefix for self joins
- $\mathbf{minprefix}(t, s_0, s_1) = |s_0| - \mathbf{minoverlap}(t, s_0, s_1) + 1$: length of optimal prefix for a particular pair of sets

Position-enhanced length filter (PEL):

- $\mathbf{pmaxsize}(t, s_0, p_0)$: new tighter limit for maximum set size based on the probing set position

**Figure 1: Overview of functions.**

The positional filter is stricter than the prefix filter and is applied on top of it. The pruning power of the positional filter is larger for prefix matches further to right (i.e., when $p_0, p_1$ increase). Since the prefix filter may produce the same candidate pair multiple times (for each match in the prefix), an interesting situation arises: a pair that passes the positional filter for the first match may not pass the filter for later matches. Thus, the positional filter is applied to pairs that are already in the candidate set whenever a new match is found. To correctly apply the positional filter we need to maintain the overlap value for each pair in the candidate set. We illustrate the positional filter with examples.

*Example 1.* Set $s_0$ in Figure 2 is the probing set (prefix length $\mathbf{maxprefix} = 4$), $s_1$ is the indexed set (prefix length $\mathbf{midprefix} = 2$, assuming self join). Set $s_1$ is returned from the index due to the match on g (first match between $s_0$ and $s_1$). The required overlap is $\lceil \mathbf{minoverlap}_C(0.8, s_0, s_1) \rceil = 8$. Since there are only 6 tokens left in $s_1$ after the match, the maximum overlap we can get is 7, and the pair is pruned. This is also confirmed by the positional filter condition (1) ($o = 0$, $p_0 = 3$, $p_1 = 1$).

*Example 2.* Assume a situation similar to Figure 2, but the match on g is the *second* match (i.e., $o = 1$, $p_0 = 3$, $p_1 = 1$). Condition (1) holds and the pair can not be pruned, i.e., it remains in the candidate set.

*Example 3.* Consider Figure 3 with probing set $s_0$ and indexed set $s_1$. The match on token a adds pair $(s_0, s_1)$ to the candidate set. Condition (1) holds for the match on **a** ($o = 0$, $p_0 = 0$, $p_1 = 0$), and the pair is not pruned by the positional filter. For the next match (on e), however, condition (1) does not hold ($o = 1$, $p_0 = 1$, $p_1 = 4$) and the positional filter removes the pair from the candidate set. Thus, the positional filter does not only avoid pairs to enter

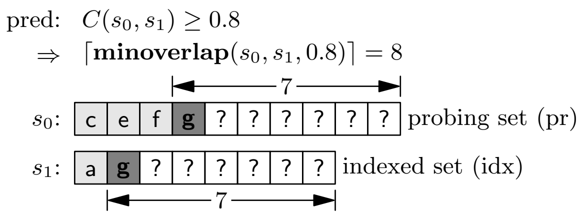

pred: $C(s_0, s_1) \geq 0.8$
$\Rightarrow \lceil \textbf{minoverlap}(s_0, s_1, 0.8) \rceil = 8$
7
$s_0$: c e f g ? ? ? ? ? ? probing set (pr)
$s_1$: a g ? ? ? ? ? ? indexed set (idx)
7

**Figure 2: Sets with matching token In prefix: match impossible due to positions of matching tokens and remaining tokens.**

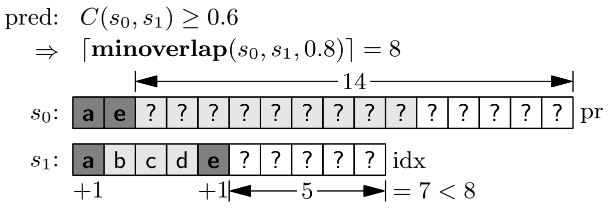

pred: $C(s_0, s_1) \geq 0.6$
$\Rightarrow \lceil \textbf{minoverlap}(s_0, s_1, 0.8) \rceil = 8$
14
$s_0$: a e ? ? ? ? ? ? ? ? ? ? ? ? ? ? pr
$s_1$: a b c d e ? ? ? ? ? idx
+1
+1
5
$= 7 < 8$

**Figure 3: Sets with two matching tokens: pruning of candidate pair by second match.**

the candidate set, but may remove them later.

## 2.2 Improving the Prefix Filter

The prefix filter often produces candidates that will be removed immediately in the next filter stage, the positional filter (see Example 1). Ideally, such candidates are not produced at all. This issue is addressed in the mpjoin algorithm [7] as outlined below.

Consider condition (1) for the positional filter. We split the condition into two new conditions by expanding the minimum such that the conjunction of the new conditions is equivalent to the positional filter condition:

$$\textbf{minoverlap}(t, s_0, s_1) \leq o + |s_0| - p_0 \tag{2}$$

$$\textbf{minoverlap}(t, s_0, s_1) \leq o + |s_1| - p_1 \tag{3}$$

The mpjoin algorithm leverages condition (2) as follows. The probing sets $s_0$ are processed in increasing size order, so $|s_0|$ grows monotonically during the execution of the algorithm. Hence, for a specific set $s_1$, **minoverlap** grows monotonically. We assume $o = 0$ (and justify this assumption later). For a given index entry $(s_1, p_1)$, the right side of condition (2) is constant, while the left side can only grow. After the condition fails to hold for the first time, it will never hold again, and the index list entry is removed. For a given index set $s_1$, this improvement changes the effective length of the prefix (i.e., the part of the sets where we may detect matches) w.r.t. a probing set $s_0$ to $\textbf{minprefix}(t, s_0, s_1) = |s_1| - \textbf{minoverlap}(t, s_0, s_1) + 1$, which is optimal. On the downside, a shorter prefix may require more work in the verification phase: in some cases, the verification can start after the prefix as will be discussed in Section 2.3.

## 2.3 Verification

Efficient verification techniques are crucial for fast set similarity joins. We revisit a baseline algorithm and two improvements, which affect the verification speed of both false and true positives. Unless explicitly mentioned, the term *prefix* subsequently refers to **maxprefix** (probing set) resp.

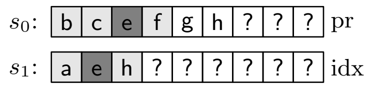


**Figure 4: Verification: where to start?**

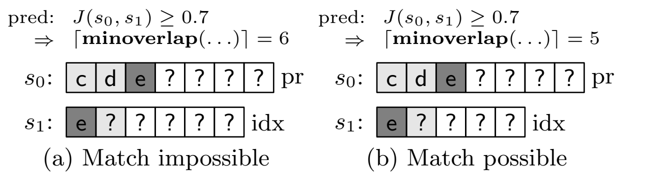

pred: $J(s_0, s_1) \geq 0.7$
$\Rightarrow \lceil \textbf{minoverlap}(\ldots) \rceil = 6$
$s_0$: c d e ? ? ? ? pr
$s_1$: e ? ? ? ? ? ? idx
(a) Match impossible
pred: $J(s_0, s_1) \geq 0.7$
$\Rightarrow \lceil \textbf{minoverlap}(\ldots) \rceil = 5$
$s_0$: c d e ? ? ? ? pr
$s_1$: e ? ? ? ? idx
(b) Match possible

**Figure 5: Impossible and possible set sizes based on position in $s_0$ and the size-dependent minoverlap.**

**midprefix** (indexing set) as discussed in earlier sections.

Since the sets are sorted, we compute the overlap in a merge fashion. At each merge step, we verify if the current overlap and the remaining set size are sufficient to achieve the threshold, i.e., we check positional filter condition (1).

*(A) Prefix overlap [12]*: At verification time we already know the overlap between the two prefixes of a candidate pair. This piece of information should be leveraged. Note that we cannot simply continue verification after the two prefixes. This is illustrated in Figure 4: there is 1 match in the prefixes of $s_0$ and $s_1$; when we start verification after the prefixes, we miss token h. Token h occurs after the prefix of $s_0$ but inside the prefix of $s_1$. Instead, we compare the last element of the prefixes: for the set with the smaller element ($s_0$), we start verification after the prefix (g). For the other set ($s_1$) we leverage the number of matches in the prefix (overlap $o$). Since the leftmost positions where these matches can appear are the first $o$ elements, we skip $o$ tokens and start at position $o$ (token e in $s_1$). There is no risk of double-counting tokens w.r.t. overlap $o$ since we start after the end of the prefix in $s_0$.

*(B) Position of last match [7]*: A further improvement is to store the position of the last match. Then we start the verification in set $s_1$ after this position (h in $s_1$, Figure 4).

**Small candidate set vs. fast verification.** The positional filter is applied on each candidate pair returned by the prefix filter. The same candidate pair may be returned multiple times for different matches in the prefix. The positional filter potentially removes existing candidate pairs when they appear again (cf. Section 2.1). This reduces the size of the candidate set, but comes at the cost of (a) lookups in the candidate set, (b) deletions from the candidate set, and (c) book keeping of the overlaps for each candidate pair. Overall, it might be more efficient to batch-verify a larger candidate set than to incrementally maintain the candidates; Ribeiro and Härder [7] empirically analyze this trade-off.

## 3. POSITION-ENHANCED LENGTH FILTERING

In this section, we motivate the *position-enhanced length filter* (PEL), derive the new filter function **pmaxsize**, discuss the effect of PEL on self vs. foreign joins, and show how to apply PEL to previous algorithms.

**Motivation.** The introduction of the position-enhanced length filter is inspired by examples for positional filtering

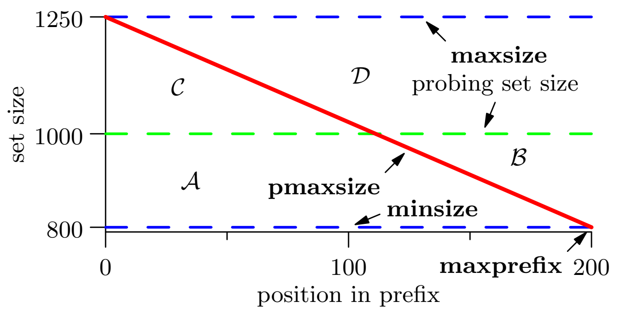


**Figure 6: Illustrating possible set sizes.**

like Figure 5(a). In set $s_1$, the only match in the prefix occurs at the leftmost position. Despite this being the leftmost match in $s_1$, the positional filter removes $s_1$: the overlap threshold cannot be reached due the position of the match in $s_0$. Apparently, the position of the token in the probing set can render a match of the index sets impossible, independently of the matching position in the index set. Let us analyze how we need to modify the example such that it passes the positional filter: the solution is to shorten index set $s_1$, as shown in Figure 5(b). This suggests that some tighter limit on the set size can be derived from the position of the matching token.

**Deriving the PEL filter.** For the example in Figure 5(a) the first part of the positional filter, i.e., condition (2), does not hold. We solve the equation **minoverlap**$(t, s_0, s_1) \leq |s_0| - p_0$ to $|s_1|$ by replacing **minoverlap** with its definition for the different similarity functions. The result is **pmaxsize**$(t, s_0, p_0)$, an upper bound on the size of eligible sets in the index. This bound is at the core of the PEL filter, and definitions of **pmaxsize** for various similarity measures are listed in Table 1.

**Application of PEL.** We integrate the **pmaxsize** upper bound into the prefix filter. The basic prefix filter algorithm processes a probing set as follows: loop over the tokens of the probing set from position $p_0 = 0$ to **maxprefix**$(t, s_0) - 1$ and probe each token against the index. The index returns a list of sets (their IDs) which contain this token. The sets in these lists are ordered by increasing size, so we stop processing a list when we hit a set that is larger than **pmaxsize**$(t, s_0, p_0)$.

Intuitively, we move half of the positional filter to the prefix filter, where we can evaluate it at lower cost: (a) the value of **pmaxsize** needs to be computed only once for each probing token; (b) we check **pmaxsize** against the size of each index list entry, which is a simple integer comparison. Overall, this is much cheaper than the candidate lookup that the positional filter must do for each index match.

**Self Joins vs. Foreign Joins.** The PEL filter is more powerful on foreign joins than on self joins. In self joins, the size of the probing set is an upper bound for the set size in the index. For all the similarity functions in Table 1, **pmaxsize** is below the probing set size in less than 50% of the prefix positions. Figure 6 gives an example: The probing set size is 1000, the Jaccard threshold is 0.8, so **minsize**$(0.8, 1000) = 800$, **maxsize**$(0.8, 1000) = 1250$, and the prefix size is 201. The x-axis represents the position in the prefix, the y-axis represents bounds for the set size of the other set. The region between **minsize** and **maxsize** is the base region. The base region is partitioned into four regions ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$) by the probing set size and **pmaxsize**. For foreign joins, our filter reduces the base region to $\mathcal{A}+\mathcal{C}$. If we assume that all set sizes occur equally likely in the individual inverted lists of the index, our filter cuts the number of index list entries that must be processed by 50%. Since the tokens are typically ordered by their frequency, the list length will increase with increasing matching position. Thus the gain of PEL in practical settings can be expected to be even higher. This analysis holds for all parameters of Jaccard and Dice. For Cosine, the situation is more tricky since **pmaxsize** is quadratic and describes a parabola. Again, this is in our favor since the parabola is open to the top, and the curve that splits the base region is below the diagonal.

For self joins, the only relevant regions are $\mathcal{A}$ and $\mathcal{B}$ since the size of the sets is bounded by the probing set size. Our filter reduces the relevant region from $\mathcal{A} + \mathcal{B}$ to $\mathcal{A}$. As Figure 6 illustrates, this reduction is smaller than the reduction for foreign joins. For the similarity functions in Table 1, $\mathcal{B}$ is always less than a quarter of the full region $\mathcal{A} + \mathcal{B}$. In the example, region $\mathcal{B}$ covers about 0.22 of $\mathcal{A} + \mathcal{B}$.

**Algorithm 1:** AllPairs-PEL$(S_p, I, t)$

```
Version using pmaxsize for foreign join;
input  : $S_p$ collection of outer sets, $I$ inverted list index
         covering maxprefix of inner sets, $t$ similarity
         threshold
output: $res$ set of result pairs (similarity at least $t$)
1  foreach $s_0$ in $S_p$ do
2      $M = \{\}$; /* Hashmap: candidate set → count */
3      for $p_0 \leftarrow 0$ to maxprefix$(t, s_0) - 1$ do
4          for $s_1$ in $I_{s_0[p]}$ do
5              if $|s_1| <$ minsize$(t, s_0)$ then
6                  remove index entry with $s_1$ from $I_{s_0[p]}$;
7              else if $|s_1| >$ pmaxsize$(t, s_0, p_0)$ then
8                  break;
9              else
10                 if $M[s_1] = \emptyset$ then
11                     $M = M \cup (s_1, 0)$;
12                 $M[s_1] = M[s_1] + 1$;
13         end
14     end
15     /* Verify() verifies the candidates in $M$ */
16     $res = res \cup Verify(s_0, M, t)$;
17 end
```

**Algorithm.** Algorithm 1 shows AllPairs-PEL[2], a version of AllPairs enhanced with our PEL filter. AllPairs-PEL is designed for foreign joins, i.e., the index is constructed in a preprocessing step before the join is executed. The only difference w.r.t. AllPairs is that AllPairs-PEL uses **pmaxsize**$(t, s_0, p_0)$ instead of **maxsize**$(t, s_0)$ in the condition on line 7. The extensions of the algorithms ppjoin and mpjoin with PEL are similar.

An enhancement that is limited to ppjoin and mpjoin is to simplify the positional filter: PEL ensures that no candidate set can fail on the first condition (Equation 2) of the split positional filter. Therefore, we remove the first part of the

[2]We use the *-PEL* suffix for algorithm variants that make use of our PEL filter.

**Table 2: Input set characteristics.**

| | #sets in collection | set size min | set size max | set size avg | # of diff. tokens |
|---|---|---|---|---|---|
| DBLP | $3.9 \cdot 10^6$ | 2 | 283 | 12 | $1.34 \cdot 10^6$ |
| TREC | $3.5 \cdot 10^5$ | 2 | 628 | 134 | $3.4 \cdot 10^5$ |
| ENRON | $5 \cdot 10^5$ | 1 | 192 000 | 298 | $7.3 \cdot 10^6$ |

minimum in the original positional filter (Equation 1), such that the minimum is no longer needed.

Note that the removal of index entries on line 6 is the easiest solution to apply **minsize**, but in real-world scenarios, it only makes sense for a single join to be executed. For a similarity search scenario, we recommend to apply binary search on the lists. For multiple joins with the same indexed sets in a row, we suggest to use an overlay over the index that stores the pointer for each list where to start.

## 4. EXPERIMENTS

We compare the algorithms AllPairs [4] and mpjoin [7] with and without our PEL extension on both self and foreign joins. Our implementation works on integers, which we order by the frequency of appearance in the collection. The time to generate integers from tokens is not measured in our experiments since it is the same for all algorithms. We also do not consider the indexing time for foreign joins, which is considered a preprocessing step. The use of PEL has no impact on the index construction. The prefix sizes are **maxprefix** for foreign joins and **midprefix** for self joins. For self joins, we include the indexing time in the overall runtime since the index is built incrementally on-the-fly. We report results for Jaccard and Cosine similarity, the results for Dice show similar behavior. Our experiments are executed on the following real-world data sets:

- DBLP[3]: Snapshot (February 2014) of the DBLP bibliographic database. We concatenate authors and title of each entry and generate tokens by splitting on whitespace.
- TREC[4]: References from the MEDLINE database, years 1987–1991. We concatenate author, title, and abstract, remove punctuation, and split on whitespace.
- ENRON[5]: Real e-mail messages published by FERC after the ENRON bankruptcy. We concatenate subject and body fields, remove punctuation, and split on whitespace.

Table 2 lists basic characteristics of the input sets. We conduct our experiments on an Intel Xeon 2.60GHz machine with 128 GB RAM running Debian 7.6 'wheezy'. We compile our code with gcc -O3. Claims about results on "all" thresholds for a particular data set refer to the thresholds $\{0.5, 0.6, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$. We stop tests whose runtime exceeds one hour.

**Foreign Joins.** For foreign joins, we join a collection of sets with a copy of itself, but do not leverage the fact that the collections are identical. Figures 7(a) and 7(b) show the performance on DBLP with Jaccard similarity threshold 0.75 and Cosine similarity 0.85. These thresholds produce result sets of similar size. We observe a speedup of factor 3.5 for AllPairs-PEL over AllPairs with Jaccard, and a speedup of 3.8 with Cosine. For mpjoin to mpjoin-PEL we observe a speedup of 4.0 with Jaccard and 4.2 with Cosine. Thus, the PEL filter provides a substantial speed advantage on these data points. For other Jaccard thresholds and mpjoin vs. mpjoin-PEL, the maximum speedup is 4.1 and the minimum speedup is 1.02. For threshold 0.5, only mpjoin-PEL finishes within the time limit of one hour. Among all Cosine thresholds and mpjoin vs. mpjoin-PEL, the maximum speedup is 4.2 ($t_C = 0.85$), the minimum speedup is 1.14 ($t_C = 0.95$). We only consider Cosine thresholds $t_C \geq 0.75$, because the non-PEL variants exceed the time limit for smaller thresholds. There is no data point where PEL slows down an algorithm. It is also worth noting that AllPairs-PEL beats mpjoin by a factor of 2.7 with Jaccard threshold $t_J = 0.75$ and 3.3 on Cosine threshold $t_C = 0.85$; we observe such speedups also on other thresholds.

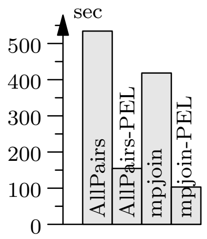
Figure 7(c) shows the performance on TREC with Jaccard threshold $t_J = 0.75$. The speedup for AllPairs-PEL compared to AllPairs is 1.64, and for mpjoin-PEL compared to mpjoin 2.3. The minimum speedup of mpjoin over all thresholds is 1.26 ($t_J = 0.95$), the maximum speedup is 2.3 ($t_J = 0.75$). Performance gains on ENRON are slightly smaller – we observe speedups of 1.15 (AllPairs-PEL over AllPairs), and 1.85 (mpjoin-PEL over mpjoin) on Jaccard threshold $t_J = 0.75$ as illustrated in Figure 7(d). The minimum speedup of mpjoin over mpjoin-PEL is 1.24 ($t_J = 0.9$ and 0.95), the maximum speedup is 2.0 ($t_J = 0.6$).

Figure 8(a) shows the number of processed index entries (i.e., the overall length of the inverted lists that must be scanned) for Jaccard threshold $t_J = 0.75$ on TREC. The number of index entries increases by a factor of 1.67 for AllPairs w.r.t. AllPairs-PEL, and a factor of 4.0 for mpjoin w.r.t. mpjoin-PEL.

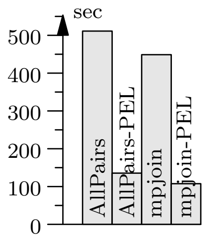
Figure 8(b) shows the number of candidates that must be verified for Jaccard threshold $t_J = 0.75$ on TREC. On AllPairs, PEL decreases the number of candidates. This is because AllPairs does not apply any further filters before verification. On mpjoin, the number of candidates increases by 20%. This is due to the smaller number of matches from the prefix index in the case of PEL: later matches can remove pairs from the candidate set (using the positional filter) and thus decrease its size. However, the larger candidate set for PEL does not seriously impact the overall performance: the positional filter is also applied in the verification phase, where the extra candidate pairs are pruned immediately.

**Self joins.** Due to space constraints, we only show results for DBLP and ENRON, i.e., the input sets with the smallest and the largest average set sizes, respectively. Figure 7(e) and 7(f) show the performance of the algorithms on DBLP and ENRON with Jaccard threshold $t_J = 0.75$. Our PEL filter provides a speed up of about 1.22 for AllPairs, and 1.17 for mpjoin on DBLP. The maximum speedup we observe is 1.70 (AllPairs-PEL vs. AllPairs, $t_J = 0.6$); for $t_J = 0.95$ there is no speed difference between mpjoin and mpjoin-PEL. On the large sets of ENRON, the performance is worse for AllPairs-PEL because verification takes more time than PEL can save in the probing phase (by reducing the number of processed index entries). There is almost no

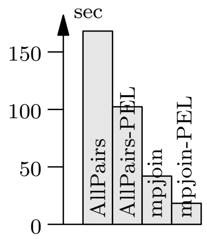
[3]`http://www.informatik.uni-trier.de/~Ley/db/`
[4]`http://trec.nist.gov/data/t9_filtering.html`
[5]`https://www.cs.cmu.edu/~enron/`



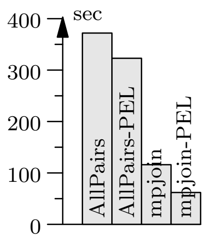
(a) Foreign join, DBLP, $t_J = 0.75$.



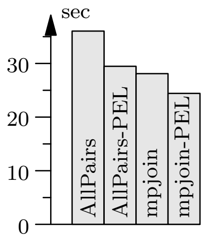
(b) Foreign join, DBLP, $t_C = 0.85$.



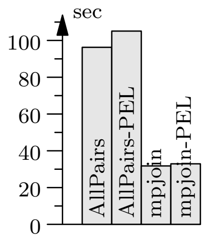
(c) Foreign join, TREC, $t_J = 0.75$



(d) Foreign j., ENRON, $t_J = 0.75$

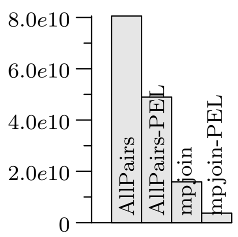


(e) Self join, DBLP, $t_J = 0.75$

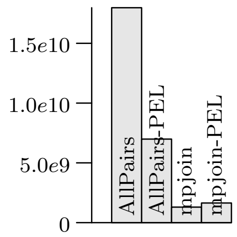


(f) Self join, ENRON, $t_J = 0.75$

**Figure 7: Join times.**



(a) Number of processed index entries.



(b) Number of candidates to be verify.

**Figure 8: TREC (foreign join):** $t_J = 0.75$

difference between mpjoin and mpjoin-PEL. The maximum increase in speed is 9% (threshold 0.8, mpjoin), the maximum slowdown is 30% (threshold 0.6, AllPairs).

Summarizing, PEL substantially improves the runtime in foreign join scenarios. For self joins, PEL is less effective and, in some cases, may even slightly increase the runtime.

## 5. RELATED WORK

Sarawagi and Kirpal [8] first discuss efficient algorithms for exact set similarity joins. Chaudhuri et al. [5] propose SSJoin as an in-database operator for set similarity joins and introduce the prefix filter. AllPairs [4] uses the prefix filter with an inverted list index. The ppjoin algorithm [12] extends AllPairs by the positional filter and introduces the suffix filter, which reduces the candidate set before the final verification. The mpjoin algorithm [7] improves over ppjoin by reducing the number of entries returned from the index. AdaptJoin [10] takes the opposite approach and drastically reduces the number of candidates at the expense of longer prefixes. Gionis et al. [6] propose an approximate algorithm based on LSH for set similarity joins. Recently, an SQL operator for the token generation problem was introduced [3].

## 6. CONCLUSIONS

We presented PEL, a new filter based on the **pmaxsize** upper bound derived in this paper. PEL can be easily plugged into algorithms that store prefixes in an inverted list index (e.g., AllPairs, ppjoin, or mpjoin). For these algorithms, PEL will effectively reduce the number of list entries that must be processed. This reduces the overall lookup time in the inverted list index at the cost of a potentially larger candidate set. We analyzed this trade-off for foreign joins and self joins. Our empirical evaluation demonstrated that the PEL filter improves performance in almost any foreign join and also in some self join scenarios, despite the fact that it may increase the number of candidates to be verified.

## 7. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. VLDB*, pages 918 – 929, 2006.

[2] N. Augsten, M. H. Böhlen, and J. Gamper. The *pq*-gram distance between ordered labeled trees. *ACM TODS*, 35(1), 2010.

[3] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. In *Proc. SIGMOD*, pages 1495 – 1506. ACM, 2014.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. *WWW*, 7:131 – 140, 2007.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. ICDE*, page 5. IEEE, 2006.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. VLDB*, pages 518–529, 1999.

[7] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62 – 78, 2011.

[8] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. SIGMOD*, pages 743 – 754. ACM, 2004.

[9] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: A large-scale study in the orkut social network. In *Proc. SIGKDD*, pages 678 – 684. ACM, 2005.

[10] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In *Proc. SIGMOD*, pages 85 – 96. ACM, 2012.

[11] C. Xiao, W. Wang, and X. Lin. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. In *Proc. VLDB*, 2008.

[12] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM TODS*, 36(3):15, 2011.