An Empirical Evaluation of Set Similarity Join Techniques

Willi Mann
University of Salzburg
Salzburg, Austria
wmann@cosy.sbg.ac.at

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Panagiotis Bouros
Aarhus University
Aarhus, Denmark
pbour@cs.au.dk

ABSTRACT

Set similarity joins compute all pairs of similar sets from two collections of sets. We conduct extensive experiments on seven state-of-the-art algorithms for set similarity joins. These algorithms adopt a filter-verification approach. Our analysis shows that verification has not received enough attention in previous works. In practice, efficient verification inspects only a small, constant number of set elements and is faster than some of the more sophisticated filter techniques. Although we can identify three winners, we find that most algorithms show very similar performance. The key technique is the prefix filter, and AllPairs, the first algorithm adopting this techniques is still a relevant competitor. We repeat experiments from previous work and discuss diverging results. All our claims are supported by a detailed analysis of the factors that determine the overall runtime.

1. INTRODUCTION

The *set similarity join* computes all pairs of similar sets from two collections of sets. Two sets are similar if their overlap exceeds some user-defined threshold. The efficient computation of set similarity joins has received much attention from both academia [4, 15, 19, 21] and industry [1, 3, 5], and a number of techniques have been developed.

The goal of this paper is to experimentally test and compare the fastest algorithms for set similarity joins. We focus on main memory algorithms and include AllPairs [3], PPJoin and PPJoin+ [21], MPJoin [15], MPJoin-PEL [9], AdaptJoin [19], and GroupJoin [4] into our analysis. We test on two synthetic and ten different real world datasets from various domains. We implemented all algorithms[1] (C++) and tested them against the available original implementations: our implementation is faster on almost all data points (i.e., measurements on a combination of dataset, algorithm, and join threshold). This paper is self-contained; in addition, detailed figures about all results are available from [10].

All tested algorithms implement a filter-verification framework. The core of the filter step is (some variant of) the so-called *prefix filter* [5], which generates a preliminary candidate set. The preliminary candidates undergo various tests that further reduce the candidate set. The resulting candidate pairs are verified in a merge-like algorithm that scans the sorted sets and produces the join result.

Results. Our extensive study provides new, interesting insights and sheds light on previous experimental results. In a nutshell, we arrive to the following conclusions:

Three algorithms on the skyline. We measure the gap to the winner on all data points in our test: PPJoin and GroupJoin show the best median and average behavior; GroupJoin is the most robust algorithm (smallest gap in the worst case); AllPairs wins on the largest number of data points.

Small performance differences. With two exceptions (AdaptJoin, PPJoin+), all algorithms show similar performance: on average, loser and winner are within 35% from each other; the loser is at most 4 times slower.

Verification is surprisingly fast. Although in general verification is linear in the set size, in our experiments the number of required comparisons in the merge-like verification routine is a small constant (often 2 or less, 18 at most) that is independent of the set size.

Sophisticated filters are too slow. Due to efficient verification, expensive filters do not pay off: we measure the slowest runtimes for AdaptJoin and PPJoin+, which apply the most sophisticated filters and produce the smallest candidate sets.

We repeat key experiments from four previous works [8, 15, 19, 21] and analyze the deviation from our result. We demonstrate that inefficient verification favors expensive filter techniques, which explains the diverging results.

Related Work. Jiang et al. [8] evaluate string similarity joins and dedicate a section to set similarity. Given the context, only string data is considered. Our analysis also includes photo meta-data, click-streams, query logs, point of sale, social media/network, and user preference data, which cover a wider range of dataset characteristics. Jiang et al. [8] do not evaluate GroupJoin, MPJoin, and MPJoin-PEL, which turn out to be relevant competitors. Further, our analysis leads to different conclusions. We repeat an experiment of [8] and discuss diverging results.

The original works that introduced the join techniques in our test [3, 4, 9, 15, 19, 21] also include empirical evaluations. Compared to our study, the scope of these experiments is limited in terms of test data, competitors, and analysis of the runtime results. We re-implement all algorithms and repeat selected experiments.

We focus on main memory techniques and do not discuss distributed set similarity algorithms, e.g., [6, 11, 18]. We further do not include older works that pre-date the prefix filter (e.g., PartEnum [1], MergeOpt [16]) or approximation techniques (e.g., LSH [7], BayesLSH [17]).

[1] Source code available: http://ssjoin.dbresearch.uni-salzburg.at/

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 9
Copyright 2016 VLDB Endowment 2150-8097/16/05.

To the best of our knowledge, this is the first empirical study that (a) includes all key players in the field, (b) covers datasets from a wide range of different applications, and (c) provides an in-depth analysis of the runtime results.

Outline. In Sections 2–4 we revisit set similarity joins, discuss relevant implementation choices, and define the experimental setup. We discuss runtime behavior in Section 5, main memory usage in Section 6, and previous experimental results in Section 7.

2. BACKGROUND

We revisit the formal definition of set similarity joins and the prefix filter, present the algorithmic framework used by all tested algorithms, and shortly introduce the individual techniques evaluated in this paper.

2.1 Set Similarity Join and Prefix Filter

Set Similarity Join. The *set similarity join* computes all pairs of similar sets from two collections of sets. The set elements are called *tokens* [2]. Two sets are similar if their token overlap exceeds a user-defined threshold. To account for the size difference between sets, the overlap is often normalized, and the threshold is given as Jaccard, Dice, or Cosine similarity [2] (cf. Table 1). Formally, given two collections, R and S, a set similarity function $\text{Sim}(r, s)$ between two sets, and a similarity threshold t, the set similarity join is defined as $R \tilde{\bowtie} S = \{(r, s) \in R \times S \mid \text{Sim}(r, s) \geq t\}$.

Prefix Filter. A key technique for efficient set similarity joins is the so-called *prefix filter* [5], which operates on pairs of sets, (r, s), and inspects only small subsets of r and s to prune the pair. The inspected subsets are called *prefixes*. The π-prefix of a set is formed by the π very first elements of the set in a given total order. With appropriate prefix sizes, two sets can be pruned if their prefixes have no common element. The prefix size depends on the similarity threshold and the similarity function. For example, the prefix filter for the overlap similarity is defined as follows: Given two sets, r and s, and an overlap threshold t; if $|r \cap s| \geq t$, then there is at least one common token within the π_r-prefix of r and the π_s-prefix of s, where $\pi_r = |r| - t + 1$ and $\pi_s = |s| - t + 1$.

Consider, for example, the sorted sets r and s in Figure 1. The respective prefixes for overlap threshold $t = 4$ are shaded. Since there is no token match in the prefix, the pair (r, s) will not satisfy the required threshold and can safely be pruned. The intuition is that the remaining three tokens (question marks) can contribute at most 3 matches to the overlap, which is not enough.

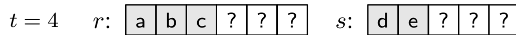


Figure 1: Prefix filter.

Length Filter. Normalized similarity functions (Jaccard, Cosine, and Dice) depend on the set size and offer additional pruning opportunities. We define the length filter [1], which is used by most algorithms in conjunction with the prefix filter. We limit our discussion to Jaccard similarity and refer to Table 1 for other normalization techniques; a detailed discussion can be found in [9]. Length filter: Set r can reach Jaccard threshold t_J only with a set s of size $lb_r \leq |s| \leq ub_r$ ($lb_r = t_J \cdot |r|$, $ub_r = |r|/t_J$, cf. Table 1); for example, if $|r| = 10$ and $t_J = 0.8$, then $8 \leq |s| \leq 12$ is required.

2.2 Algorithmic Framework

All tested algorithms follow a similar filter-verification approach, which is illustrated in Figure 2. The similarity join is executed as an index nested loop join: (1) an index lookup returns a set of pre-candidates; (2) the pre-candidates are deduplicated and filtered; (3) the resulting candidate pairs undergo a verification phase to generate the final result. We discuss the layout of the pre-processed input data, the structure of the index, and each of the three join steps in detail.

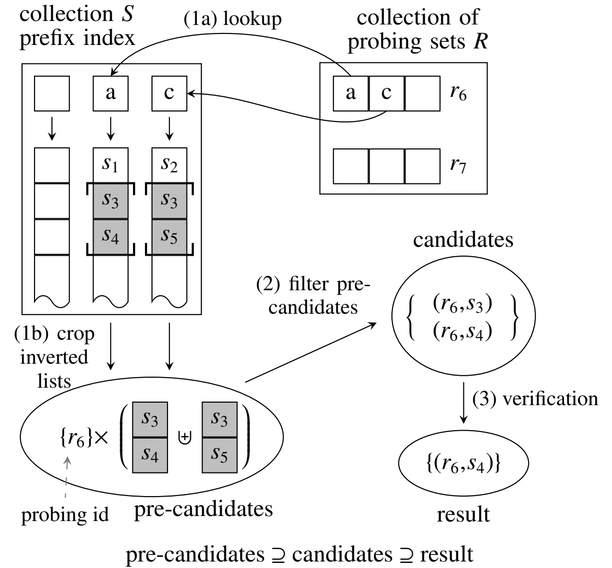


Figure 2: Set similarity join with prefix index.

Input Data and Prefix Index. The input consists of two collections of sets, R and S. Following previous work [8, 21], the tokens are sorted by their frequency in the collections such that the prefixes are formed by infrequent tokens. The tokens in each set are unique.

An inverted index is built on top of collection S, considering only the prefix of each set. For each token in S, the inverted list stores all sets in S that contain that token in the prefix. The set entries in the inverted list are sorted by increasing set size.

Outline of Join Algorithm. The probing sets, $r \in R$, are processed in increasing size order, and for each probing set the following steps are performed (cf. Figure 2):

(1) Pre-candidate generation. For each token τ in the prefix of r, the lookup (1a) returns an inverted list. The inverted lists are cropped (1b) using the length filter. Each entry s_i in the cropped list forms a pre-candidate (r, s_i) with the current probing set r. Pre-candidates may contain duplicates (e.g., (r_6, s_3) in Figure 2).

(2) Candidate generation. The pre-candidates are deduplicated and filtered. Pre-candidates that pass all filters are *candidates*. Some algorithms remove obsolete entries from the inverted lists during candidate generation, thus modifying the inverted list index.

(3) Verification. False positives are removed from the candidate set to form the join result. The verification of a candidate pair (r, s) accesses the sorted sets in the input collections and scans r and s in a merge-like fashion. Information from previous filter steps is used to avoid redundant checks. For reference, we show the verification routine in Algorithm 1.

Overlap and Prefix Size for Normalized Similarity Functions. The join algorithms do not directly work with normalized similarity functions, but translate the normalized threshold into an *equivalent overlap*. The equivalent overlap depends on the set size and may vary for each pair of sets (cf. Table 1, eqoverlap). The prefix size is $\pi_r = |r| - \lceil \text{eqoverlap}(r, s) \rceil + 1$. Since the size of s is not known

Table 1: Similarity functions and set size bounds for set r and s [9].

Sim. Func.	Definition	eqoverlap	lb_r	ub_r	ub_{PEL}
Jaccard	$\frac{\|r \cap s\|}{\|r \cup s\|}$	$\frac{t_J}{1+t_J}(\|r\| + \|s\|)$	$t_J \cdot \|r\|$	$\frac{\|r\|}{t_J}$	$\frac{\|r\|-(1+t_J)\cdot p_r}{t_J}$
Cosine	$\frac{\|r \cap s\|}{\sqrt{\|r\| \cdot \|s\|}}$	$t_C \sqrt{\|r\| \cdot \|s\|}$	$t_C^2 \|r\|$	$\frac{\|r\|}{t_C^2}$	$\frac{(\|r\|-p_r)^2}{\|r\| \cdot t_C^2}$
Dice	$\frac{2 \cdot \|r \cap s\|}{\|r\|+\|s\|}$	$\frac{t_D(\|r\|+\|s\|)}{2}$	$\frac{t_D \|r\|}{2-t_D}$	$\frac{(2-t_D)\|r\|}{t_D}$	$\frac{(2-t_D)\cdot\|r\|-2p_r}{t_D}$
Overlap	$\|r \cap s\|$	t_O	t_O	∞	∞

eqoverlap: equivalent overlap for normalized thresholds t_J (Jaccard), t_C (Cosine), t_D (Dice)
lb_r: size lower bound on join partners for r
ub_r: size upper bound on join partners for r
ub_{PEL}: size upper bound on join partners for r at matching position p_r (MPJoin-PEL)

when π_r is computed, $|s| = lb_r$ is assumed to get an upper bound for the prefix size. With $|s| = lb_r$ we get eqoverlap$(r, s) = lb_r$, thus the prefix size is $\pi_r = |r| - \lceil lb_r \rceil + 1$.

Self Join. In a self-join, where $R = S$, the result is symmetric, i.e., $(r, s) \in R \tilde{\bowtie} S \Rightarrow (s, r) \in R \tilde{\bowtie} S$. The self-join algorithms discussed in this paper report only one of the symmetric result pairs, and trivial pairs (r, r), are omitted. The symmetry of the result pairs is leveraged to incrementally build the index on the fly during the join. A set $r \in R$ is indexed after probing it against the existing index, which initially is empty. Since the sets in R are processed in increasing size order, no set in the index is longer than the current probing set. This allows us to use shorter prefixes of size $|r| - \lceil \text{eqoverlap}(r, r) \rceil + 1$ for indexing [21].

2.3 Algorithms

We introduce the set similarity join algorithms evaluated in this paper. Each algorithm is identified by a short acronym that we use throughout the paper. The algorithms mainly differ by the filters applied during pre-candidate and candidate generation (cf. Table 2).

Table 2: Algorithms with their filters.

Algorithm	Pre-Cand. Generation (Lookup and Crop)	Cand. Generation (Filter Pre-candidates)
ALL	prefix + length	
PPJ	prefix + length	positional
PP+	prefix + length	positional + suffix
MPJ	prefix + length	positional + removal
PEL	prefix + length + PEL	positional + removal
ADP	adapt + length	
GRP	group + prefix + length	positional

AllPairs (ALL). ALL [3] was the first main memory algorithm to use the prefix filter. For normalized thresholds, the length filter is applied to crop the inverted lists (lb_r and ub_r in Table 1).

PPJoin (PPJ). PPJ [20, 21] extends ALL with the positional filter on pre-candidates. The positional filter reasons over the matching position of tokens in the prefix. For example, the pair (r, s) in Figure 3 (prefix shaded) is a pre-candidate since it passes prefix and length filter. The Jaccard threshold ($t_J = 0.8$) is translated into an equivalent overlap ($t = 9$, cf. Table 1) for the given set pair. The first match in s is on position 1 (zero-based numbering), thus only 8 tokens of s are left to match a token in r, and the pre-candidate is rejected.

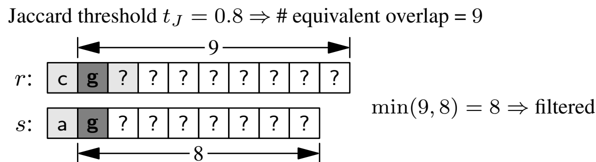


Figure 3: Positional filter.

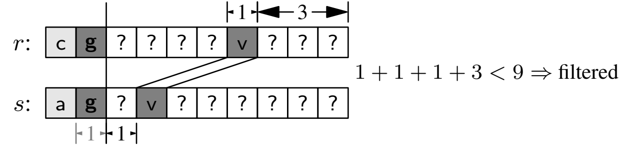


Figure 4: Suffix filter.

PPJoin+ (PP+). PP+ [20, 21] extends PPJ with the suffix filter, which recursively partitions pre-candidate pairs to tighten the upper bound on the token overlap. Consider Figure 4: The tokens to the right of the prefix match g are called *suffix*. The suffix of r is partitioned into two parts of similar size, the token at the partition border (v) is the *pivot*. A binary search in the suffix of s identifies the position of the pivot and partitions s. Tokens in the left partition of r can only match tokens in the left partition of s, thus there is at most 1 match in this partition. Similarly, at most 3 tokens match in the right partition. Overall, at most 6 tokens match between r and s, and the pre-candidate pair is rejected. The suffix filter is recursively applied to the left and right partition until the pair is filtered or a user-defined maximum recursion depth is reached.

GroupJoin (GRP). GRP [4] extends PPJ and leverages the fact that different sets may have identical prefixes. Sets with identical prefixes are grouped and treated like a single set during candidate generation (group), which allows for pruning candidates in large batches. The grouped candidate pairs are expanded (un-grouped) during verification.

MPJoin (MPJ). MPJ [15] extends PPJ by removing obsolete entries from the inverted list (removal filter). An entry is obsolete if it will be filtered by all future applications of the positional filter. MPJ leverages the fact that the probing sets are processed in increasing size order such that the equivalent overlap increases monotonically. Consider, for example, set s in Figure 3. The match on prefix token g triggers the positional filter for the probing set of size $|r| = 10$. Since $|r|$ cannot decrease (Section 2.2), any future match on g in s will also trigger the positional filter. Thus s can be safely removed from the inverted list of token g.

MPJoin-PEL (PEL). PEL [9] extends MPJ with the *position-enhanced length filter* (PEL), which establishes a tighter upper bound $ub_{\text{PEL}} \leq ub_r$ on the set size than the length filter. Consider Figure 5, where token m in set r is probed against the index. The inverted list, cropped with the length filter ($|s_i| \leq ub_r = 12.5$), contains three sets. Given the position of m in r ($p_r = 2$), at most 8 matches are possible with any set s_i. For sets s_2 and s_3, however, an equivalent overlap of $t = 9$ is required. PEL crops the list after s_1 ($|s_i| \leq ub_{\text{PEL}} = 8$) and thus reduces the pre-candidate set.

AdaptJoin (ADP). ADP [19] generalizes the prefix filter: a pair (r, s) is pruned if there are less than e token matches in the $(\pi_r + e - 1)$-prefix of r and the $(\pi_s + e - 1)$-prefix of s (cf. Section 2.1). For the standard prefix filter, $e = 1$. ADP computes e per probing set using a cost function. The prefix index is replaced by the adaptive prefix index (adapt), which supports longer prefixes dynamically.

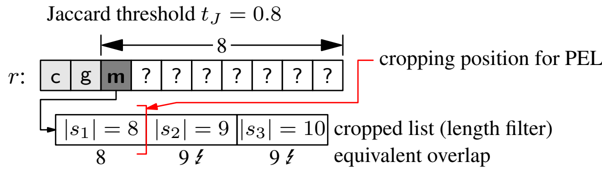


Figure 5: PEL: position-enhanced length filter.

3. IMPLEMENTATION NOTES

We have re-implemented all algorithms in C++ following the descriptions in the original papers. None of the algorithms required specialized data structures, and we used the C++ STL `vector` class to implement the input sets, the prefix index, and the candidate sets. Some implementation choices have a relevant impact on the runtime. We have implemented and tested different options and run all experiments with the fastest version of each algorithm. Our implementation choices are detailed in this section.

For some algorithms, the source code or binaries are available. Xiao et al. [21] (PPJ/PP+) provide the source code of ALL, PPJ, and PP+ (`src-xiao`[2]); Wang et al. [19] (ADP) provide the binary of ADP (`bin-wang`[3]); Jiang et al. [8] (experimental evaluation of string similarity joins) provide the binaries of ALL, PPJ, PP+, and ADP (`bin-jiang`[4]). Our implementation is faster than the available implementations on all tested data points. An exception is `bin-jiang`, which is faster for ADP on some selected data points (mostly for very small thresholds, $t_J \in \{0.5, 0.6\}$). However, the runtime difference has no impact on the overall winner on these data points. In Section 7, we repeat previous experiments with the original implementations provided by the authors.

Input Data and Prefix Index. All tokens are numbered without gaps and are represented by their integer IDs. The token numbering is based on token frequencies: the lower the frequency of a token, the lower its ID. The sets of the two input collections are stored as sorted arrays of integers (ascending sort order).

The sets of a collection are ordered by their size. Sets with the same size are sorted lexicographically. We investigate the influence of lexicographical sorting in Section 5.7.

The index is an array with pointers to inverted lists. The token number is an array position, and the inverted list (possibly empty) is accessed in constant time. A list entry for token τ is a pair (s, p_s), where s is a set ID and p_s the position of token τ in the prefix of s. Thus, all required data for the positional filter is locally available during pre-candidate generation. The position is not stored for ALL and ADP, which do not use the positional filter[5].

Candidate Set. The candidates, (r, s_i), are generated and verified per probing set r. We collect all candidates s_i for a given probing set r in a dynamic array of integers. Following [15], we store all temporary values required for processing and verifying a candidate pair physically close to s_i in the input collection. The temporary values are reset during verification.

Pre-Computation of Overlap. Normalized thresholds (e.g., Jaccard) must be translated into equivalent overlap thresholds. The required overlap for a pair (r, s) depends on the set sizes of r and s. Following `src-xiao` [21] (ALL/PPJ/PP+), we precompute these values in each probing cycle: for a given probing size $|r|$, the required overlaps for all eligible sets sizes $|s_i|$ are stored in an array.

[2] http://www.cse.unsw.edu.au/~weiw/project/simjoin.html
[3] https://www2.cs.sfu.ca/~jnwang/projects/adapt
[4] http://dbgroup.cs.tsinghua.edu.cn/ligl/simjoin
[5] Wang et al. [19] discuss position-aware pruning for search queries only; an extension to joins (ADP) is not discussed.

Algorithm 1: Verify($r, s, t, olap, p_r, p_s$)

Input: r, s: sets to be verified (sorted arrays);
t: required overlap;
$olap, p_r, p_s$: overlap up to positions p_r, p_s in r, s
Result: true iff $|r \cap s| \geq t$, i.e., (r, s) is in the result set

1. $maxr \leftarrow |r| - p_r + olap$; $maxs \leftarrow |s| - p_s + olap$;
2. **while** $maxr \geq t$ **and** $maxs \geq t$ **and** $olap < t$ **do**
3.   **if** $r[p_r] = s[p_s]$ **then**
4. $p_r \leftarrow p_r + 1$; $p_s \leftarrow p_s + 1$; $olap \leftarrow olap + 1$;
5.   **else if** $r[p_r] < s[p_s]$ **then**
6. $p_r \leftarrow p_r + 1$; $maxr \leftarrow maxr - 1$;
7.   **else** $p_s \leftarrow p_s + 1$; $maxs \leftarrow maxs - 1$;
8. **return** $olap \geq t$;

The precomputed overlap is accessed multiple times for each set s_i, and many sets may have the same size. Overall, we measure speedups of up to 30% and no slowdown on any of our datasets.

Verification. All join algorithms generate a set of candidates which must be verified. To get a fair comparison, we use the same verification routine for all competitors (Algorithm 1). The merge-like verification loop terminates as soon as the required threshold is met or cannot be reached due to the current matching position [15].

In addition to the candidate pair (r, s) and the required threshold, also the overlap up to positions p_r, p_s in r, s is passed to the verification routine. This overlap accounts for matches in the prefix and initializes the verification step. The value of this initial overlap depends on the filters applied during candidate generation and varies between different algorithms.

Algorithm-specific Notes. We discuss some implementation choices that are specific to the individual algorithms.

ALL. We do not need to store the token position in the inverted lists (no positional filter), which reduces their size by 50%.

PPJ. We follow the implementation of the original authors (`src-xiao`) [21] and apply the positional filter only to the first match in the prefix. This reduces the overhead of the positional filter, which is particularly relevant when only a small number of pre-candidates can be filtered. During pre-candidate generation, the position of the last matching token pair is maintained, which avoids redundant comparisons during verification [15].

PP+. As suggested by Xiao et al. [21], the suffix filter is applied to only one copy of each duplicate pre-candidate pair, (r, s_i). We maintain a flag with each set s_i in the input collection to keep track of suffix filter applications.

The suffix filter requires the user parameter MAXDEPTH, which controls the recursion depth. We use MAXDEPTH=2, which was also used by the original authors [21] for Jaccard and is the default in their implementation (`src-xiao`).

MPJ, PEL. MPJ deletes obsolete entries from the inverted lists, but no suitable data structure is discussed [15]. An obvious candidate is a linked list, which however shows poor scan performance. In our implementation we use a dynamic array and flag deleted items. We skip sequences of deleted items by storing a negative offset in the first item of a deleted sequence (cf. Figure 6). In our experiments, this data structure is up to two times faster and never slower than linked lists. The same data structure is used for PEL.

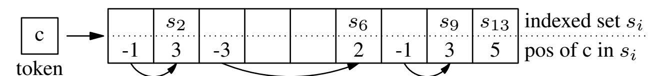


Figure 6: Inverted lists in MPJ and PEL.

ADP. The adaptive prefix index of ADP supports different prefix sizes. Instead of one inverted list per token, multiple inverted lists (one for each prefix size) are stored in the index. While ADP supports prefixes of arbitrary size, we observe that the cost function (which decides on the size of the prefix extension) rarely assumes values beyond 8. We limit the number of prefix sizes to 8 and store the inverted lists as an array of arrays. This design decision improves memory usage and runtime on all data points covered in this paper. The precision of the cost function for the prefix extension is controlled by a user parameter, which we set to $K = 3$ as suggested by the authors of ADP [19].

GRP. Sets with identical prefixes must be grouped during candidate generation. To identify duplicate prefixes in a single scan over the sets in the input collection, the sets must be lexicographically ordered by their prefixes (in addition to set sizes). We discuss the impact of sorting in Section 5.7.

Complexity of Source Code. The complexity of the implementation greatly varies between the algorithms. We report the lines of code (LoC) of our implementation in Table 3. Large portions of the code are shared between the algorithms: PPJ extends ALL with the positional filter; PP+, MPJ, and PEL extend PPJ; GRP needs additional code compared to PPJ to collapse and expand groups. With the adaptive prefix index and the cost function, ADP has the smallest overlap with the other algorithms.

Table 3: Lines of code for different join algorithms.

	ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
LoC	215	220	370	240	245	500	350

4. SETUP AND DATA SETS

Experimental Setup. We conduct our experiments on a machine with a six-core Intel Xeon E5-2630 v2 CPU with 2.6 GHz, 256 GB of RAM, 15 MB L3 cache (shared with the other cores), and 256 KB L2 cache (per core). We execute one join at a time with no other load on the machine. We compile our code with gcc -O3.

In this paper we focus on self-joins; Xiao et al. [21] discuss the transformation of nonself-joins to self-joins. Due to space constraints, we mainly discuss results for Jaccard normalization. All experiments were also conducted for Cosine and Dice with similar results; a summary is provided in Section 5.1, for details see [10].

Data Sets. We use 10 real-world datasets from different domains with very different characteristics (cf. Table 4):

AOL Query log of AOL search engine.[6] A set represents a search string; a token is a keyword in the search string.

BMS-POS Point of sale data.[7] A set is a purchase in a shop; a token is a product category in that purchase.

DBLP 100k random articles from DBLP bibliography.[8] A set is a publication; tokens are character q-grams of the concatenated title and author strings ($q = 2$, case insensitive).

ENRON Real e-mail data.[9] A set represents an e-mail; a token is a word from the subject or the body field.

FLICKR Photo meta-data [4]. A set is a photography; a token is a tag or a word from the title.

Table 4: Characteristics of datasets.

	#sets in coll.	set size max	set size avg	# diff. tokens
AOL	$1.0 \cdot 10^7$	245.0	3.0	$3.9 \cdot 10^6$
BMS-POS	$3.2 \cdot 10^5$	164.0	9.3	1657.0
DBLP	$1.0 \cdot 10^5$	869.0	82.7	6864.0
ENRON	$2.5 \cdot 10^5$	3162.0	135.2	$1.1 \cdot 10^6$
FLICKR	$1.2 \cdot 10^6$	102.0	10.1	$8.1 \cdot 10^5$
LIVEJ	$3.1 \cdot 10^6$	300.0	36.4	$7.5 \cdot 10^6$
KOSARAK	$6.1 \cdot 10^5$	2497.0	11.9	$4.1 \cdot 10^4$
NETFLIX	$4.8 \cdot 10^5$	$1.8 \cdot 10^4$	209.5	$1.8 \cdot 10^4$
ORKUT	$2.7 \cdot 10^6$	$4 \cdot 10^4$	119.7	$8.7 \cdot 10^6$
SPOT	$4.4 \cdot 10^5$	$1.2 \cdot 10^4$	12.8	$7.6 \cdot 10^5$
UNIFORM	$1.0 \cdot 10^5$	25.0	10.0	209.0
ZIPF	$1.0 \cdot 10^5$	84.0	50.0	$1.0 \cdot 10^5$

KOSARAK Click-stream data.[10] A set represents the user-behavior recorded on a Hungarian on-line news portal; a token is a link clicked by the user.

LIVEJ Social media data from LiveJournal.[11] A set represents a user; the tokens are user interests.

NETFLIX Social media data.[12] A set represents a user; a token is a movie rated by the user.

ORKUT Data from ORKUT social network.[11] A set is a user; a token is a group membership of the user.

SPOT Data from Spotify music streaming service.[13] A set is a user; a token is a track the user listened to.

We also generate two synthetic datasets with 100k sets each. We draw the set sizes from a Poisson distribution and the tokens from two different token distributions (Zipf, Uniform). Tokens are randomly assigned to sets until the precomputed set size is reached.

ZIPF Zipfian token distribution ($z = 1$), avg. set size 50.

UNIFORM Uniform token distribution, avg. set size 10.

We apply a single tokenization technique per dataset. Duplicate tokens that appear during the tokenization process are deduplicated with a counter that is appended to each duplicate (i.e., a unique integer from 1 to d is assigned to each of the d copies of a token). This deduplication technique increases the number of different tokens, e.g., DBLP has 3713 unique q-grams[14]; however, due to many duplicate q-grams per set, the deduplicated number of tokens is 6864.

The data sets have different characteristics (see Table 4). In particular, the sets in AOL are very short with a high number of different tokens. BMS-POS and DBLP both have a small number of different tokens and vary in the set sizes. ENRON, ORKUT, and NETFLIX feature long sets with a large number of different tokens. Most datasets, like FLICKR (cf. Figure 7), show a Zipf-like distribution and contain a large number of infrequent tokens (less than 10 occurrences), which favors the prefix filter. In contrast, NETFLIX has almost no tokens that occur less than 100 times, in BMS-POS only a quarter of the tokens occurs less than 10 times.

We use real-world data sets from different domains to avoid domain bias (e.g., the frequency of word tokens in string data typically follows a Zipf distribution). Although all tokens are translated to

[6] AOL: http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection

[7] BMS-POS: http://www.kdd.org/kdd-cup/view/kdd-cup-2000, provided by Blue Martini Software for KDD 2000 cup [22]

[8] DBLP: http://dblp.uni-trier.de/db (Feb. 2014)

[9] ENRON: https://www2.cs.sfu.ca/~jnwang/projects/adapt, preprocessed by Wang et al. [19]

[10] KOSARAK: http://fimi.ua.ac.be/data

[11] LIVEJ, ORKUT: http://socialnetworks.mpi-sws.org/data-imc2007.html [12]

[12] NETFLIX: http://www.cs.uic.edu/~liub/Netflix-KDD-Cup-2007.html, from Netflix Prize and KDD 2007 cup

[13] SPOT: http://dbis-twitterdata.uibk.ac.at/spotifyDataset [14]

[14] The large number of q-grams ($q = 2$) is due to the many different Unicode characters in titles and author names.

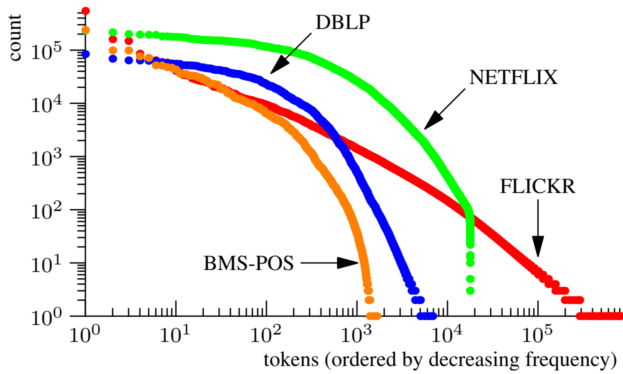


Figure 7: Histogram of token frequencies.

integers such that the original format gets lost, the characteristics of the dataset that impact the join performance are retained.

5. RUNTIME

In this section, we measure the runtime and do an in-depth analysis of the results. Each measurement is an average over 5 independent runs. We measure CPU time with the getrusage system call, which has a resolution of 4 ms.

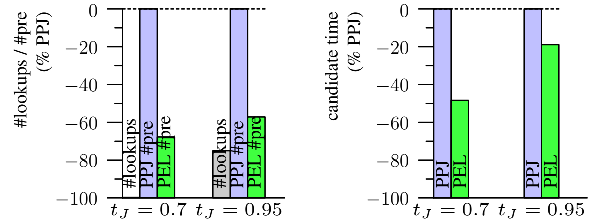
We break down the overall join time into candidate time and verification time. The *candidate time* includes pre-candidate and candidate generation (cf. Section 2.2), which are tightly interwoven in the implementation and result in a candidate set. The prefix index is built incrementally during pre-candidate generation and contributes to the candidate time. The *verification time* includes scanning the candidate set and verifying each candidate. The *join time* is the sum of candidate and verification time, and does not include the overhead for preprocessing (i.e., loading and sorting the collections, sorting the tokens by frequency).

5.1 Join Time

We measure the overall join time for all datasets on eight thresholds in the range [0.50, 0.95]. We next give a short overview of the results and provide a detailed discussion of the main influence factors in the remaining section.

Winners. Table 5 shows the fastest algorithm and its runtime (in seconds) for each data point; we also list the runner-ups that are within 10% from the fastest runtime. ALL wins on most data points (31), followed by PPJ (21), GRP (16), ADP (12), PEL (12), and MPJ (6). Often the winning algorithms show very similar performance: there are many runner-ups in the 10% range (and winners with identical runtimes within the resolution of the measurement method in two cases). Even the gap between the fastest and the slowest algorithm (*gap factor*) is surprisingly small: we measure a gap factor of at most 6.41; in many cases, the gap factor is much smaller (cf. Table 6). If we remove PP+ and ADP (the slowest algorithms) from the ranking, the gap factor is below 4.

Robustness. We analyze the robustness of the algorithms and measure the average, median, and maximum gap factor for each algorithm in Table 7(a). The most robust algorithm is GRP: on average, it is only 12% slower than the winner (including the cases when GRP wins), the gap is within 10% in half of the cases (median) and is 47% in the worst case (max). Also ALL and PPJ show good average and median performance, but are up 2.16 resp. 3.10 times slower than the winner in the worst case; the maximum gaps were observed on synthetic data (ALL, PPJ: UNIFORM, GRP: ZIPF); on real data, the maxima are 1.71 (ALL), 2.35 (PPJ), and 1.4 (GRP).



(a) Lookups and pre-candidates. (b) Candidate time.

Figure 8: Effect of lookups and pre-candidates on candidate time (NETFLIX).

PEL, which works best on nonself-joins [9], performs only slightly better than MPJ in our self-join scenario.

Threshold. In Table 8 we analyze the relationship between threshold and winning algorithm. We observe that ALL and GRP tend to be faster on large thresholds, while PPJ and ADP perform the best on small thresholds. This behavior is expected. For large thresholds the prefix filter (and grouping for GRP) is very effective, leading to pre-candidates with a small percentage of false positives. Any additional filter overhead for reducing the candidate set is hard to compensate for during verification. On small thresholds, however, filtering pre-candidates pays off. The good performance of PEL for small thresholds is discussed in Section 5.2.

ADP, PP+. Table 9 shows the slowest algorithm and all algorithms that are only up to 10% faster. The slowest algorithm is either ADP (60 times) or PP+ (36 times) with a clear gap to the second-slowest algorithm. PP+ applies the suffix filter to pre-candidates; unfortunately, the suffix filter is too slow compared to verification and never pays off. ADP spends much time on generating a small candidate set and cannot compensate this overhead during verification. We analyze these effects in Sections 5.4 (PP+) and 5.5 (ADP).

Cosine, Dice. Table 7 summarizes the results for Cosine and Dice: there is little difference w.r.t. Jaccard. This is not surprising since all normalized thresholds are translated into overlap thresholds between pairs of sets. All results in the remaining paper are based on Jaccard similarity; detailed results for Cosine and Dice are available from [10].

5.2 Candidate Time

We count the number of lookups (#lookups), pre-candidates (#pre), and candidates (#cand), which turn out to be key indicators for the candidate time. A lookup returns an inverted list for a probing token. We discuss ALL, PPJ, MPJ, and PEL. ADP, GRP, and PP+ adopt different techniques and are treated in separate sections.

Lookups. The number of lookups is identical for ALL, PPJ, MPJ, and PEL. In many cases, #lookups $\ll$ #pre, and the candidate time mainly depends on the number of pre-candidates. Figure 8 illustrates that #lookups has a visible impact on the candidate time if #lookups $\sim$ #pre. Figure 8(a) shows #lookups and #pre for PPJ and PEL; all numbers are percentages of #pre for PPJ. Figure 8(b) shows the respective candidate times (percentage of PPJ candidate time).

For threshold $t_J = 0.7$, #lookups $\ll$ #pre (by a factor of 108 for PEL resp. 336 for PPJ): the small number of pre-candidates of PEL w.r.t. PPJ translates into a much faster candidate time. For threshold $t_J = 0.95$, #lookups $\sim$ #pre: although PEL is still faster in generating candidates, the large number of lookups leads to a runtime offset, and the reduced number of pre-candidates is less visible in the overall candidate time.

Table 5: Fastest algorithms.

Dataset	Jaccard threshold 0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	PEL 333 PPJ	PEL 83.7 PPJ GRP	GRP 13.2 ALL PPJ	ALL 8.58 GRP	GRP 4.20 ALL	GRP 1.77	GRP 1.46	GRP 1.43
BMS-POS	PPJ 44.9	PPJ 15.6	PPJ 4.78 ADP GRP	PPJ 2.74 GRP ADP ALL	ALL 1.27 GRP PPJ	ALL 0.447 GRP	GRP 0.170	GRP 0.068
DBLP	ADP 105	ADP 48.5	ADP 19.4	ADP 10.8	ADP 5.15	ADP 2.08	PPJ 0.690 ADP	ALL 0.116 PPJ
ENRON	PPJ 53.3 GRP	PPJ 16.2 GRP	PPJ 4.79 PEL GRP MPJ	PEL 2.63 PPJ GRP MPJ	PEL 1.51 PPJ MPJ GRP	PEL 0.884 MPJ PPJ ALL GRP	ALL MPJ 0.396 PEL PPJ GRP	ALL 0.174 MPJ PEL PPJ
FLICKR	PEL 14.4 PPJ	PPJ 5.77 PEL ALL	ALL 2.73 PPJ PEL	ALL 2.03 PPJ PEL	ALL 1.21 PPJ PEL	ALL 0.696 PPJ PEL	ALL 0.403	ALL 0.243 GRP PEL MPJ
KOSARAK	PEL 47.3 PPJ GRP	PEL 9.43 PPJ GRP	PPJ 1.60 ALL GRP PEL	ALL 0.909 PPJ	ALL 0.484 GRP	ALL 0.232 GRP	ALL 0.140 GRP	GRP 0.087 ALL
LIVEJ	PPJ 345 PEL	PEL 88.9 PPJ	PEL 22.1 PPJ	PEL 12.0 PPJ GRP	PEL 6.52 PPJ GRP MPJ	PPJ 3.50 GRP PEL MPJ ALL PP+	MPJ 1.88 ALL PEL PPJ GRP PP+	ALL 1.02 MPJ PEL
NETFLIX	ALL 1235 PPJ GRP	ALL 494 ADP PPJ GRP	ADP 146	ADP 76.4	ADP 36.6	ADP 15.6 PPJ GRP	PPJ 4.73 GRP	PPJ 0.894 ALL PEL GRP MPJ
ORKUT	PPJ 213 GRP	PPJ 79.4 GRP	PPJ 33.4 GRP PEL	GRP 21.0 PPJ PEL MPJ	GRP 12.9 PPJ MPJ PEL	MPJ 7.69 GRP PPJ PEL ALL PP+	MPJ 4.28 ALL PEL GRP PPJ PP+	MPJ 2.06 ALL PEL PP+ PPJ
SPOT	ALL 0.542 PPJ PEL MPJ	ALL 0.321 MPJ PEL PPJ	ALL 0.198 MPJ PEL	ALL 0.166 MPJ PEL PP+	ALL 0.134 MPJ PEL PP+	ALL MPJ 0.110 PEL PP+	ALL 0.090 MPJ PEL PP+	ALL 0.073 MPJ PEL PP+ PPJ
UNIFORM	ADP 44.5	ADP 24.5 PPJ ALL	GRP 8.99 PPJ	GRP 4.54	GRP 1.80	GRP 0.533	GRP 0.245	GRP 0.055
ZIPF	PPJ 2.41 ADP	PPJ 0.930	PPJ 0.412 ALL	PPJ 0.286 ALL	ALL 0.190 PPJ	ALL 0.114 PPJ	ALL 0.065 MPJ	ALL 0.032 MPJ

Pre-Candidates and Candidates. In Figure 9, we analyze the impact of pre-candidates and candidates on the runtime. We discuss ALL, PPJ, MPJ, and PEL, which produce different numbers of pre-candidates; the following relationship holds between the pre-candidate sets: $PEL \subseteq MPJ \subseteq PPJ = ALL$. The number of candidates is always smaller than (or equal to) the pre-candidate number, and the following relationship holds: $PEL = MPJ = PPJ \subseteq ALL$. We pick the ENRON and the AOL datasets, which respond very differently to (pre-)candidate filters.

ALL and PPJ process the largest number of pre-candidates (cf. Figure 9(a)). ALL deduplicates the pre-candidates in a single

Table 6: Gap factor per dataset and threshold.

Dataset	Jaccard threshold 0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	3.37	3.57	3.64	4.39	5.13	5.75	6.35	6.41
BMS-POS	1.70	1.58	1.52	1.50	1.58	1.76	2.35	4.53
DBLP	3.69	3.93	3.51	3.17	2.79	2.29	1.65	2.06
ENRON	2.70	2.21	1.72	1.70	1.94	2.09	2.41	2.82
FLICKR	2.06	1.98	2.04	2.14	2.44	2.92	3.67	4.80
KOSARAK	3.06	3.00	1.81	1.87	2.31	3.03	3.78	4.76
LIVEJ	1.99	2.09	2.23	2.34	2.53	2.85	3.40	4.52
NETFLIX	3.08	2.93	2.92	2.63	2.25	1.77	1.43	2.10
ORKUT	2.53	1.84	1.92	2.06	2.17	2.19	2.41	3.19
SPOT	2.22	2.72	3.19	3.38	3.63	3.90	4.38	5.02
UNIFORM	2.29	1.74	1.68	2.07	2.76	3.19	3.64	4.45
ZIPF	2.45	2.11	1.70	1.81	2.23	3.01	4.06	5.90

Table 7: Summary statistics of gap factors.

(a) Jaccard.

Gap Factor	Algorithm ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
average	1.19	1.17	1.81	1.35	1.28	2.46	1.12
median	1.11	1.07	1.64	1.25	1.14	2.17	1.10
maximum	2.16	3.10	4.45	3.80	3.49	6.41	1.47

(b) Cosine.

	Algorithm ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
average	1.28	1.10	1.96	1.32	1.23	2.15	1.14
median	1.22	1.03	1.72	1.25	1.10	1.97	1.11
maximum	1.95	2.39	4.05	2.83	2.67	6.32	1.70

(c) Dice.

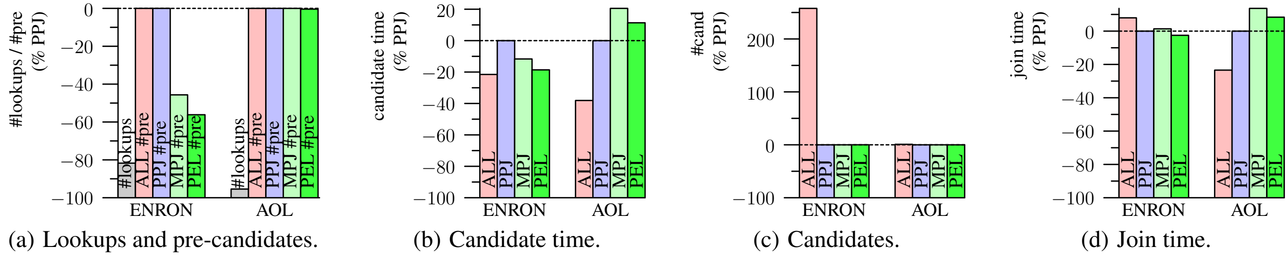
	Algorithm ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
average	1.24	1.11	1.97	1.33	1.26	2.04	1.13
median	1.19	1.03	1.69	1.26	1.12	1.87	1.11
maximum	2.08	2.50	4.05	2.95	2.75	6.28	1.71

Table 8: Winner per threshold.

Jaccard threshold	Algorithm ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
0.50	2	5	0	0	3	2	0
0.60	2	5	0	0	3	2	0
0.70	2	5	0	0	1	2	2
0.75	4	2	0	0	2	2	2
0.80	5	0	0	0	2	2	3
0.85	5	1	0	2	1	2	2
0.90	5	2	0	3	0	0	3
0.95	6	1	0	1	0	0	4
sum	31	21	0	6	12	12	16

Table 9: Slowest algorithms.

Dataset	Jaccard threshold 0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
BMS-POS	PP+	PP+	PP+	PP+	PP+	PP+	ADP	ADP
DBLP	PP+	PP+	PP+	PP+	PP+	PP+	PP+	ADP
ENRON	PP+	PP+	PP+	ADP	ADP	ADP	ADP	ADP
FLICKR	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
KOSARAK	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
LIVEJ	ADP PP+	ADP	ADP	ADP	ADP	ADP	ADP	ADP
NETFLIX	PP+	PP+	PP+	PP+	PP+	PP+	PP+	ADP
ORKUT	PP+	PP+ ADP	ADP	ADP	ADP	ADP	ADP	ADP
SPOT	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
UNIFORM	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ZIPF	PP+	PP+	PP+	ADP	ADP	ADP	ADP	ADP



(a) Lookups and pre-candidates. (b) Candidate time. (c) Candidates. (d) Join time.

Figure 9: Pre-candidates and candidates, $t_J = 0.8$.

scan of the inverted lists (by marking duplicate sets in the input collections) and applies no additional filter to generate candidates. Thus, ALL is very fast at generating candidates (cf. Figure 9(b)), but compared to PPJ, the candidate set is larger. PPJ additionally applies the positional filter (cf. Figure 9(c)) and must compensate for the longer candidate time during verification. The overhead pays off for ENRON but not for AOL, where the positional filter has almost no effect (cf. Figure 9(d)).

MPJ and PEL reduce the number of pre-candidates: MPJ removes hopeless inverted list entries from the index, PEL in addition leverages the matching position to better crop the lists. Fewer pre-candidates and shorter inverted lists for subsequent lookups improve the runtime and must outweigh the filter overhead and a more costly candidate verification (cf. Section 5.3). PEL is very effective on ENRON and reduces #pre by 56%; as a result, PEL outperforms PPJ in candidate (by 19 %) and join time (by 2%); MPJ cannot outweigh the increase in verification time. On AOL, MPJ and PEL reduce #pre by less than 1% and are outperformed by PPJ. We observe that the pre-candidate filters are typically more effective on small thresholds, e.g., PEL reduces #pre by less than 1% (20%) for $t_J = 0.95$, but 23 % (71 %) for $t_J = 0.5$ on AOL (ENRON).

5.3 Verification

The verification algorithm computes the overlap of two sets in a merge-like fashion and stops early whenever possible (cf. Section 3). We show that verification is surprisingly efficient: although it is linear in the set size in the worst case, the average runtime is constant for most datasets.

The efficiency of verification puts pressure on filter techniques that reduce the number of candidates (cf. Table 2). These filters only pay off if they are much faster on false positives than verification. Note that true positives must still go through verification: any filter effort on them is lost.

The verification of a candidate pair (r, s) proceeds in two steps: (a) decide on the start positions p_r, p_s in the two sets; (b) iterate through the verification loop (cf. Algorithm 1). The start position is right after the so-called *effective prefix* in one of the sets and the last matching token in the other set. The matches in the effective prefix are computed during candidate generation and need no extra effort during verification. The effective prefix is identical to the (extended) prefix for ALL, PPJ, GRP, and ADP, but is shorter for the indexed set in MPJ and for both sets in PEL. For computing the starting positions, we need to access the last token in the extended prefixes of r and s in the input collections.

Number of Token Comparisons. We count the number of token comparisons for false positives, i.e., the number of iterations in the verification loop (line 2 in Algorithm 1). Token comparisons need to access the input collections R and S, which are unlikely to be in the cache. The average number of comparisons for PPJ is shown in Table 10. Note that the number of comparisons can be zero if the loop conditions hold before the loop is entered. Such a situation is illustrated in Figure 10: there are not enough tokens left in r to satisfy the required overlap (p_r and p_s are circled).

Table 10: Avg. number of token comparisons, PPJ.

Dataset	Jaccard threshold							
	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	1.1	1.1	1.2	1.2	1.1	1.0	1.0	1.0
BMS-POS	0.86	0.84	0.97	1.0	1.1	1.1	1.1	1.0
DBLP	1.5	0.85	0.64	0.57	0.55	0.54	0.55	0.89
ENRON	0.81	1.0	1.9	2.9	4.9	11	13	18
FLICKR	1.2	1.5	2.3	3.2	4.5	4.9	2.5	2.4
KOSARAK	1.1	1.1	1.2	1.2	1.3	1.4	1.3	1.2
LIVEJ	0.66	0.69	0.88	0.99	1.1	1.2	1.3	1.1
NETFLIX	1.3	0.64	0.42	0.36	0.32	0.30	0.29	0.34
ORKUT	1.0	0.98	0.93	0.93	0.91	0.87	0.82	0.93
SPOT	0.82	0.88	0.92	0.99	0.99	1.0	1.0	1.0
UNIFORM	1.3	1.1	1.1	1.1	1.1	1.3	1.3	1.1
ZIPF	0.72	0.60	0.58	0.56	0.54	0.55	0.64	0.62

Jaccard threshold $t_J = 0.8 \Rightarrow$ # required token matches = 9

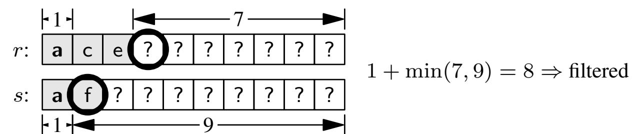


Figure 10: Verification with zero token comparisons.

On most datasets, on average only 0.3 to 1.5 token comparisons are required to identify a false positive. This is extremely efficient. We find larger values only for FLICKR (up to 4.9) and ENRON (up to 18). Interestingly, the efficiency of the verification does not depend on the set size: FLICKR has small sets (avg. size 10.1) but is expensive during verification; NETFLIX has long sets (avg. size 209.5), but requires less than 1 comparison on most thresholds.

We discuss the numbers for ALL, MPJ, and PEL, which are not reported in Table 10. ALL often requires slightly fewer token comparisons than PPJ, but may also require more (e.g., on NETFLIX and ORKUT). This stems from two competing effects: (a) ALL verifies additional candidates that PPJ eliminates with the positional filter; these candidates are easy to verify. (b) ALL does not store positional information, thus the last prefix matching position must be approximated by the number of matching tokens, which is only a lower bound. MPJ, PEL: Since the effective prefix is shorter, the verification is typically more costly. In particular for long sets (ENRON, NETFLIX, ORKUT, LIVEJ) we get larger numbers, e.g., MPJ needs 3.9 to 16.0 and PEL needs 3.9 to 18.4 token comparisons on NETFLIX.

5.4 PP+: Suffix Filter

PP+ extends PPJ with the suffix filter. The suffix filter accesses the sets in the input collections and rejects false positive pre-

candidates. All true positives and possibly some false positives are accepted. The accepted pre-candidates go through verification. PPJ does not apply the suffix filter and sends the input of the suffix filter directly to verification.

Unfortunately, the suffix filter is too slow compared to verification and typically does not pay off. Figure 11 shows the slowdown of PP+ w.r.t. the winner on all data points: PP+ never wins and cannot compete with the best algorithms.

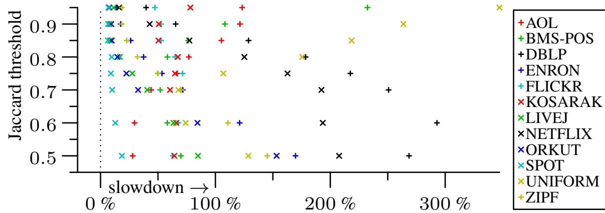


Figure 11: PP+ vs. competing algorithms.

We investigate the performance of the suffix filter vs. verification in Figure 12, where we count the average number of CPU cycles for PPJ (only verification) and PP+ (suffix filter and verification).[15] We analyze datasets with short (BMS-POS), medium (LIVEJ), and long sets (ORKUT).

On BMS-POS and LIVEJ, we observe that PPJ is faster than PP+ in rejecting false positives during verification. PPJ also verifies the false positives that the suffix filter rejects in the case of PP+, i.e., the suffix filter removes easy-to-verify candidates. There is almost no runtime difference for ORKUT.

For the suffix filter to pay off, it must reject false positives faster than verification. Thereby, we need to consider the verification time of PPJ. Figure 12 shows that the suffix filter is always slower, which explains the poor overall performance. Note that the suffix filter also spends time on candidates that pass the filter, e.g., true positives; this time is lost and adds to the verification time of PP+. The overhead is significant: the suffix filter is slower on candidates that pass the filter since on rejected candidates it stops early.

The poor performance of the suffix filter w.r.t. verification is surprising since (in the worst case) verification is linear in the set size and the suffix filter is constant (2^{MAXDEPTH}). We attribute this result to two effects: (a) on average, verification accesses only a small, constant number of tokens, as discussed in Section 5.3; (b) verification does a linear scan and benefits from memory locality, while the suffix filter uses binary search. Interestingly, longer sets are not necessarily in favor of the suffix filter: PP+ slightly outperforms PPJ on SPOT (avg. set size 12.8), but is much slower on ENRON, NETFLIX, and ORKUT (avg. set size > 100).

5.5 ADP: Extended Prefix

We analyze the performance of ADP and compare it to ALL (which ADP extends). ADP inspects an extended prefix and maintains multiple inverted lists per token in the index (cf. Sections 2.3). The non-extended prefix part is treated like in ALL; the prefix extension requires additional index lookups such that ADP produces at least the pre-candidates of ALL. The extension is useful to reduce the candidate set, which is always a subset of ALL.

In Figure 13 we show our experimental results on LIVEJ, which represents the typical performance of ADP, and NETFLIX, where ADP performs particularly well. As expected, ADP does more lookups and processes more pre-candidates than ALL. On both datasets, ADP is very effective at reducing the candidate set, which translates into short verification times. However, the extended prefix leads to much longer candidate times. For LIVEJ, the time spent during candidate generation cannot be recovered during verification, increasing the overall join time. In the case of NETFLIX, the extended prefix pays off and ADP is faster than ALL. — Some of the candidate generation time is spent to build the index, which is more expensive for ADP: 5.8 s for LIVEJ (ALL: 2.2 s), 0.75 s on NETFLIX (ALL: 0.35 s).

On Zipf-like distributions, the prefix filter performs very well and the extended prefix does not pay off; in fact, ADP is the slowest algorithm on all thresholds for the Zipf-like datasets AOL, FLICKR, KOSARAK, LIVEJ, and SPOT. When there are few infrequent tokens in the dataset (BMS-POS, DBLP, NETFLIX, cf. Figure 7), the prefix filter generates many false positives; on these datasets ADP wins on some thresholds. On UNIFORM, ADP wins only on very small thresholds, but is never among the slowest algorithms; on ZIPF, ADP is the slowest algorithm for $t_J \geq 0.75$, and the performance gradually improves with smaller thresholds.

5.6 GRP: Dealing with Duplicate Prefixes

GRP groups sets with identical prefixes during candidate generation. During verification, the groups are expanded since identical prefixes may stem from non-identical sets. Grouping leads to fewer lookups and fewer pre-candidates. In the absence of duplicate prefixes, GRP behaves like PPJ.

Figure 14(a) shows the performance of GRP vs. PPJ on KOSARAK (no duplicate sets). GRP reduces the number of lookups and pre-candidates, indicating the presence of duplicate prefixes (32.3% for $t_J = 0.8$). The candidate time benefits from grouping, and GRP also achieves better join time than PPJ. Larger thresholds lead to shorter prefixes and more duplicates. On KOSARAK, GRP has the greatest advantage over PPJ on $t_J = 0.95$ and is slightly slower than PPJ for $t_J = 0.5$. This is also visible from Table 8: the higher the threshold, the better the performance of GRP.

Verification: Although GRP and PPJ have identical candidate sets, GRP involves the overhead of unfolding groups, which explains the verification times in Figure 14(a). The unfolding overhead is partially absorbed by better memory locality, which is the predominant effect in Figure 14(b).

In Figure 14(b) we compare GRP vs. PPJ on a version of KOSARAK with duplicates, i.e., we execute the join without removing duplicate sets from the collections. The percentage of duplicate sets is 38.7%, the resulting number of duplicate prefixes is 58.5%. In this setting, GRP clearly has an edge over PPJ. Note, however, that GRP also must verify all pairs of identical (duplicate) sets, which does not scale for duplicates that appear frequently.

5.7 Lexicographical Sorting

The sets in the two input collections are ordered by size, and sets of the same size are lexicographically sorted by tokens. Infrequent tokens precede frequent tokens in the sort order. The sort order is unique, and sorting ensures deterministic behavior of the algorithms.

We compare the join time for collections that are lexicographically sorted vs. randomly shuffled. Figure 15 shows that lexicographical sorting has a significant impact on the runtime, and all algorithms benefit from sorting (while only GRP explicitly leverages the sort order to detect duplicate prefixes). We observe that sorting is more beneficial for shorter prefixes: while on AOL (short sets) PPJ is 4.9 times faster with sorting, the best improvement on ENRON (long sets) is below 5%. Also high thresholds decrease the

[15] We use the benchmarking method described in [13] to acquire the CPU cycles for each candidate.

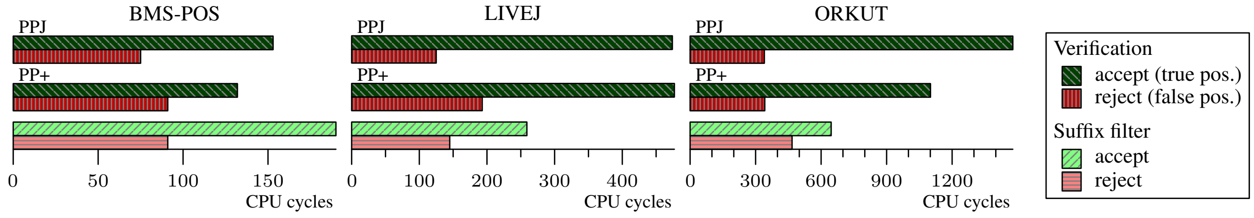
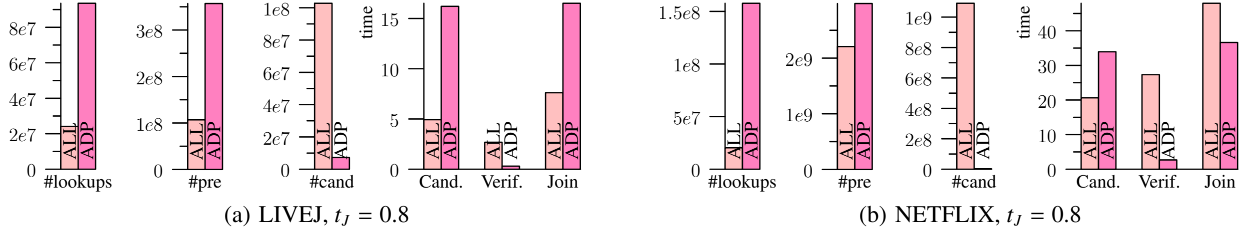
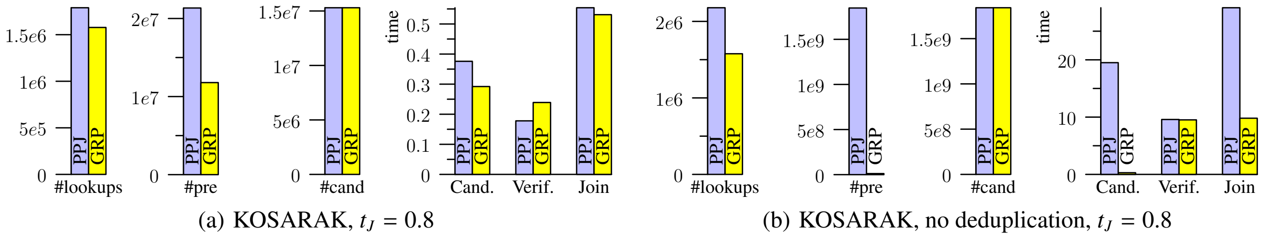


Figure 12: Cost of suffix filter and verification, average CPU cycle count per candidate, $t_J = 0.8$.



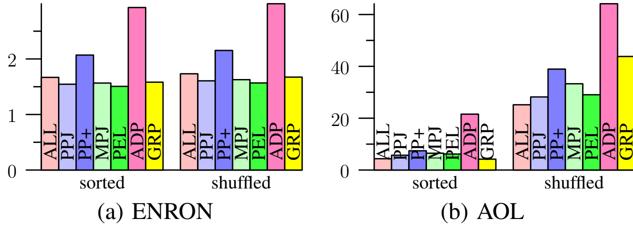
(a) LIVEJ, $t_J = 0.8$ (b) NETFLIX, $t_J = 0.8$

Figure 13: ADP vs. ALL: Candidate generation, verification, and join time.



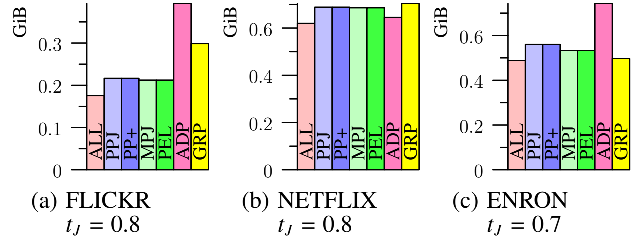
(a) KOSARAK, $t_J = 0.8$ (b) KOSARAK, no deduplication, $t_J = 0.8$

Figure 14: GRP vs. PPJ: Candidate generation, verification, and join time.



(a) ENRON (b) AOL

Figure 15: Runtime, $t_J = 0.8$.



(a) FLICKR $t_J = 0.8$ (b) NETFLIX $t_J = 0.8$ (c) ENRON $t_J = 0.7$

Figure 16: Memory usage (in bytes).

prefix size: PPJ on DBLP benefits with 20% for $t_J = 0.95$, but only with 2% for $t_J = 0.7$.

We identify two beneficial effects of sorting: (a) Cache friendliness: we measure 10 times more cache misses for shuffled input (PPJ, AOL, $t_J = 0.8$)[16]. (b) Less pre-candidates: The index is built on the fly. Sorting favors infrequent tokens (with short inverted lists) to be indexed first, thus subsequent lookups return fewer pre-candidates. However, this effect is less significant: the maximum reduction of pre-candidates in our test is 13% (ALL and PPJ on AOL, $t_J = 0.75$).

6. MEMORY USAGE

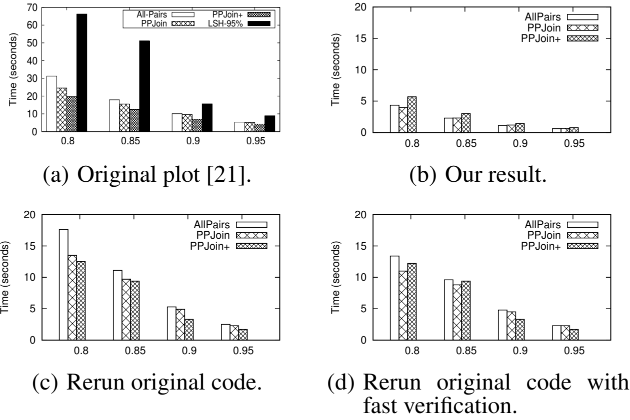
We study the memory usage[17] of all join algorithms in Figure 16. The following structures are stored on the heap: (a) the input collections, (b) the (extended) prefix index, (c) the candidate set. Minor size differences between the input collections are due to meta data that is stored with the input sets and may vary between the algorithms. The candidate set is reset per probing set and its size is negligible. Thus, the main factor for memory differences is the index.

Figure 16(a) depicts the typical behavior: ALL uses the smallest amount of memory (since it does not store positional information), MPJ and PEL use less memory than PPJ (since obsolete entries are removed and the free position may be reused), GRP requires some extra storage to keep track of prefix groups, ADP must also store the prefix extensions.

Figure 16(b) shows an interesting case in which ADP requires less memory than most other algorithms: the lack of position information (like in ALL) outweighs the overhead for the extended prefix. Figure 16(c) shows the only data point where GRP is the runner-up after ALL: the grouping overhead is outweighed by the smaller index (since only one representative per group is indexed).

[16] Cache misses counted with Linux perf tools.

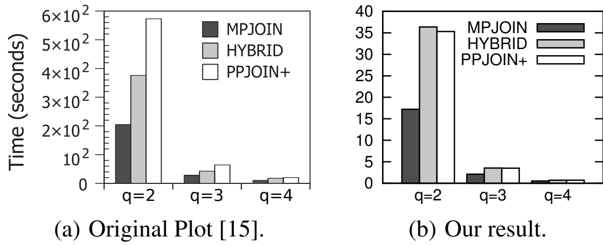
[17] Heap memory measured with Linux memusage.



(a) Original plot [21]. (b) Our result.

(c) Rerun original code. (d) Rerun original code with fast verification.

Figure 17: Reproducing results of [21]: PPJ, PP+, and ALL on ENRON (with duplicates), Jaccard.



(a) Original Plot [15]. (b) Our result.

Figure 18: Reproducing results of [15]: MPJ, HYBRID, and PP+ on DBLP (q-grams), Jaccard.

7. PREVIOUS EXPERIMENTAL RESULTS

We repeat core runtime results of three major works on set similarity joins: Xiao et al. [21] (PPJ, PP+), Ribeiro and Härder [15] (MPJ), and Wang et al. [19] (ADP). We further discuss results by Jiang et al. [8], who evaluate set similarity joins in the context of string similarity algorithms.

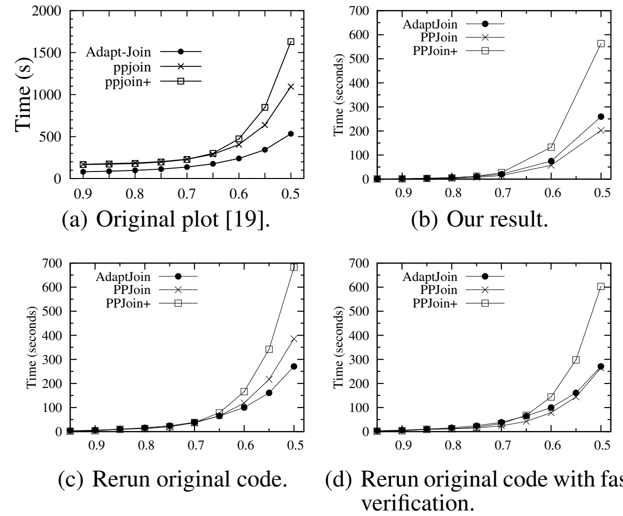
The overall outcome is the following. (1) We are able to reproduce all results with the join implementations that were used in the original papers. (2) With our own implementation, we get different runtime results and in some cases cannot confirm the relative performance of the competing algorithms. (3) We identify the efficiency of the verification routine as the main origin of the runtime differences between the implementations. In our implementation, we use the same (efficient) verification routine for all join algorithms. An inefficient verification step favors algorithms that spend more time on filtering candidates (cf. Section 5.3).

7.1 PPJoin/PPJoin+

Xiao et al. [21] present PPJ and PP+. We repeat a self join experiment on ENRON[18], which compares the runtimes of PPJ, PP+, ALL, and LSH-95% (approximate algorithm, not covered in this paper). Figure 17(a) shows the plot of the original paper [21], our results are shown in Figure 17(b). The absolute runtimes differ, which is expected since we use a faster processor. In addition, in our experiment (1) PP+ is slower than all other algorithms, and (2) the relative performance of ALL is better.

We analyze the relative performance loss of PP+. With the original code provided by the authors (`src-xiao`) we are able to reproduce their results on our hardware, as shown in Figure 17(c). Note that our implementation is faster on all data points. An analysis of the original source code reveals that the verification misses important optimization opportunities [15]. PP+ applies the suffix filter on the candidate set produced by PPJ, thus the suffix filter only pays off if it is faster than the final verification (cf. Section 5.4). We attribute the poor relative performance of PP+ in our experiments to our fast verification routine. This hypothesis is supported by Figure 17(d), where we substitute the verification routine in the original source code with the verification that we use in our experiments (Algorithm 1).

[18]We do not remove duplicate sets to reproduce [21].



(a) Original plot [19]. (b) Our result.

(c) Rerun original code. (d) Rerun original code with fast verification.

Figure 19: Reproducing results of [19]: ADP, PPJ, and PP+ on ENRON (with duplicates), Jaccard.

7.2 MPJoin

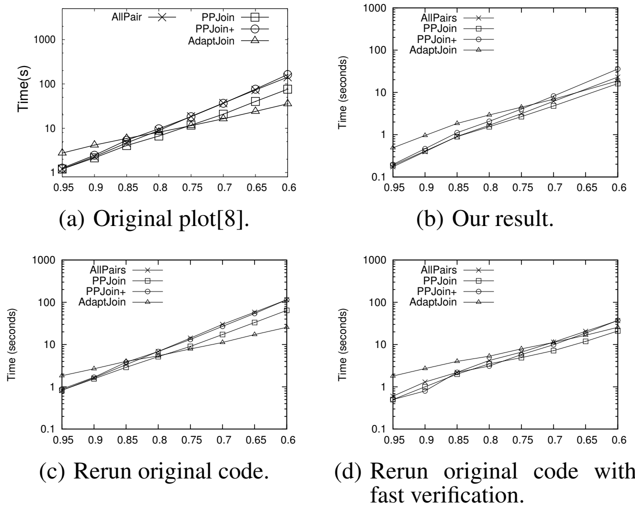
Ribeiro and Härder [15] present MPJ. We repeat a self-join experiment on DBLP (character q-grams, $q \in \{2, 3, 4\}$, $t_J = 0.75$), with MPJ, HYBRID (MPJ with suffix filter), and PP+. The original results are shown in Figure 18(a), Figure 18(b) shows our results. The absolute runtimes differ due to different hardware and programming language (Java vs. C++).

Our results confirm the good performance of MPJ. Unlike in the original plot, PP+ is at least as fast as HYBRID in our experiments. A breakdown of the runtime shows that PP+ outperforms HYBRID in generating the input for the suffix filter. The reason is that we precompute the overlap per probing set (cf. Section 3), while the overlap is computed per pre-candidate in the original code (as confirmed by the authors of [15]). PP+ benefits more from this optimization than HYBRID since PP+ processes more pre-candidates.

7.3 AdaptJoin

Wang et al. [19] present ADP. We repeat a self-join experiment on ENRON (with duplicates, like [19]), which compares the runtimes of PPJ, PP+, and ADP. Figures 19(a) and 19(b) show the original plot and our results, respectively. The original plot includes preprocessing, which explains the offset w.r.t. our plot.

In our experiment, PPJ is faster than ADP. The difference is due to the slow verification algorithm used by the original authors. We run a similar experiment as in Section 7.1 to verify this claim. Figure 20(c) shows the plot generated with the original code (preprocessing offset removed), in Figure 19(d) we apply efficient verification. We use the following setting: ADP runs with the binaries provided by the original authors (`bin-wang`) [19], the other algo-



(a) Original plot[8]. (b) Our result.

(c) Rerun original code. (d) Rerun original code with fast verification.

Figure 20: Reproducing results of [8]: ALL, PPJ, PP+, and ADP on ENRON (dedup.), Jaccard.

rithms run with the original PPJ/PP+ code (`src-xiao`) [21]. We estimate the preprocessing overhead of the ADP binary as the runtime for $t_J = 0.99$ (negligible join time) and subtract this offset (55.1 s) in the plot. We cannot substitute the verification in the ADP binary; however, ADP benefits less from efficient verification than its competitors due to the smaller candidate set (from -7% for $t_J = 0.95$ to -96% for $t_J = 0.6$ w.r.t. PPJ).

7.4 String Similarity Join

Jiang et al. [8] evaluate set similarity joins in the context of string similarity techniques. We repeat a self-join experiment on ENRON (without duplicates, like [8]) in Figure 20. In our experiment, PPJ consistently outperforms ADP. We proceed as in Section 7.1. In Figure 20(c) we use the binaries provided by the authors (`bin-jiang`) [8]; in Figure 20(d) we use the original PPJ/PP+ code (`src-xiao`) [21], but apply efficient verification.

8. CONCLUSIONS

We have studied seven recent algorithms for set similarity joins, which all use a filter-verification framework. We showed that verification is surprisingly fast and plays a key role in the runtime comparison between algorithms. Most previous work has focused on building effective and complex filters, which turn out to be too slow compared to efficient verification: The plain application of the prefix-filter in AllPairs is still competitive; later improvements, in particular, PPJoin and GroupJoin, moderately outperform AllPairs on average, whereas complex techniques like PPJoin+ and AdaptJoin are rarely competitive. Based on our findings, we do not expect significant impact from future techniques that sit on top of the prefix filter, but see opportunities in fast candidate generation.

Acknowledgments. This work was partially funded by the Austrian Science Fund (FWF) through the Doctoral College GIScience (DK W 1237-N23) at the University of Salzburg. We thank Daniel Kocher for typesetting Figure 2.

9. REFERENCES

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

[2] N. Augsten and M. H. Böhlen. *Similarity Joins in Relational Database Systems*. Synt. Lect. on Data Management. Morgan & Claypool Publishers, 2013.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. WWW*, pages 131–140, 2007.

[4] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, Nov. 2012.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. ICDE*, pages 5–16, 2006.

[6] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. MassJoin: A MapReduce-based method for scalable string similarity joins. In *Proc. ICDE*, pages 340–351, 2014.

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

[8] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[9] W. Mann and N. Augsten. PEL: Position-enhanced length filter for set similarity joins. In *Proc. GvD (Foundations of Databases)*, pages 89–94, 2014.

[10] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. Technical Report 2016-03, Department of Computer Sciences, University of Salzburg, Austria, Apr. 2016.

[11] A. Metwally and C. Faloutsos. V-SMART-Join: A scalable MapReduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[12] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf. (IMC)*, 2007.

[13] G. Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures (white paper). Technical report, Intel Corporation, Sept. 2010.

[14] M. Pichl, E. Zangerle, and G. Specht. Combining spotify and twitter data for generating a recent and public dataset for music recommendation. In *Proc. GvD (Foundations of Databases)*, pages 35–40, 2014.

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

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

[17] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[18] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proc. SIGMOD*, pages 495–506, 2010.

[19] 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, 2012.

[20] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. WWW*, pages 131–140, 2008.

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

[22] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. KDD*, pages 401–406, 2001.