

An Empirical Evaluation of Set Similarity Join Techniques [Experiments and Analyses]

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 Bours
Aarhus University
Aarhus, Denmark
pbours@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 technique 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 novel techniques have been developed. The fastest algorithms use a filter-verification framework and leverage (some variant of) *prefix filtering* [5].

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 algorithms¹ (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].

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: 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 most of the 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 length, in our experiments the number of required comparisons is a small constant (often 2 or less, 18 at most), independent of the set length.

Sophisticated filters are too slow. Due to efficient verification, expensive filters do not pay off: we measure the slowest runtimes for AdaptJoin (extended prefix) or PPJoin+ (suffix filter), which produce the smallest candidate sets.

We repeat key experiments from four previous works [21, 15, 19, 8] 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. Further, [8] does not evaluate GroupJoin and MPJoin(-PEL), which turn out to be relevant competitors. We repeat an experiment of [8] and discuss diverging results.

The works that introduced the join techniques in this test [3, 21, 15, 9, 19, 4] empirically evaluate their solutions. 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 from the original works.

We focus on main memory techniques and do not discuss distributed set similarity algorithms, e.g., [18, 11, 6]. 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]).

To the best of our knowledge, this is the first empirical study that (a) includes all key players in the field, (b) covers

¹Source code available: <http://dbresearch.uni-salzburg.at/ssjoin>

datasets from a wide range of different applications, and (c) provides an in-depth analysis of the runtime results.

Outline. In Section 2 we revisit set similarity joins. Section 3 discusses relevant implementation choices, and Section 4 defines 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. Two sets are similar if their 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]. 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 $R \bowtie S = \{(r, s) \in R \times S \mid \text{Sim}(r, s) \geq t\}$.

Prefix Filter. A key technique used by all state-of-the-art set similarity join algorithms 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 k -prefix of a set is formed by the k very first elements of the set in a given total order. With appropriate prefix lengths, two sets can be pruned if their prefixes have no common element. The prefix length depends on the similarity threshold and the similarity function. For example, the prefix filter for overlap similarity is defined as follows: Given two sets, r and s , and an overlap threshold t , if $|r \cap s| \geq t$, there is at least one common token within the $|r| - t + 1$ -prefix of r and the $|s| - t + 1$ -prefix of s . The same principle also holds for normalized similarity measures, for example, for Jaccard the prefix length is $|r| - \lceil t|r \rceil + 1$ (see [9]).

2.2 Algorithmic Framework

All tested algorithms follow a similar filter-verification approach, which is illustrated in Fig. 1. 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.

Input data and prefix index. The input consists of two collections of sets, R and S . Following previous work [8, 21], the set elements, called *tokens* [2], 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.

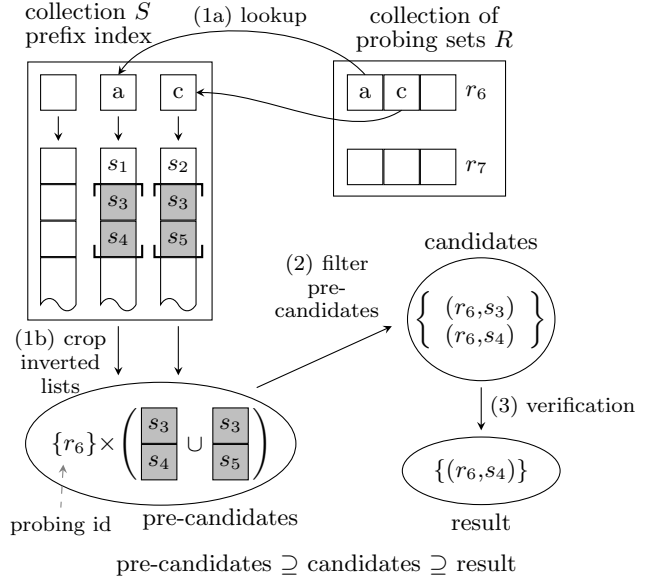


Figure 1: Set similarity join with prefix index.

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. Fig. 1):

(1) *Pre-candidate generation.* For each token in the prefix of r , the lookup (1a) returns an inverted list. The inverted lists are cropped (1b) using lower and upper bounds that are derived from the size of the probing set or the sort position of the probing token. 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 Fig. 1).

(2) *Candidate generation.* The pre-candidates are deduplicated and filtered. Pre-candidates that pass all filters are *candidates*. During the evaluation of the filter conditions, some algorithms remove obsolete entries from the inverted lists, 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 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.

Self Join. In a self-join, where $R = S$, the result is symmetric, i.e., $(r, s) \in R \bowtie S \Rightarrow (s, r) \in R \bowtie S$. 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 shorter prefixes to be used in the index [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 (see Table 1).

Table 1: 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. The **length** filter is used to crop the inverted lists. Based on the length of a probing set r , an upper and lower bound on the length of eligible index sets s_i is computed. Assume, for example, Jaccard threshold $t_J = 0.8$ and a probing set of length $|r| = 10$. Due to the definition of Jaccard similarity ($J(r, s) = |r \cap s| / |r \cup s|$), only sets of size $8 \leq |s| \leq 12$ can reach the threshold.

PPJoin (PPJ). PPJ [20, 21] extends ALL by applying the **positional** filter to the pre-candidates. The positional filter reasons over the matching position of tokens in the prefix. For example, the pair (r, s) in Fig. 2 is a pre-candidate since it passes prefix and length filter (the prefix is shaded). The Jaccard threshold ($t_J = 0.8$) is translated into a *required overlap* ($t = 9$) for the given set pair. The first match in s is on position 2, thus only 8 tokens of s are left to match a token in r , and the pre-candidate is rejected.

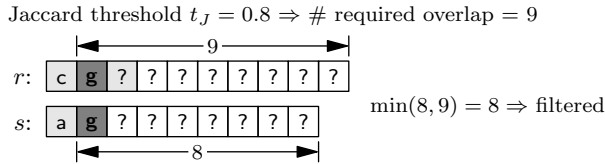


Figure 2: Positional filter.

PPJoin+ (PP+). PP+ [20, 21] extends PPJ with the **suffix** filter, which recursively partitions pre-candidate pairs to tighten the upper bound on matching tokens. Consider Fig. 3: 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 (**p**) is the pivot element. 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. Similar, there are at most 3 matches in the right partition. Overall, there can be at most 6 matches 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.

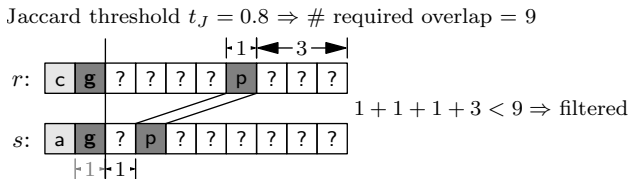


Figure 3: Suffix filter.

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 required overlap in the positional filter increases monotonically.

MPJoin-PEL (PEL). PEL [9] extends MPJ with the *position-enhanced length filter* (PEL) to crop the inverted lists in the index. PEL provides a tighter upper bound than the length filter, leading to a smaller number of pre-candidates.

AdaptJoin (ADP). ADP [19] generalizes the **prefix** filter. The prefix length is increased by e tokens, and $e + 1$ prefix matches are required for a pre-candidate pair. The prefix extension e is computed per probing set using a cost function. The prefix index is replaced by the adaptive prefix index (**adapt**), which supports longer prefixes dynamically.

3. IMPLEMENTATION NOTES

We have re-implemented all algorithms in C++ following the descriptions in the original papers. 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 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.

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

In the index, 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

²<http://www.cse.unsw.edu.au/~weiw/project/simjoin.html>

³<http://www.cs.berkeley.edu/~jnwang/projects/adapt>

⁴<http://dbgroup.cs.tsinghua.edu.cn/lgl/simjoin>

during pre-candidate generation. The position is not stored for ALL and ADP, which do not use the positional filter⁵.

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 reserve space and 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/PP/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. The precomputed overlap is accessed multiple times for each set s_i , and many sets may have the same length. 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 in the appendix). 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. ALL does not use the positional filter. Thus, we do not store the token position in the inverted lists, which reduces the length of the lists 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 if only a small number of pre-candidates is 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 data structure for the lists 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. Fig. 4). 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.

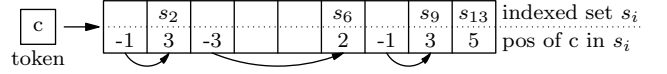


Figure 4: Inverted lists in MPJ and PEL.

ADP. The adaptive prefix index of ADP supports different prefix lengths. Instead of one inverted list per token, multiple inverted lists (one for each prefix length) are stored in the index. While ADP supports prefixes of arbitrary length, we observe that the cost function (which decides on the length of the prefix extension) rarely assumes values beyond 8. We limit the number of prefix lengths 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 2. 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 2: 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 (see Table 3):

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

⁵Wang et al. [19] discuss position-aware pruning for search queries only; an extension to joins (ADP) is not discussed.

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

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 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.

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 tokens from different distributions and randomly assigned them to the sets until the precomputed set size (which follows a Poisson distribution) 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 -grams¹⁴; however, due to many duplicate q -grams per set, the deduplicated number of tokens is 6864.

The data sets have different characteristics (see Table 3). 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 numbers of different tokens. Most datasets, like FLICKR (cf. Fig. 5), show a Zipf-like distribution. These datasets 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.

⁷BMS-POS: <http://www.sigkdd.org/kdd-cup-2000-online-retailer-website-clickstream-analysis>, provided by Blue Martini Software for KDD 2000 cup [22]

⁸DBLP: <http://www.informatik.uni-trier.de/~Ley/db> (Feb. 2014)

⁹ENRON: <http://www.cs.berkeley.edu/~jnwang/projects/adapt>, preprocessed by Wang et al. [19]

¹⁰KOSARAK: <http://fimi.ua.ac.be/data>

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

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

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

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

Table 3: Characteristics of real world datasets.

	#sets in coll.	set size		# diff. tokens
		max	avg	
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	$1.0 \cdot 10^5$

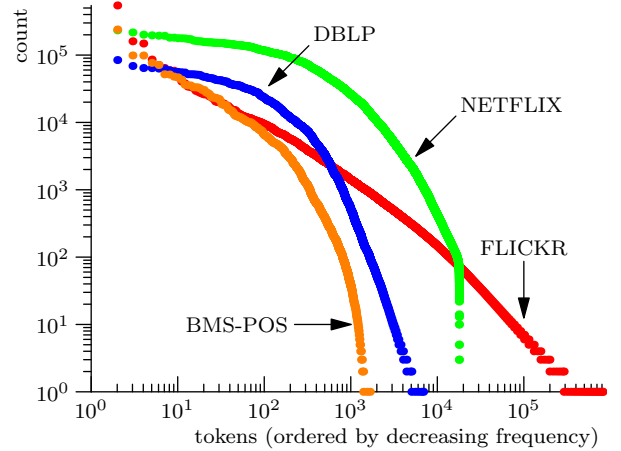


Figure 5: Histogram of token frequencies.

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 4 shows the fastest algorithm and its runtime (in seconds) for each data point; we also list the runners-ups that are within 10% from the fastest runtime. ALL wins on most data points (31), followed by PPJ (24), 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 two identical runtimes within the resolution of the measurement method). 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 5). 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 6(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). PEL, which works best on nonself-joins [9], performs only slightly better than MPJ in our self-join scenario.

Threshold. In Table 7 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 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 8 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 6 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 6 illustrates that #lookups has a visible impact on the candidate time if #lookups \sim #pre. Fig. 6(a) shows #lookups and #pre for PPJ and PEL; all numbers are

Table 4: Fastest algorithms.

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

percentages of #pre for PPJ. Fig. 6(b) shows the respective candidate times (percentage of PPJ candidate time).

For threshold $t_J = 0.7$, #lookups \ll #pre (by three orders of magnitude): 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.

Pre-Candidates and Candidates. In Fig. 7, we analyze the impact of pre-candidates and candidates on the run-

Table 5: Gap factor per dataset and threshold.

Dataset	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 6: 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.

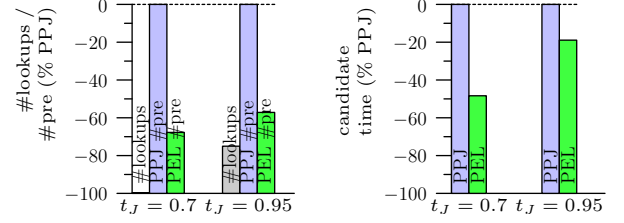
	Algorithm						
	ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
average	1.24	1.11	1.97	1.33	1.26	2.04	1.13
median	1.19	1.03	1.69	1.26	1.12	1.87	1.11
maximum	2.08	2.50	4.05	2.95	2.75	6.28	1.71

Table 7: Winner per threshold.

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 8: Slowest algorithms.

Dataset	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	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
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.

Figure 6: Effect of lookups and pre-candidates on candidate time (NETFLIX).

time. 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. Fig. 7(a)). ALL deduplicates the pre-candidates in a single 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. Fig. 7(b)), but compared to PPJ, the candidate set is larger. PPJ additionally applies the positional filter (cf. Fig. 7(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. Fig. 7(d)).

MPJ and PEL reduce the number of pre-candidates: MPJ removes hopeless pre-candidates (inverted list entries) from the index, PEL in addition leverages the matching position to better crop the lists. The overhead for these filters must be outweighed by a smaller number of pre-candidates. 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%); the pre-candidate reduction of MPJ is not large enough to outperform PPJ in terms of runtime. On AOL, MPJ and PEL reduce #pre by less than 1% and cannot outweigh the filter overhead. 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 1). 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 set and the last matching token in the other

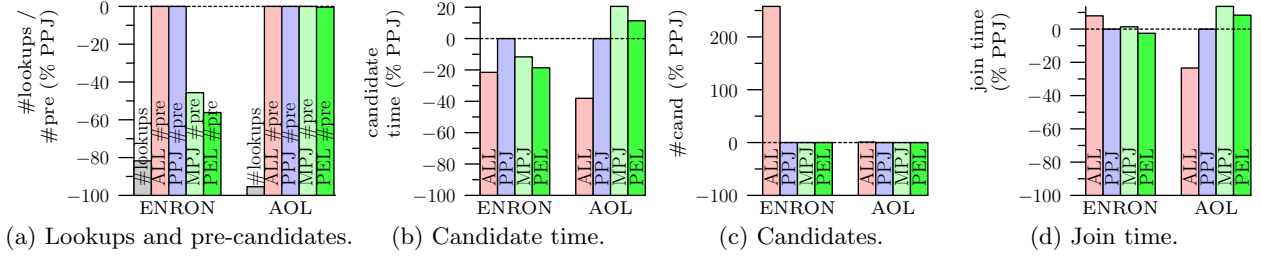


Figure 7: Pre-candidates and candidates, $t_J = 0.8$.

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 9. 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 Fig. 8: there are not enough tokens left in r to satisfy the required overlap (p_r and p_s are circled).

Table 9: Avg. number of token comparisons, PPJ.

Dataset	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

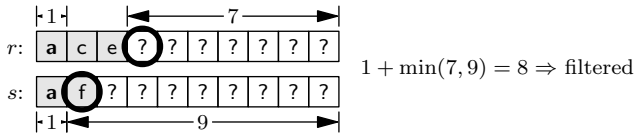


Figure 8: Verification with zero token comparisons.

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 9. ALL often requires slightly less 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, 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 access 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. Fig. 9 shows the slowdown of PP+ w.r.t. the winner on all data points: PP+ never wins and cannot compete with the best algorithms.

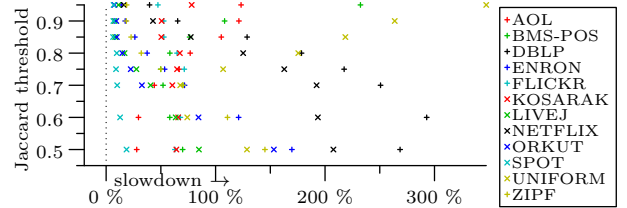


Figure 9: PP+ vs. competing algorithms.

We investigate the performance of the suffix filter vs. verification in Fig. 10, where we count the average number of CPU cycles for PPJ (only verification) and PPJ+ (suffix filter and verification).¹⁵ We analyze datasets with short (BMS-POS), medium (LIVEJ), and long sets (NETFLIX).

On BMS-POS and LIVEJ, we observe that PPJ is faster than PP+ in rejecting false positives during verification. PPJ must also verify the false positives that the suffix filter would reject, 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 be faster than verification on false positives. Thereby, we need to consider

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

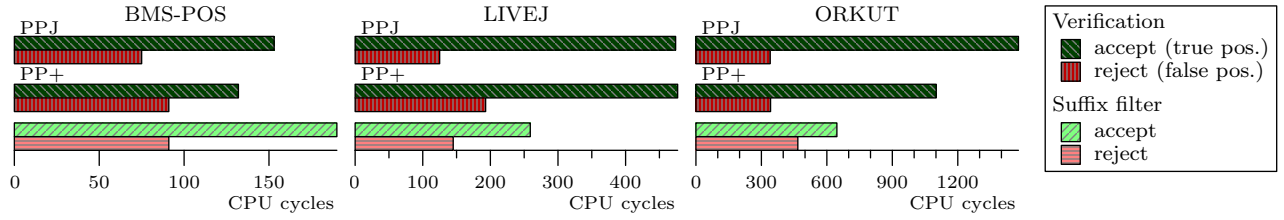


Figure 10: Cost of suffix filter and verification, average CPU cycle count per candidate, $t_J = 0.8$.

the verification time of PPJ. Figure 10 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. The overhead is significant: the suffix filter is slower on candidates that pass the filter since it stops as soon as a candidate is rejected.

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 Fig. 11 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: 5.8s for LIVEJ (ALL: 2.2s), 0.75s on NETFLIX (ALL: 0.35s).

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. Fig. 5), 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 prefix during candidate generation. During verification the groups are expanded since identical prefixes may stem from non-identical sets. Grouping leads to fewer lookups and pre-candidates. In the absence of duplicate prefixes, GRP behaves like PPJ.

Figure 12(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 7: 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 Fig. 12(a). The unfolding overhead is partially absorbed by better memory locality, which is the predominant effect in Figure 12(b).

In Fig 12(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 input collections are ordered by length, and sets of the same length 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 13 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 prefix length: 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 shuf-

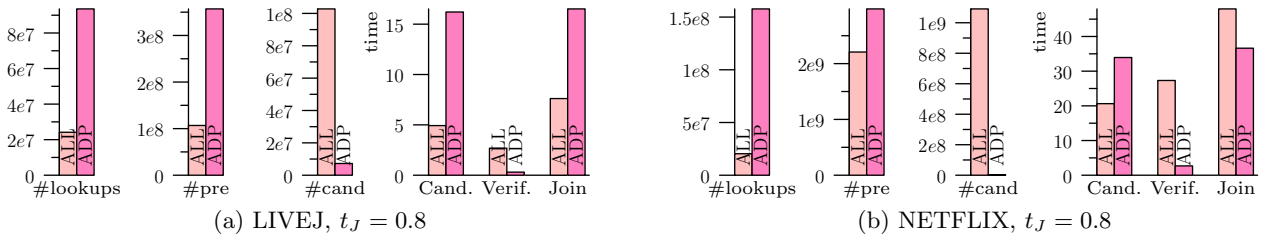


Figure 11: ADP vs. ALL: Candidate generation, verification, and join time.

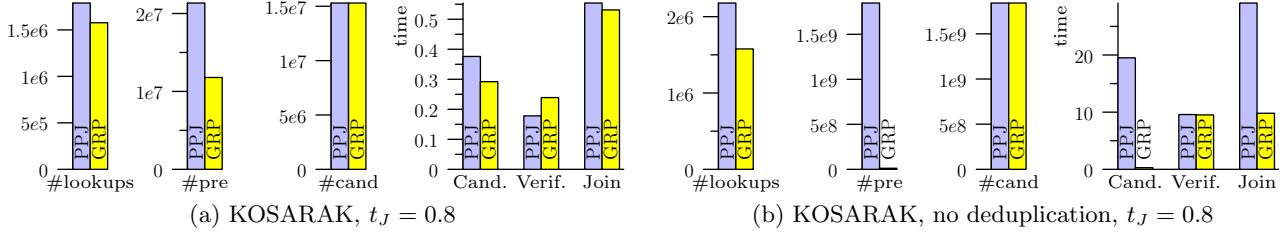


Figure 12: GRP vs. PPJ: Candidate generation, verification, and join time.

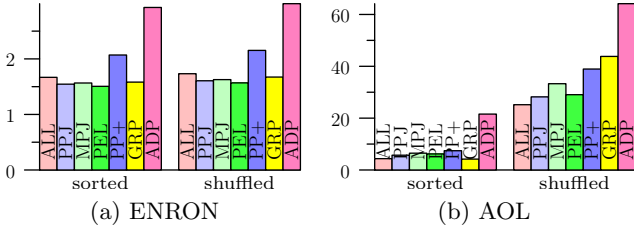


Figure 13: Runtime, $t_J = 0.8$.

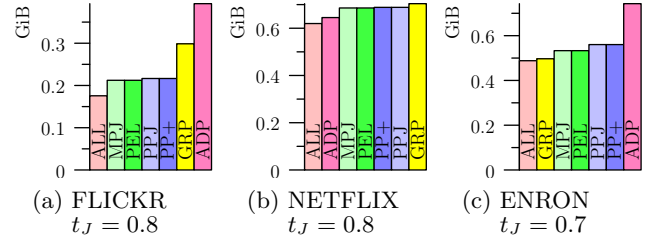


Figure 14: Memory usage (in bytes).

fled 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+PPJ, AOL, $t_J = 0.75$).

6. MEMORY USAGE

We study the memory usage¹⁷ of all join algorithms in Fig. 14. 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 14(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 14(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 14(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).

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 reproduce 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 ENRON¹⁸, which compares the runtimes of PPJ, PP+, ALL, and LSH-95% (approximate algorithm, not covered in this paper). Figure 15(a) shows the plot of

¹⁶Cache misses counted with Linux `perf` tools.

¹⁷Heap memory measured with Linux `memusage`.

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

the original paper [21], our results are shown in Fig. 15(b). The absolute runtimes differ, which is expected since we use a faster processor. In addition, in our experiment (1) PP+ is slower than the 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 Fig. 15(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 Fig. 15(d), where we substitute the verification routine in the original source code with the verification that we use in our experiments (Algorithm 1).

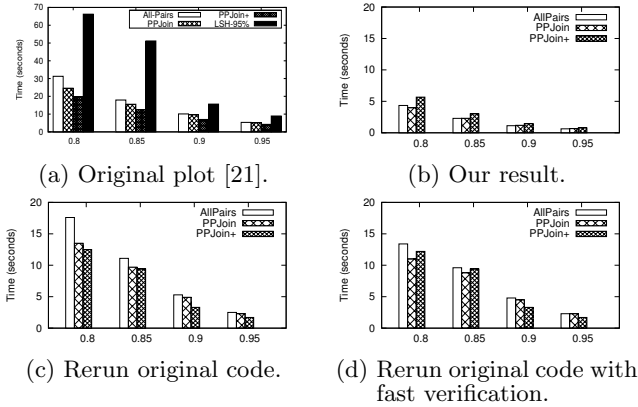


Figure 15: Reproducing results of [21]: PPJ, PP+, and ALL on ENRON (with duplicates), Jaccard.

7.2 MPJoin

Ribeiro and Härder [15] present MPJ. We repeat a self-join experiment on DBLP (q -grams, $q \in \{2, 3, 4\}$, $t_J = 0.75$), with PPJ, HYBRID (MPJ with suffix filter), and PP+. The original results are shown in Fig. 16(a), Fig. 16(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

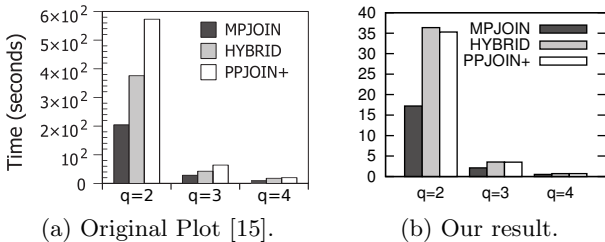


Figure 16: Reproducing results of [15]: MPJ, HYBRID, and PP+ on DBLP (q -grams), Jaccard.

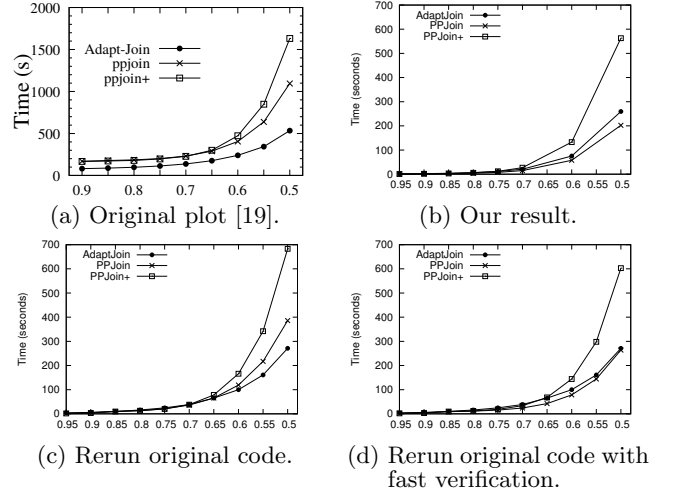


Figure 17: Reproducing results of [19]: ADP, PPJ, and PP+ on ENRON (with duplicates), Jaccard.

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 17(a) and 17(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 18(c) shows the plot for the original code (pre-processing offset removed), in Fig. 17(d) we apply efficient verification. We use the following setting: ADP runs with the binaries provided by the original authors¹⁹, the other algorithms run with the original PPJ/PP+ code [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%/ $t_J = 0.95$ to -96%/ $t_J = 0.6$ 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 Fig. 18. In our experiment, PPJ consistently outperforms ADP. We proceed as in Section 7.1. In Fig 18(c) we use the binaries provided by the authors [8]; in Fig 18(d) we use the original PPJ/PP+ code [21], but apply efficient verification.

¹⁹<http://www.cs.berkeley.edu/~jnwang/projects/adapt/>

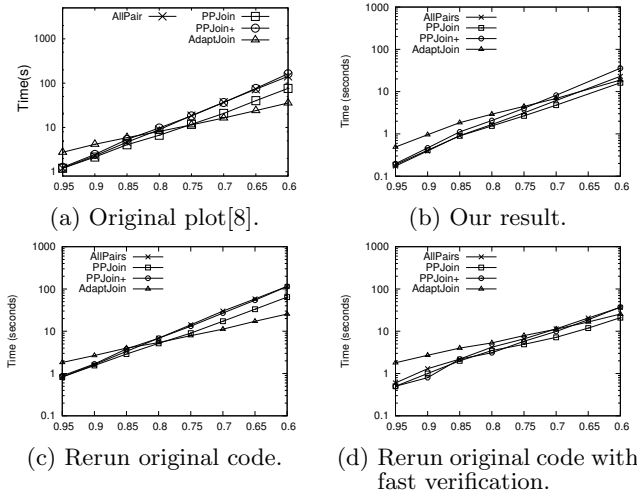


Figure 18: Reproducing results of [8]: ALL, PPJ, PP+, and ADP on ENRON (dedup.), Jaccard.

8. CONCLUSIONS

We have studied seven algorithms for set similarity joins. We could show that efficient verification plays a key role and needs more attention. We further found that the most effective technique is the prefix filter, first implemented in AllPairs. Later improvements, in particular, PPJoin and GroupJoin, outperform AllPairs on average, but the gain is moderate. We believe that the framework of AllPairs is maxed out and we do not expect significant impact from future algorithms that extend this framework.

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), 2014.
- [9] W. Mann and N. Augsten. PEL: Position-enhanced length filter for set similarity joins. In *Proc. GvD (Foundations of Databases)*, 2014.
- [10] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. Technical report, University of Salzburg, Austria, 2015. <http://dbresearch.uni-salzburg.at/ssjoin>.
- [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. IMC 2007*, San Diego, CA, October 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, September 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*, 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.

APPENDIX

Algorithm 1: Verify($r, s, t, \text{olap}, p_r, p_s$)

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

```

A. RUNTIME

For reference, we add plots with all runtime results that we obtained for this paper.

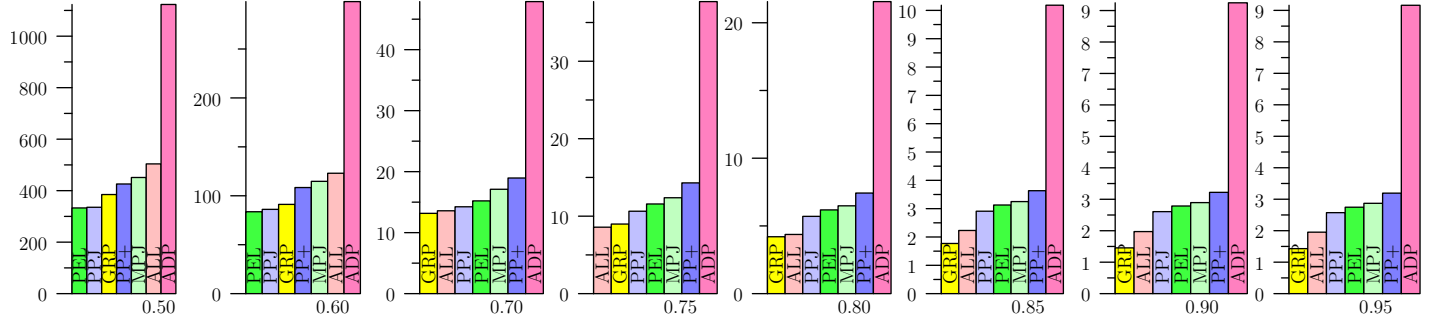


Figure 19: AOL, Jaccard

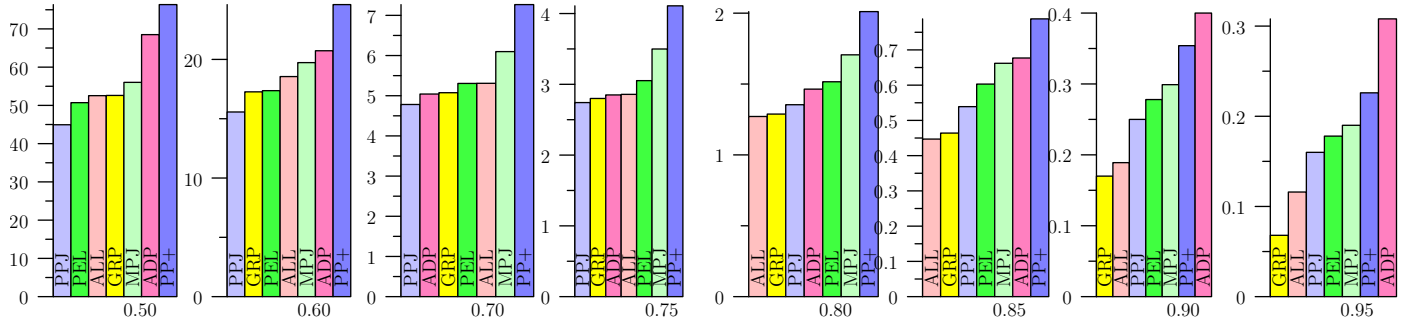


Figure 20: BMS-POS, Jaccard

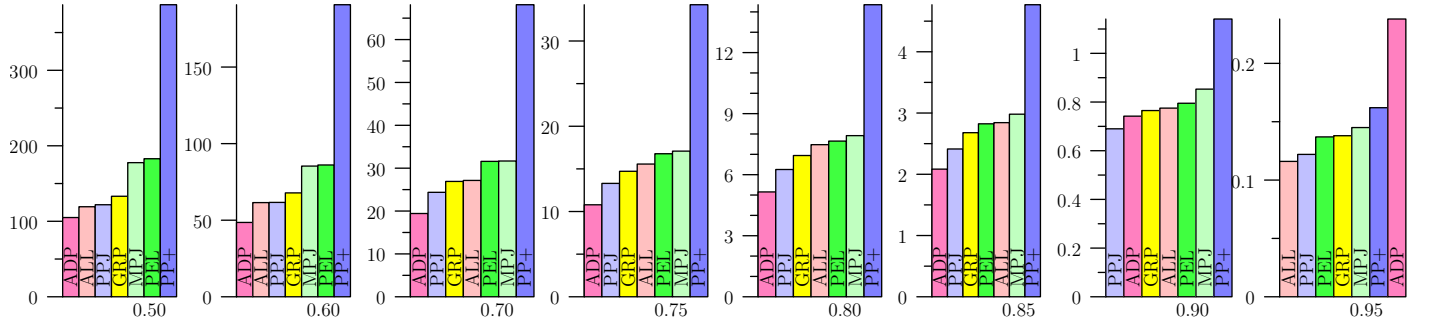


Figure 21: DBLP, Jaccard

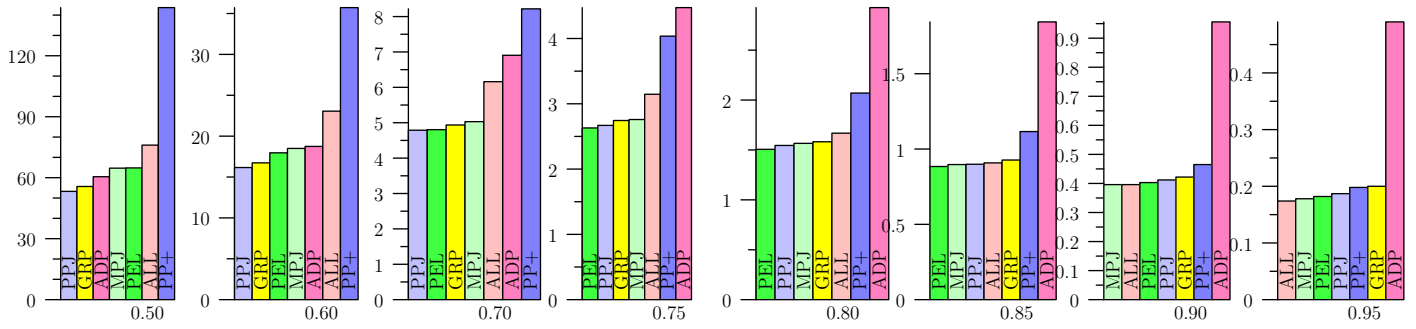


Figure 22: ENRON, Jaccard

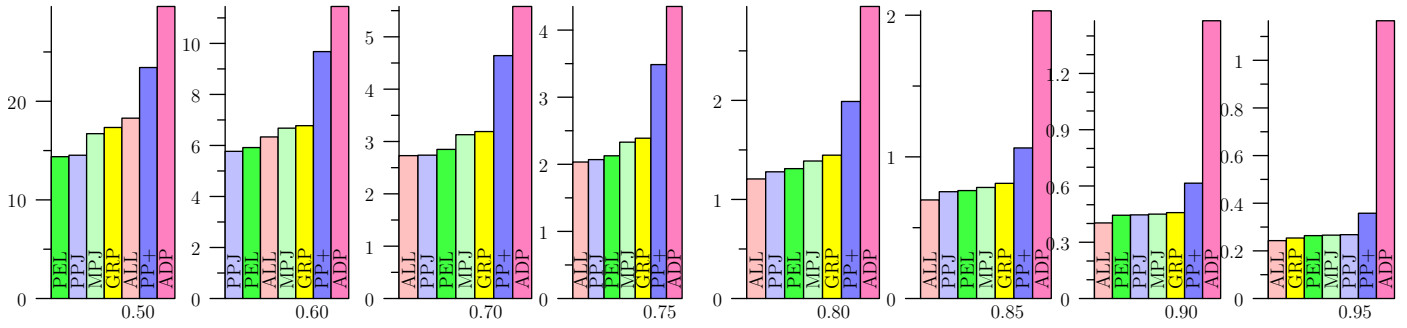


Figure 23: FLICKR, Jaccard

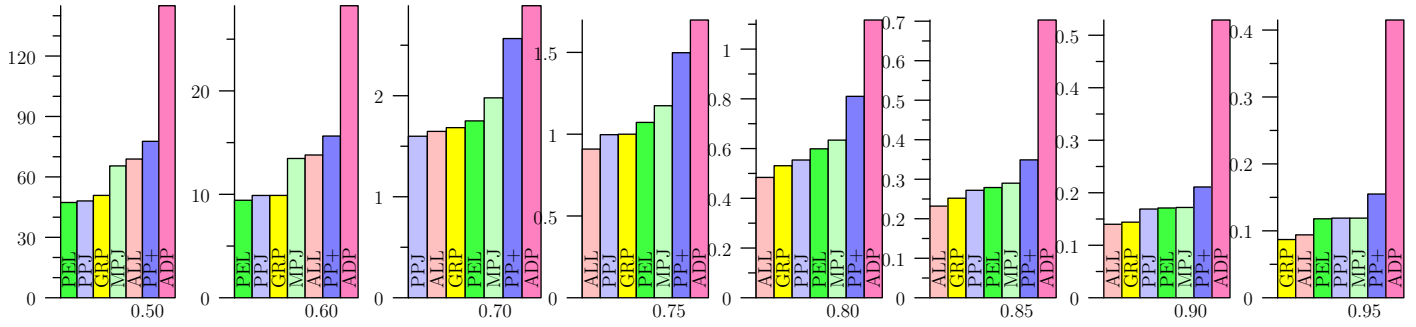


Figure 24: KOSARAK, Jaccard

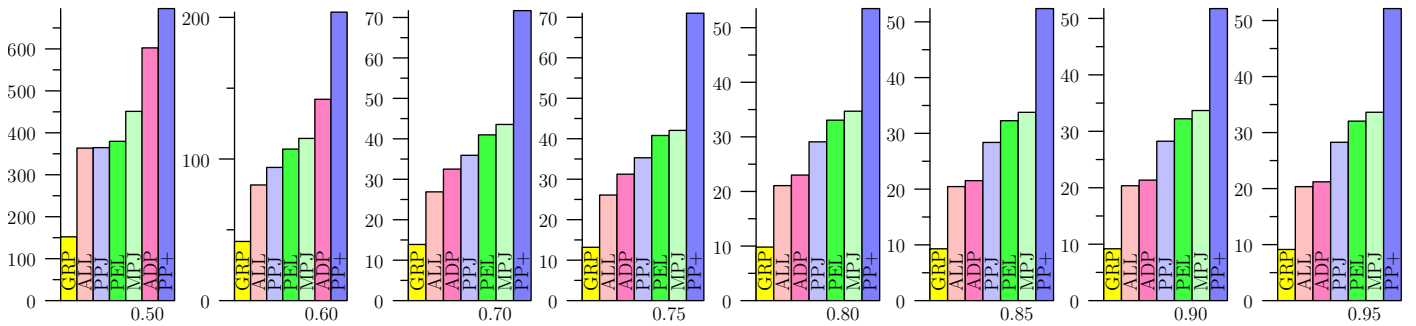


Figure 25: KOSARAK, no deduplication, Jaccard

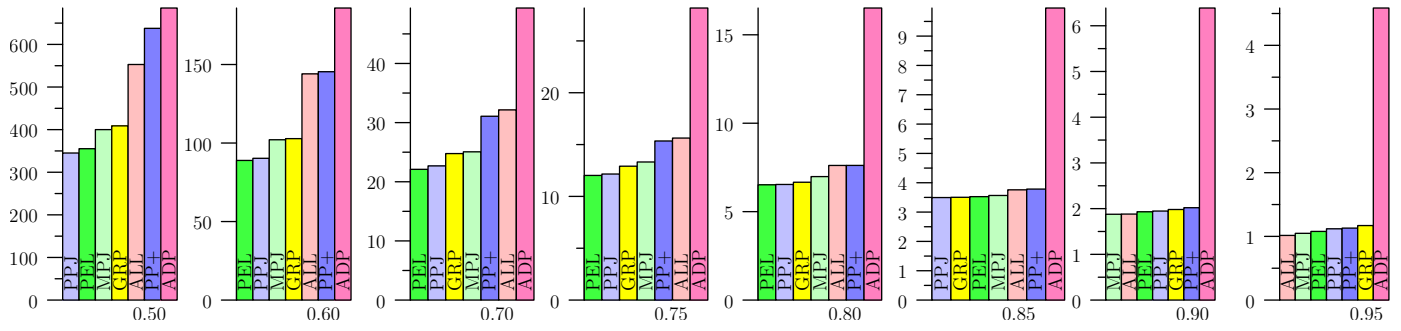


Figure 26: LIVEJ, Jaccard

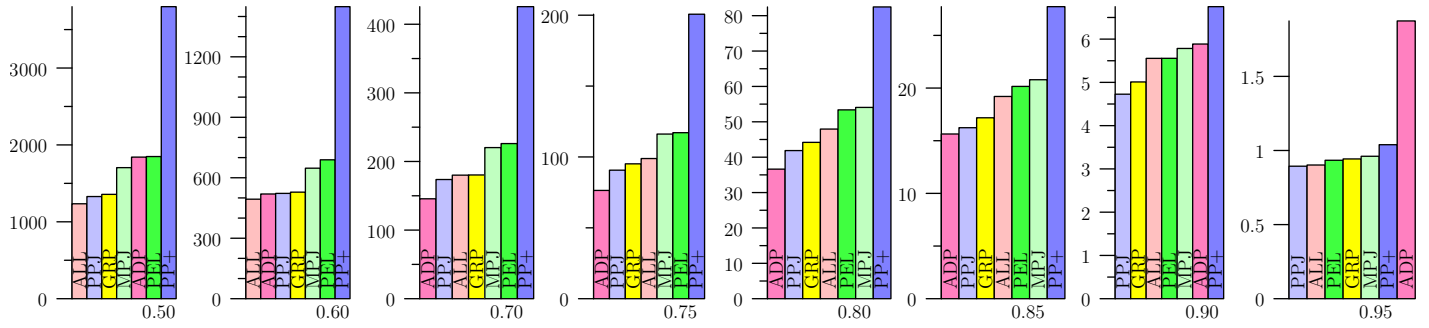


Figure 27: NETFLIX, Jaccard

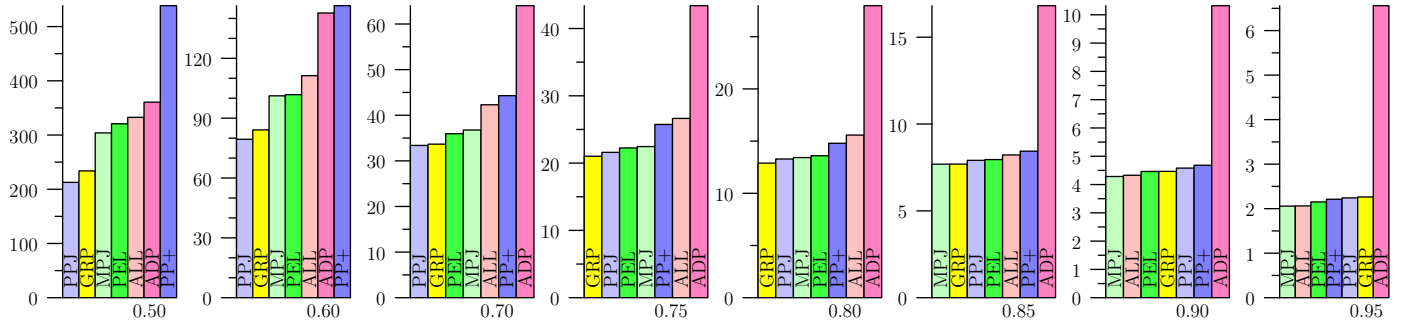


Figure 28: ORKUT, Jaccard

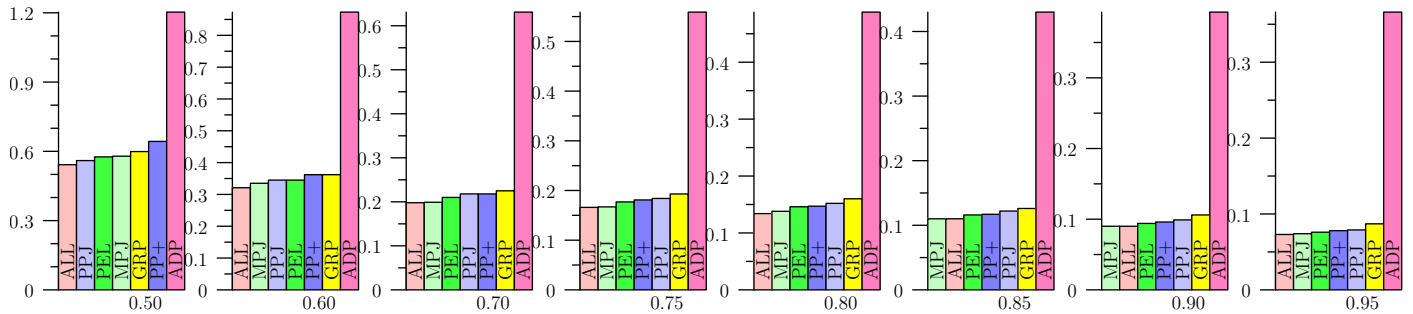


Figure 29: SPOT, Jaccard

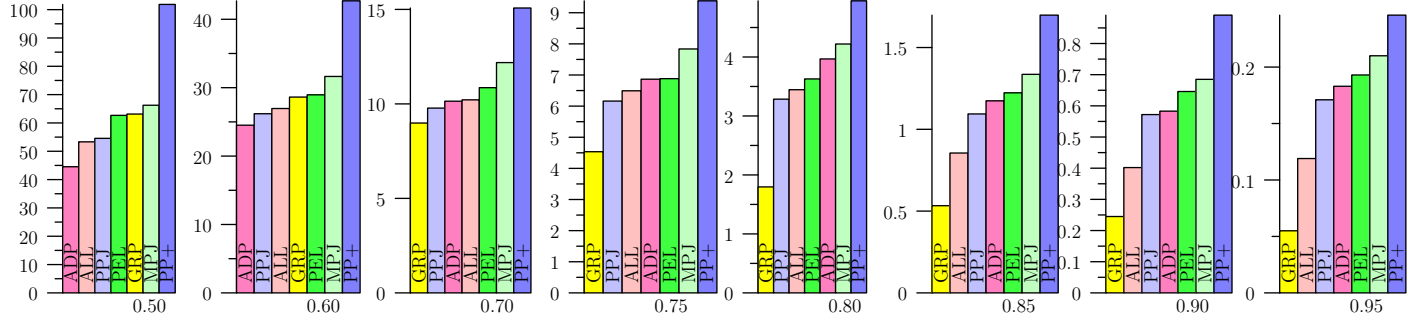


Figure 30: UNIFORM, Jaccard

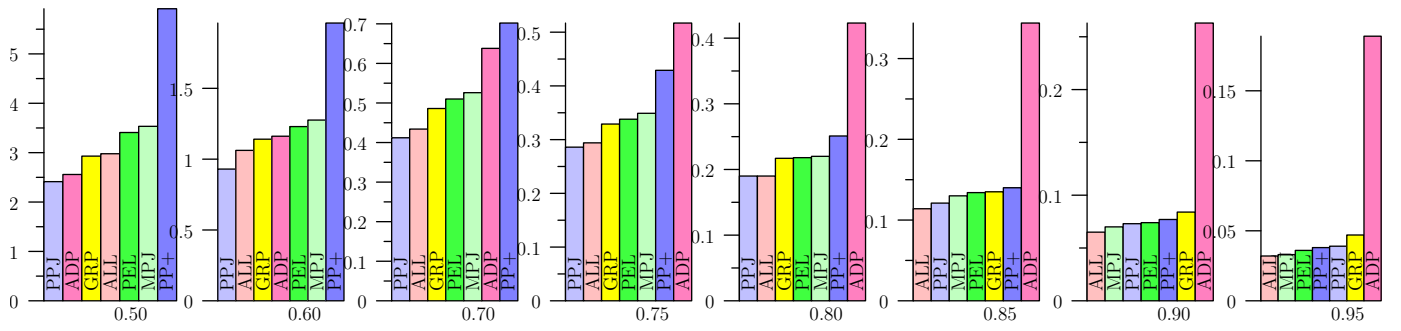


Figure 31: ZIPE, Jaccard

B. CANDIDATE GENERATION TIME

For reference, we add plots with all candidate generation time results that we obtained for this paper.

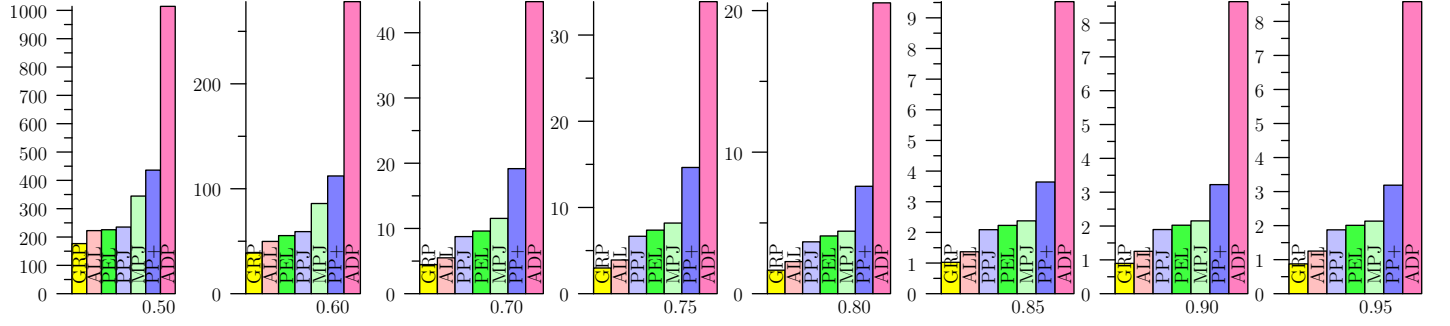


Figure 32: AOL, Jaccard

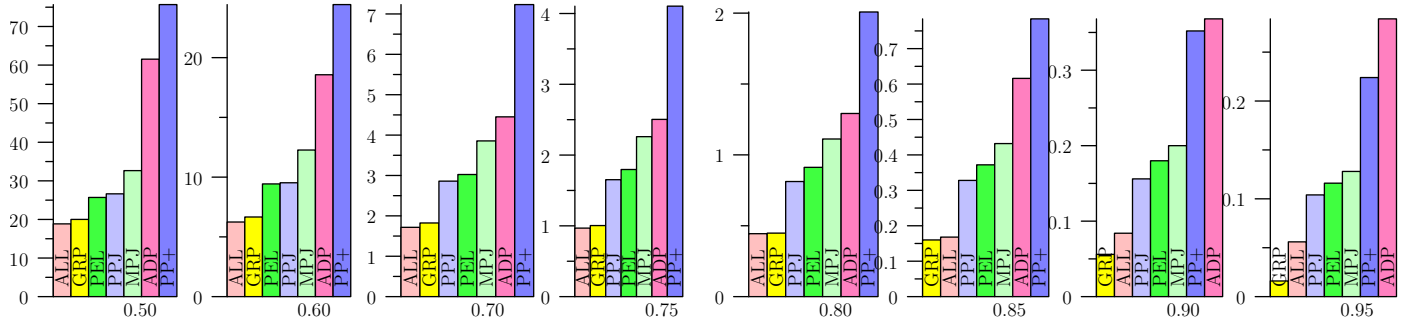


Figure 33: BMS-POS, Jaccard

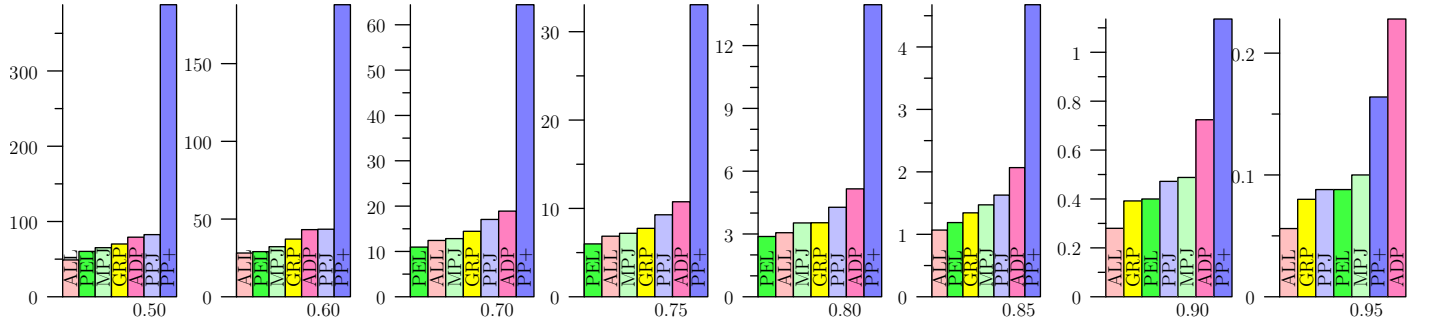


Figure 34: DBLP, Jaccard

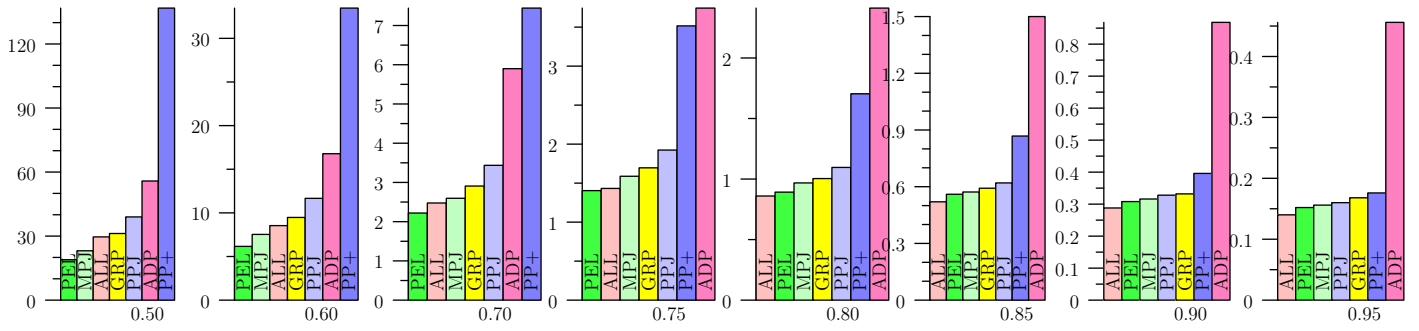


Figure 35: ENRON, Jaccard

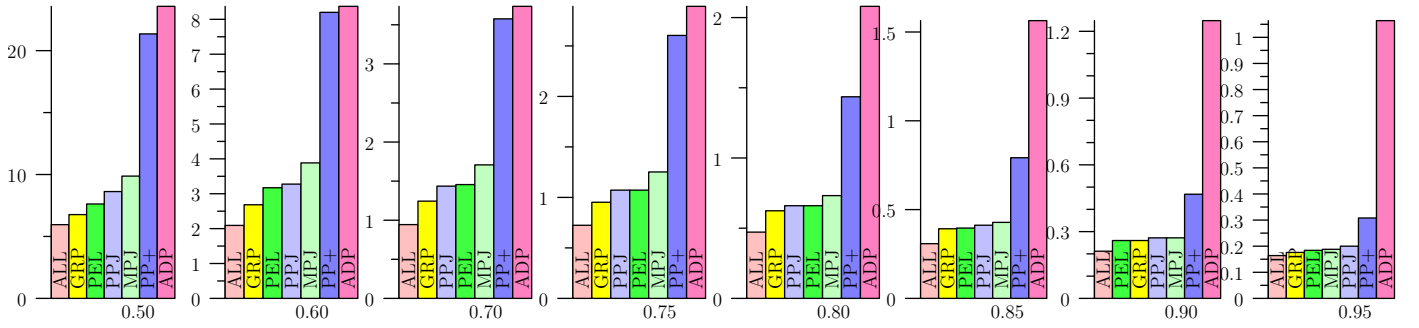


Figure 36: FLICKR, Jaccard

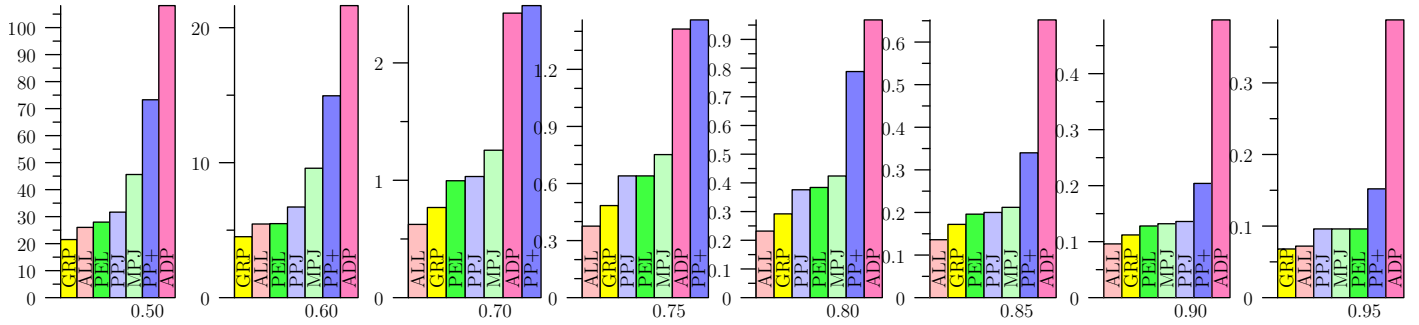


Figure 37: KOSARAK, Jaccard

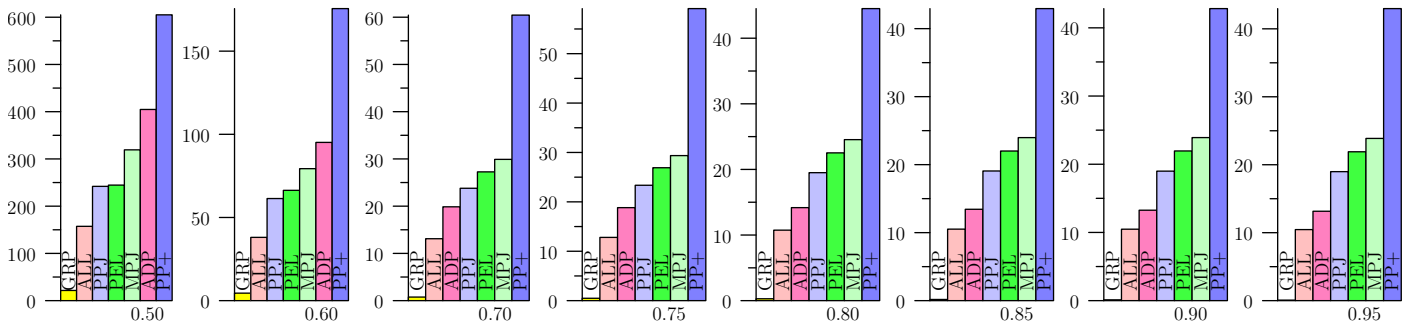


Figure 38: KOSARAK, no deduplication, Jaccard

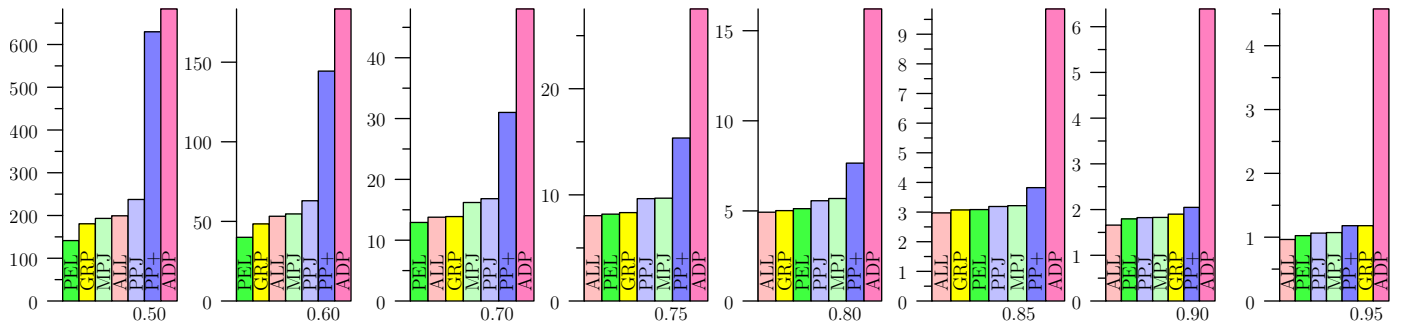


Figure 39: LIVEJ, Jaccard

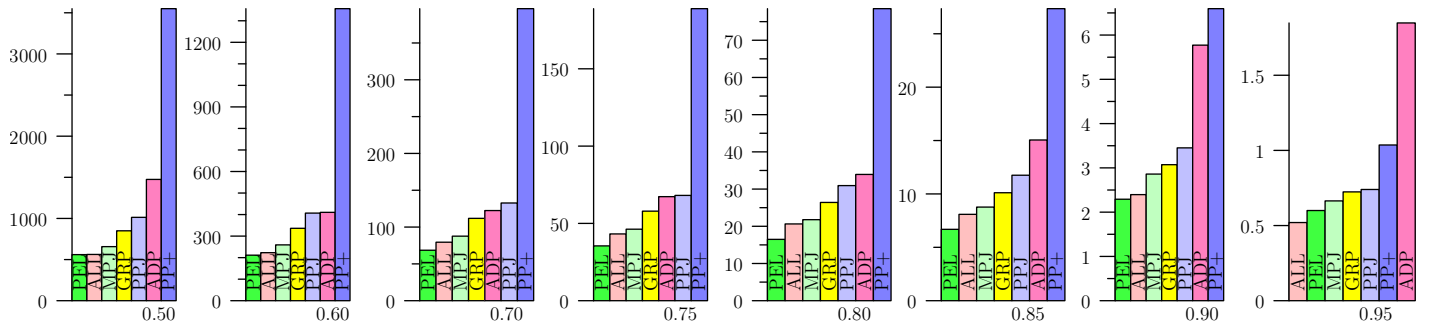


Figure 40: NETFLIX, Jaccard

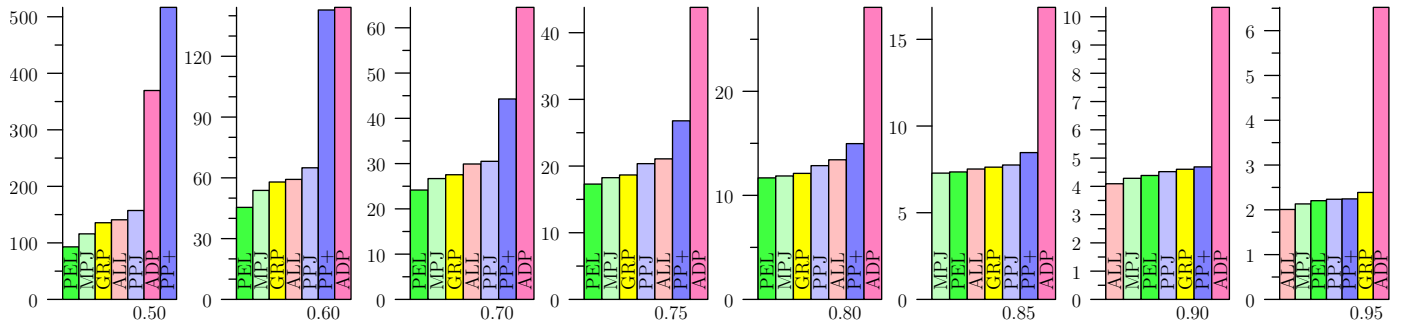


Figure 41: ORKUT, Jaccard

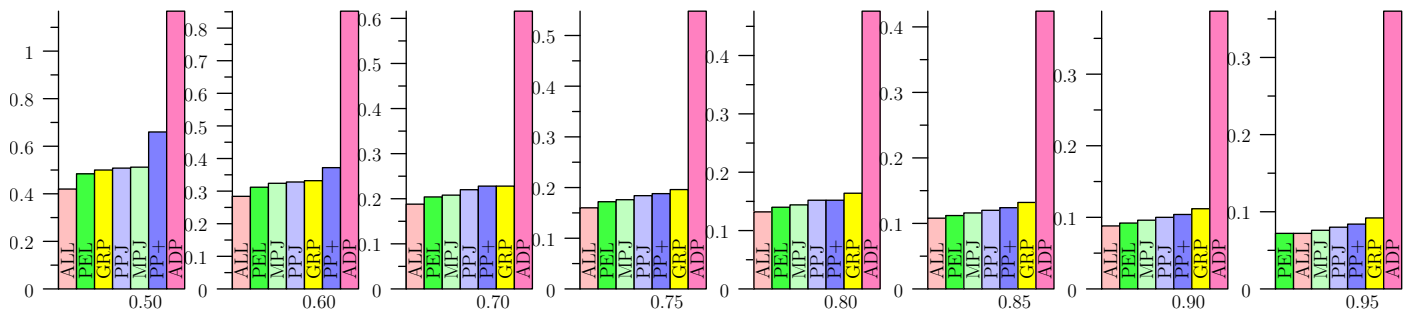


Figure 42: SPOT, Jaccard

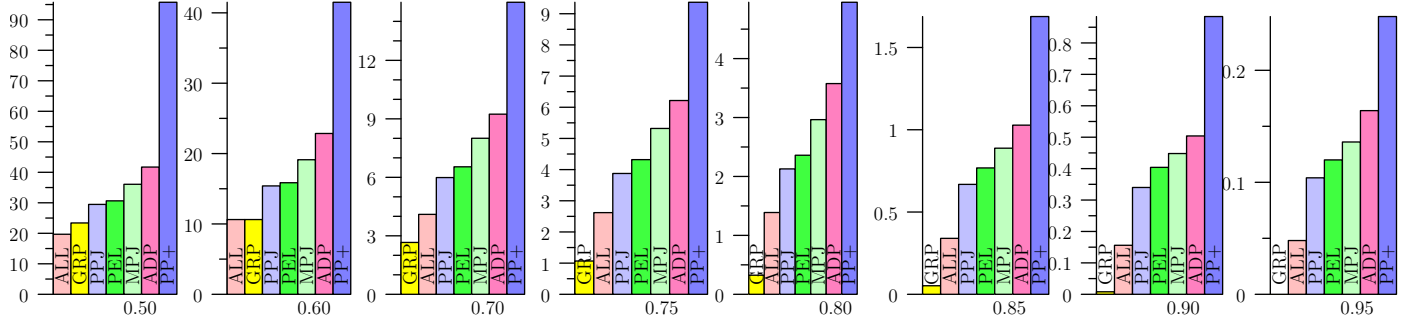


Figure 43: UNIFORM, Jaccard

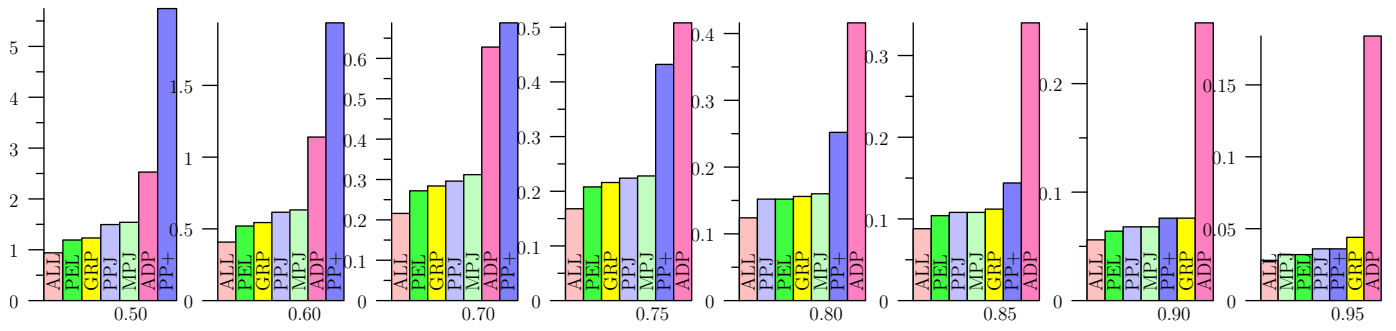


Figure 44: ZIPE, Jaccard

C. LOOKUPS

We add plots with all lookups measured for the algorithms.

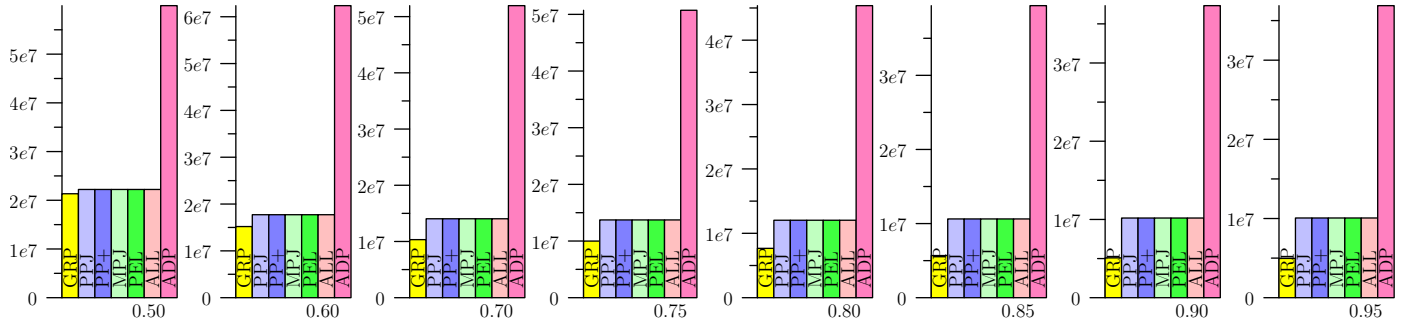


Figure 45: AOL, Jaccard

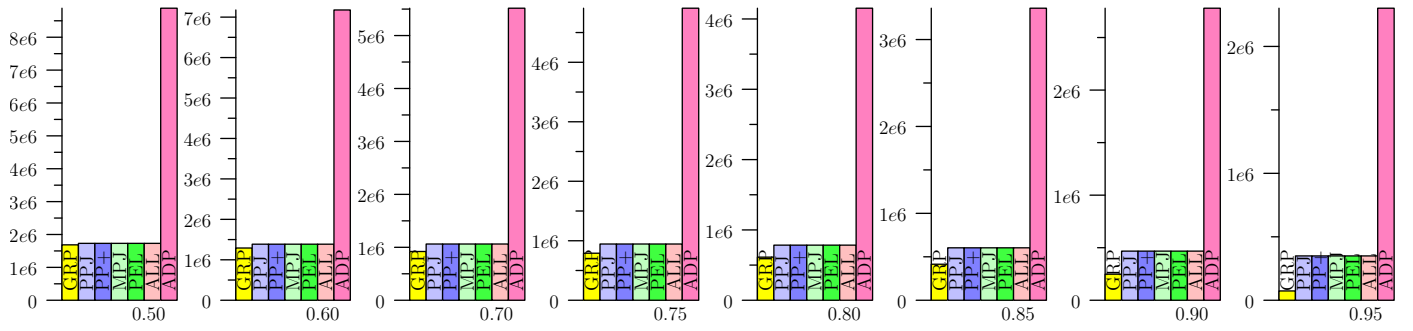


Figure 46: BMS-POS, Jaccard

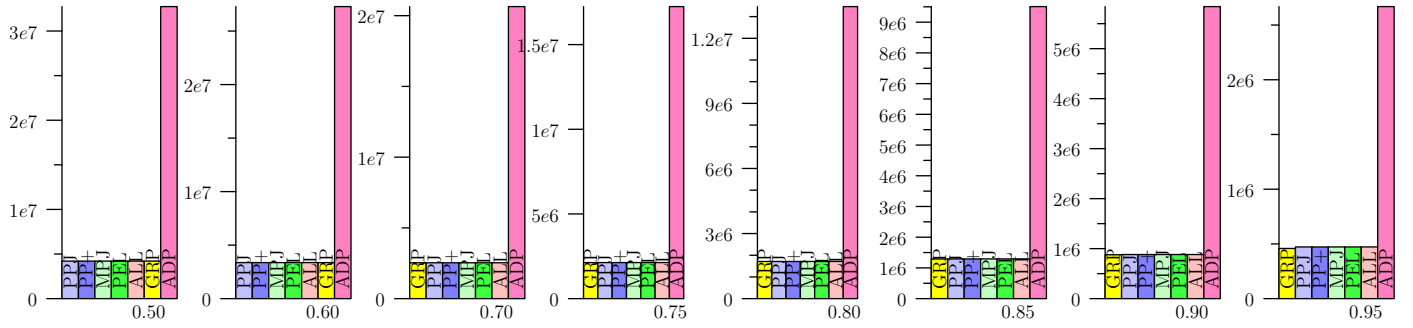


Figure 47: DBLP, Jaccard

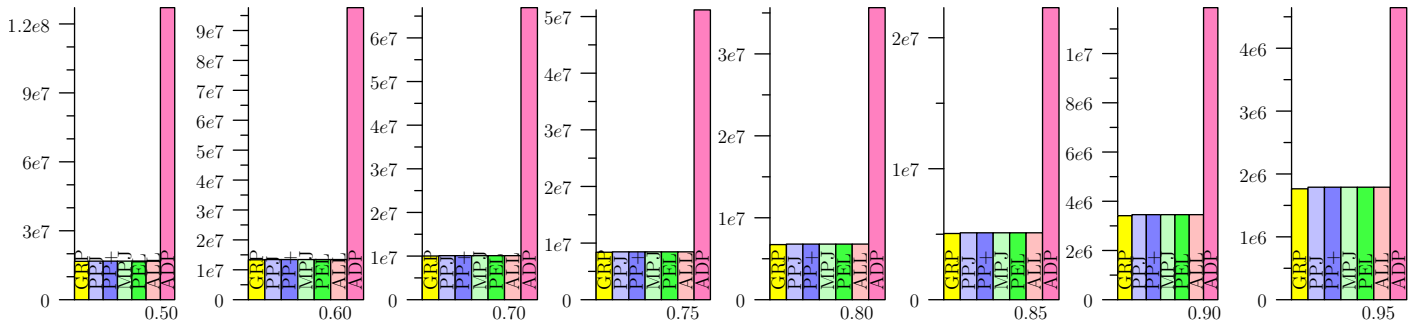


Figure 48: ENRON, Jaccard

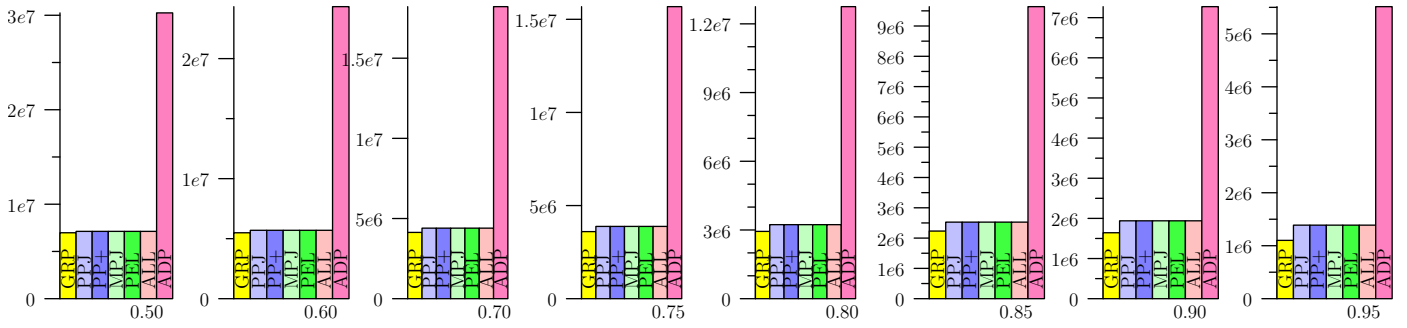


Figure 49: FLICKR, Jaccard

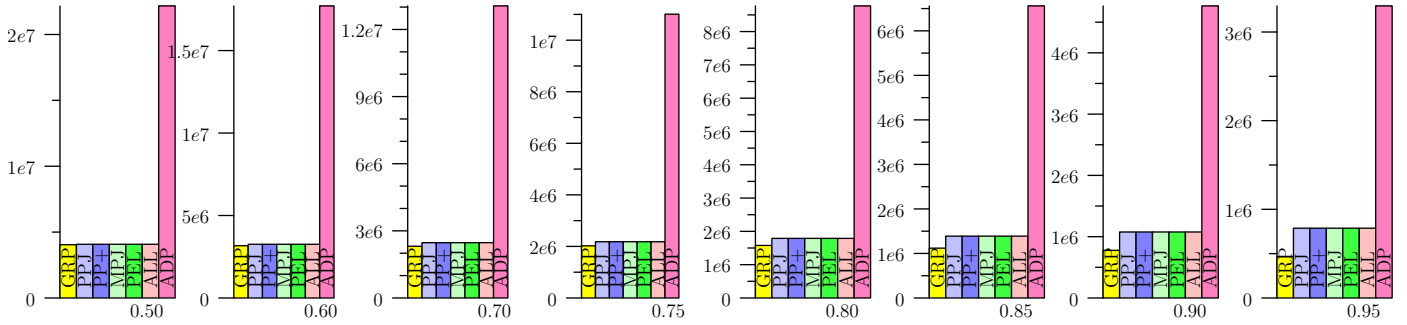


Figure 50: KOSARAK, Jaccard

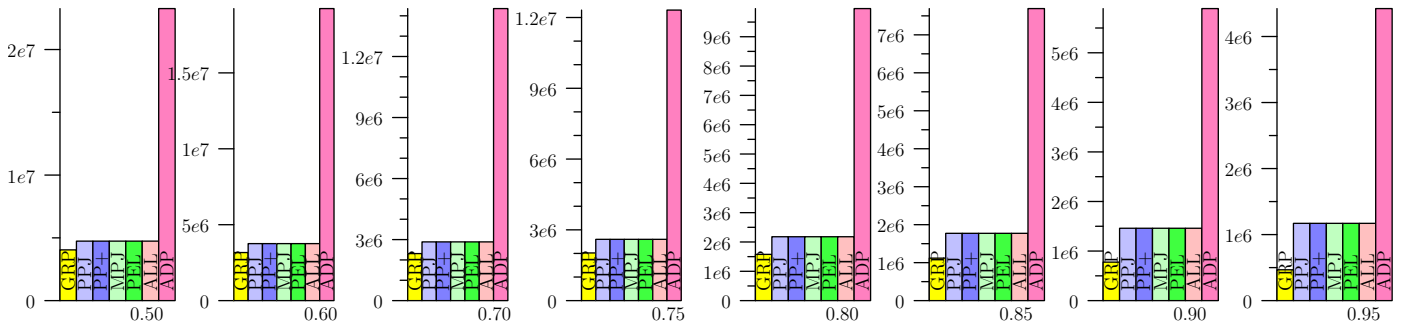


Figure 51: KOSARAK, no deduplication, Jaccard

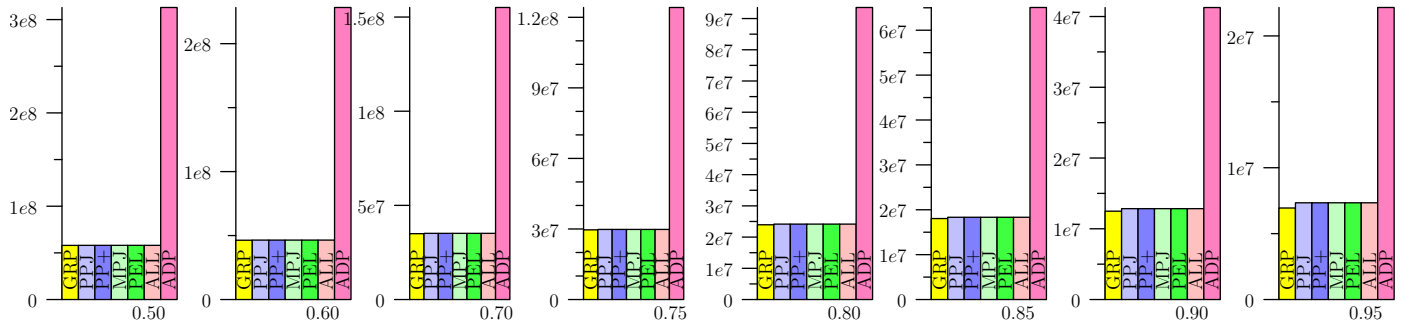


Figure 52: LIVEJ, Jaccard

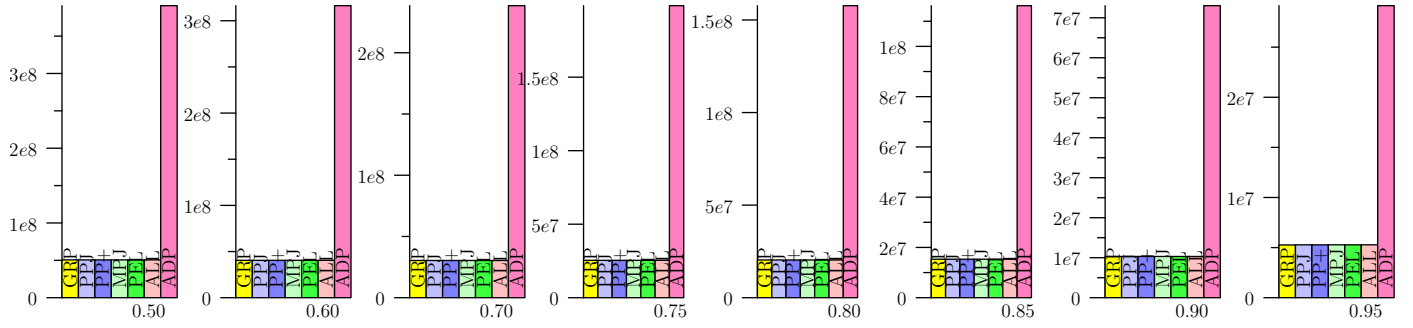


Figure 53: NETFLIX, Jaccard

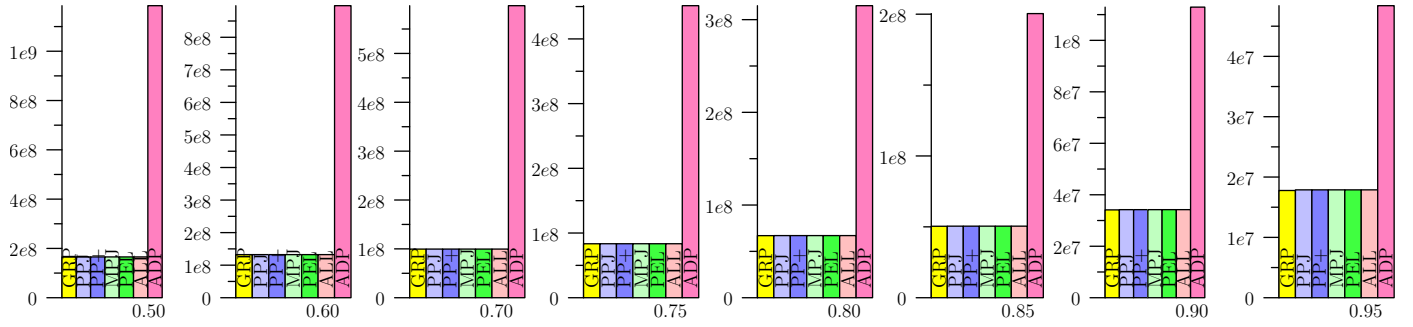


Figure 54: ORKUT, Jaccard

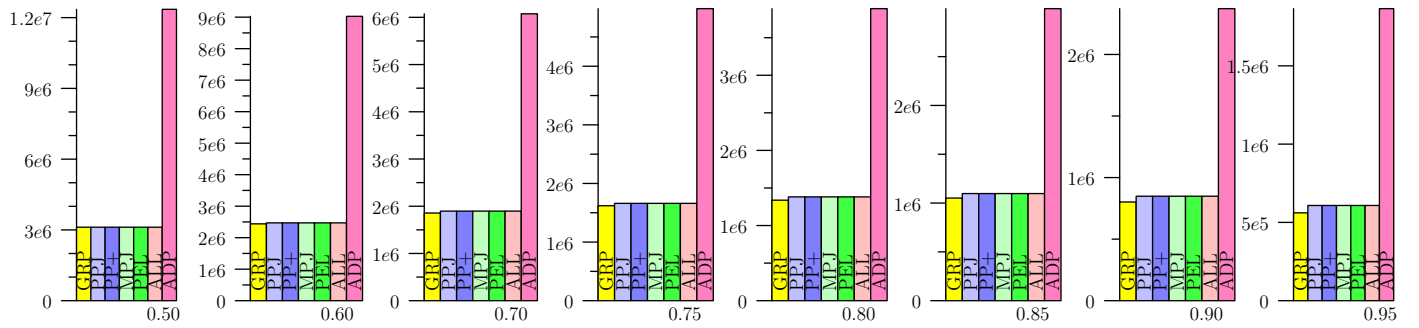


Figure 55: SPOT, Jaccard

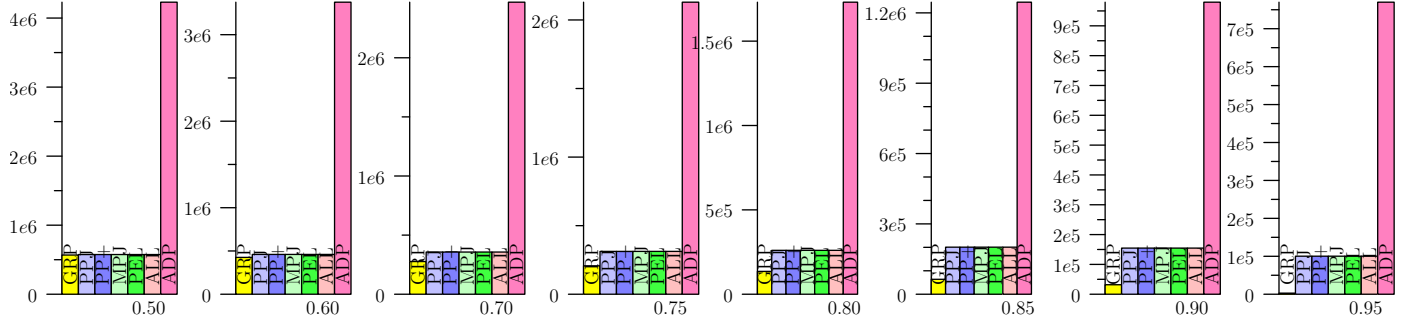


Figure 56: UNIFORM, Jaccard

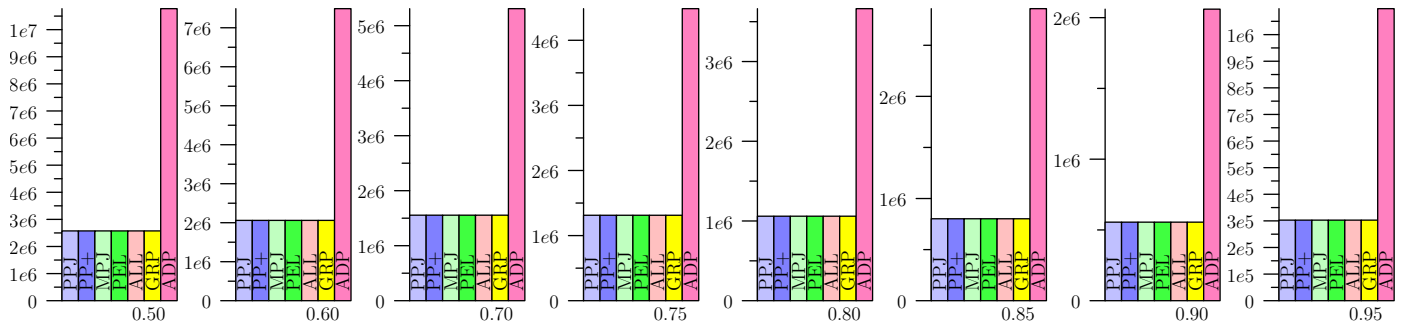


Figure 57: ZIPIF, Jaccard

D. PRE-CANDIDATES

We add plots with all pre-candidates measured for the algorithms.

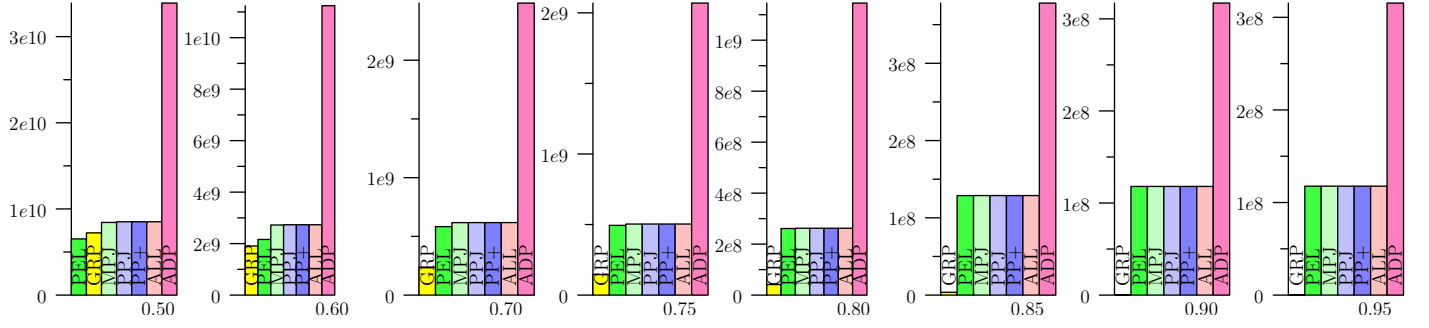


Figure 58: AOL, Jaccard

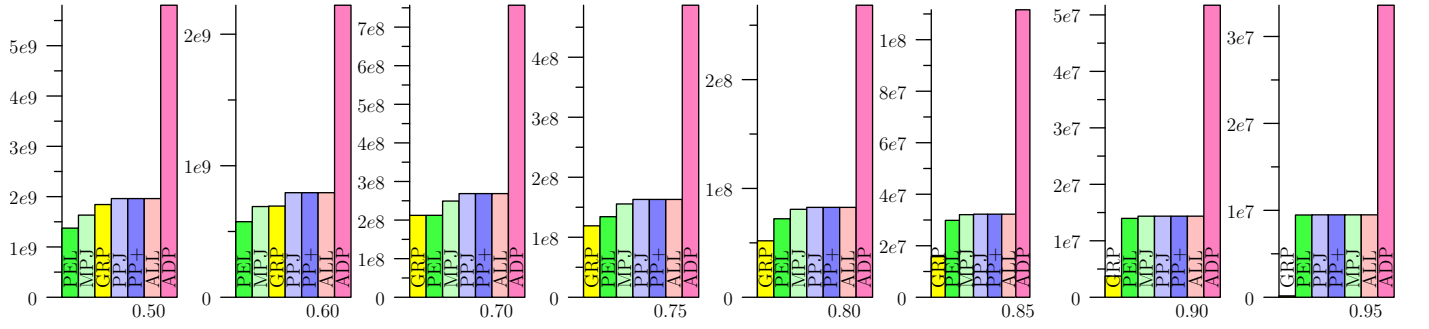


Figure 59: BMS-POS, Jaccard

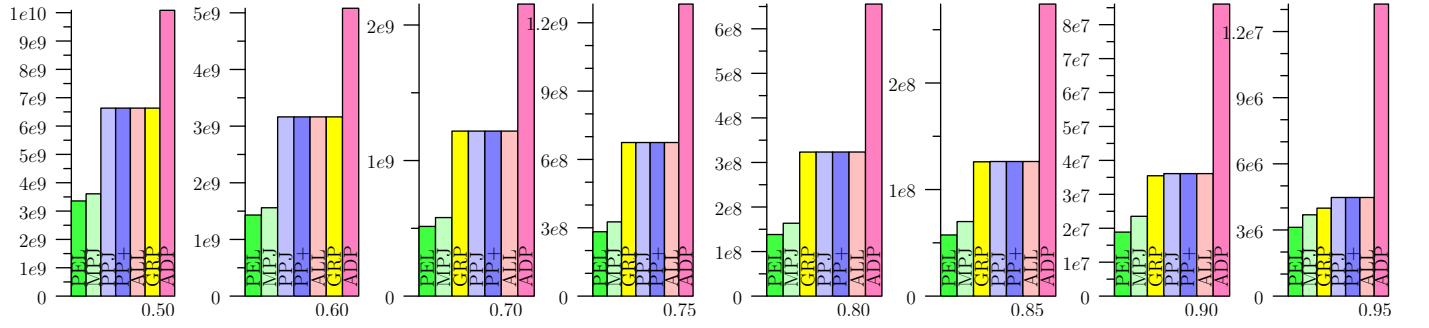


Figure 60: DBLP, Jaccard

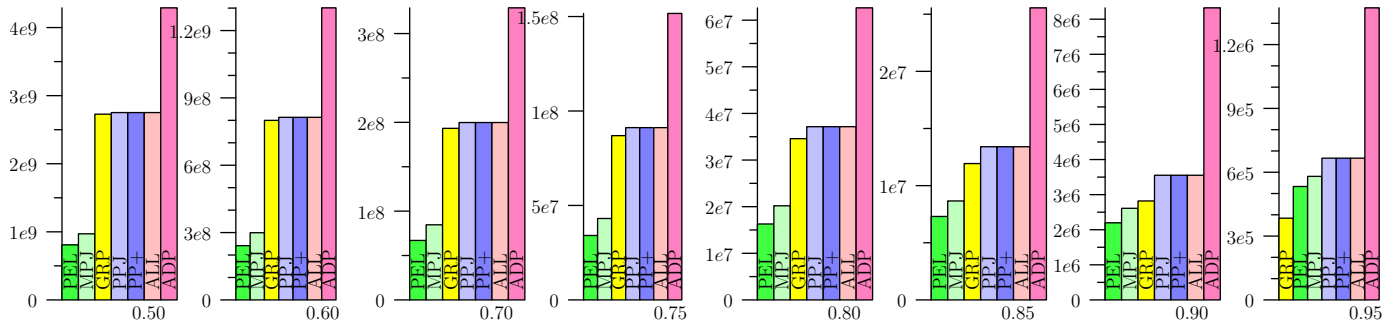


Figure 61: ENRON, Jaccard

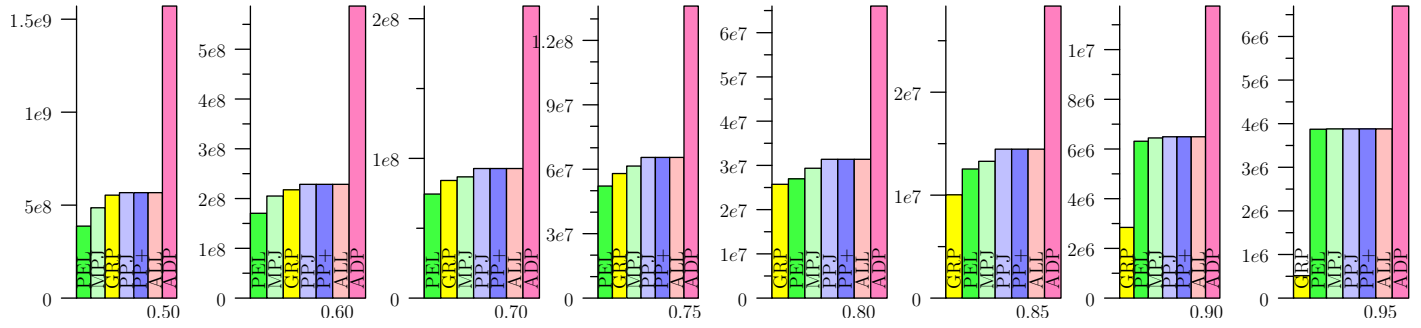


Figure 62: FLICKR, Jaccard

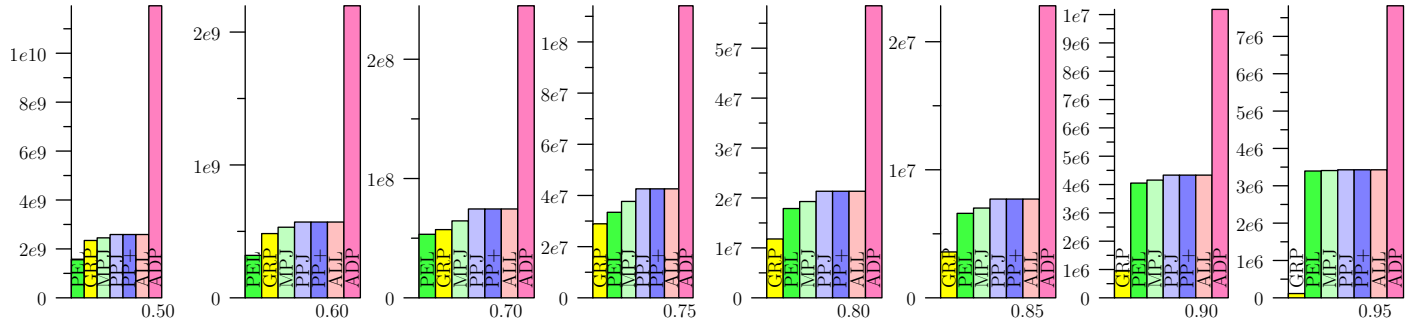


Figure 63: KOSARAK, Jaccard

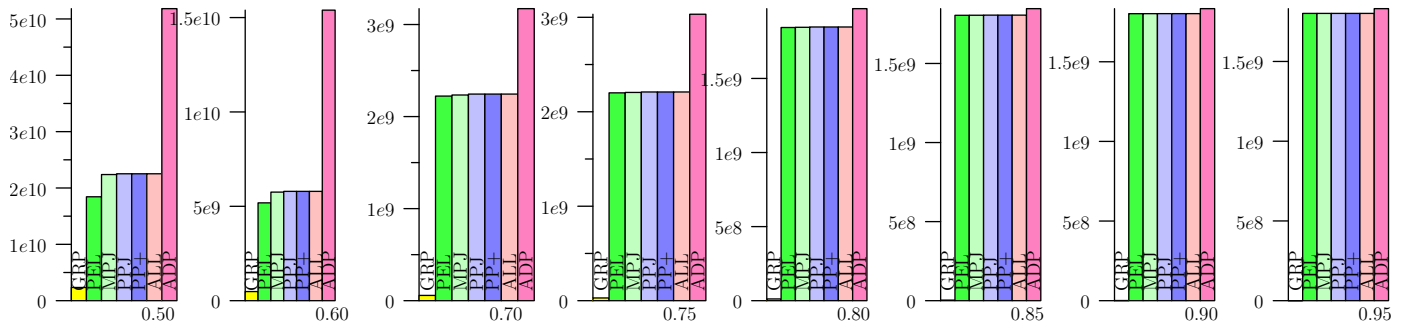


Figure 64: KOSARAK, no deduplication, Jaccard

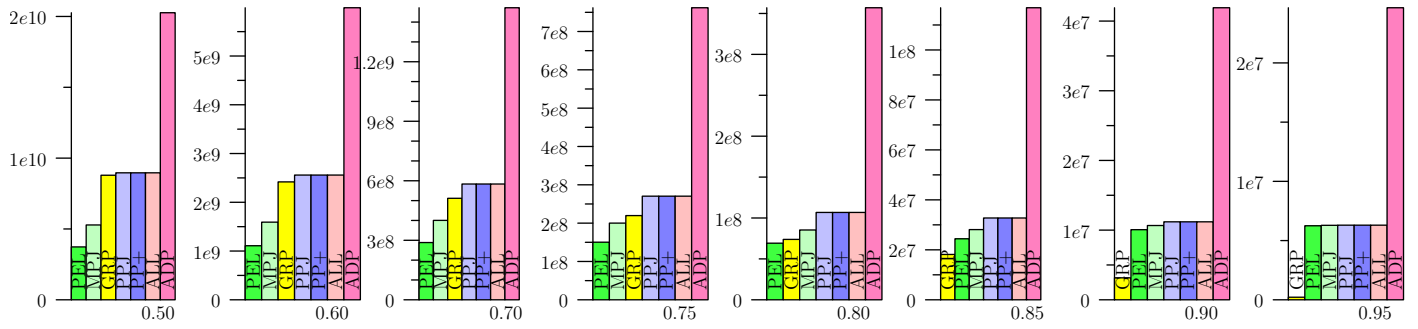


Figure 65: LIVEJ, Jaccard

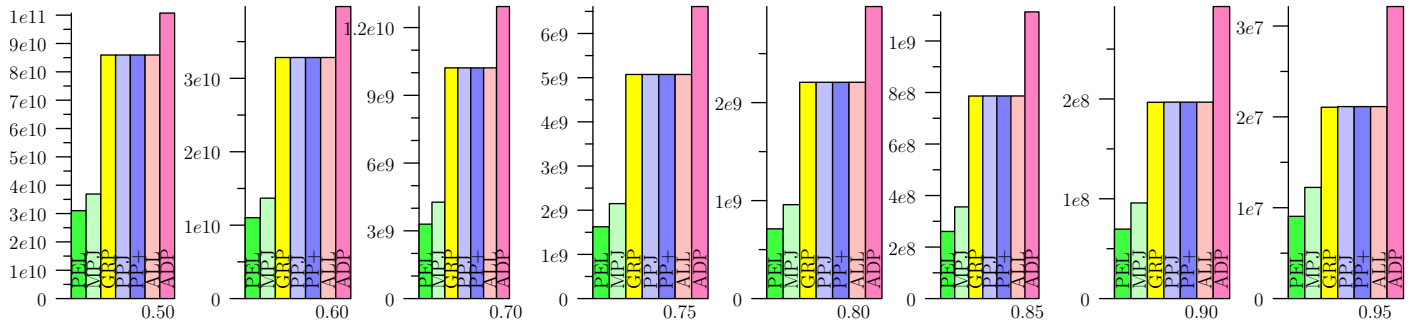


Figure 66: NETFLIX, Jaccard

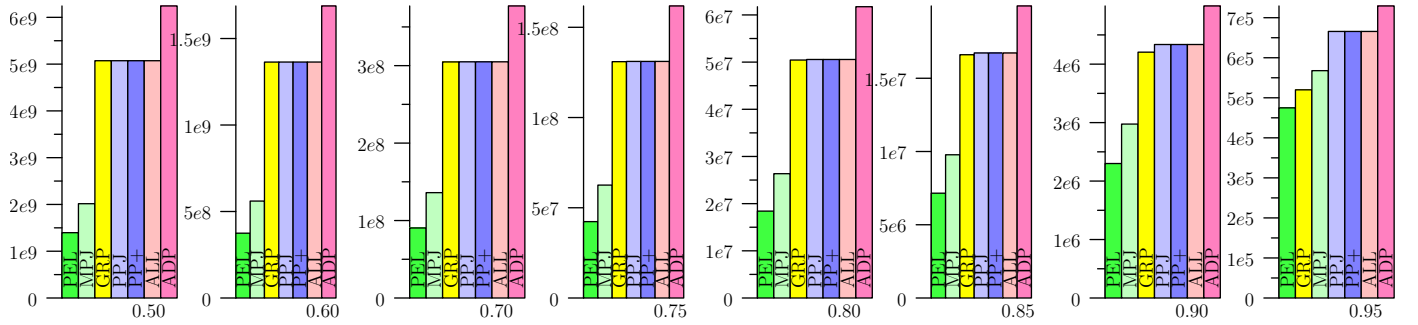


Figure 67: ORKUT, Jaccard

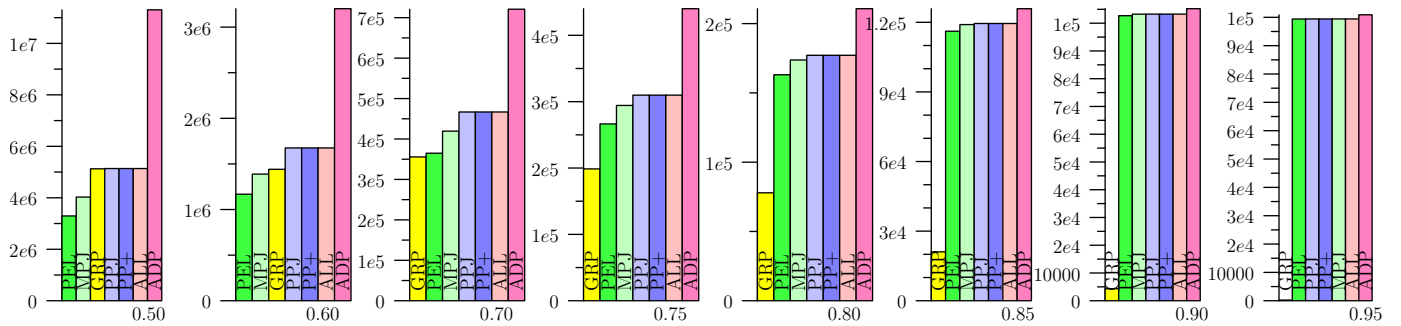


Figure 68: SPOT, Jaccard

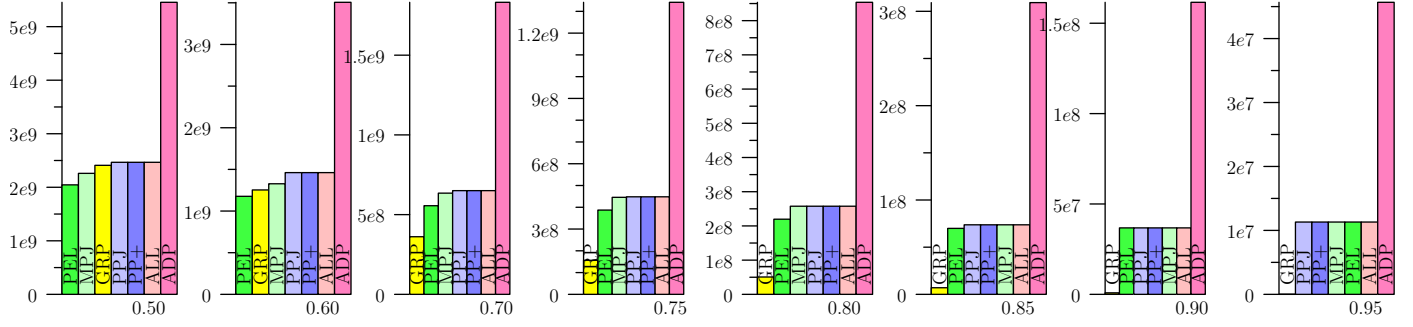


Figure 69: UNIFORM, Jaccard

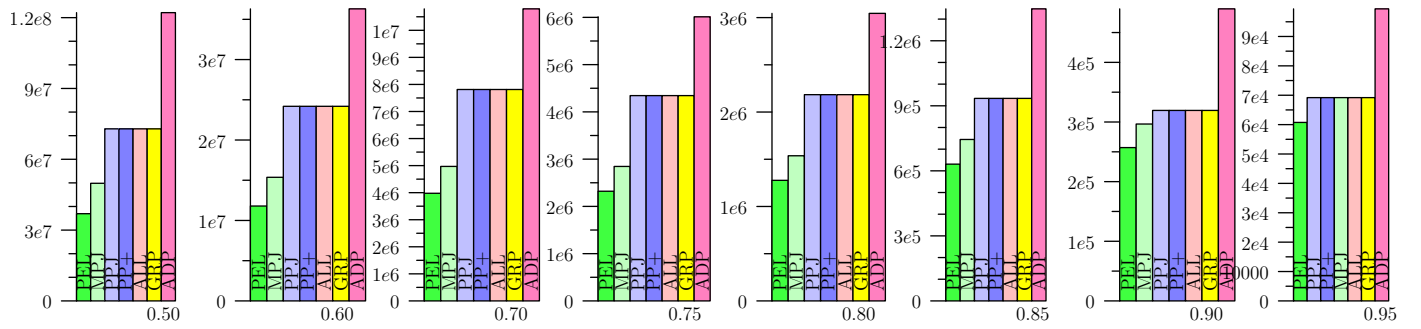


Figure 70: ZIPF, Jaccard

E. CANDIDATES

We add plots with all candidates measured for the algorithms.

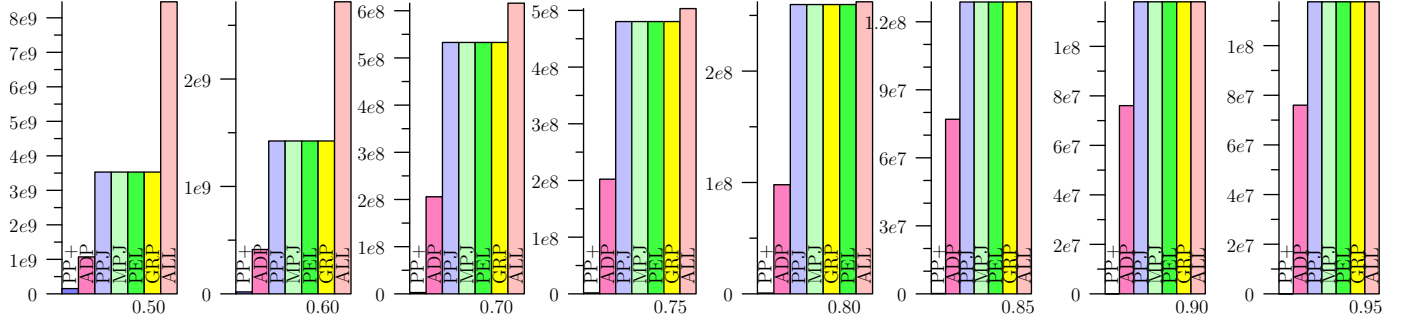


Figure 71: AOL, Jaccard

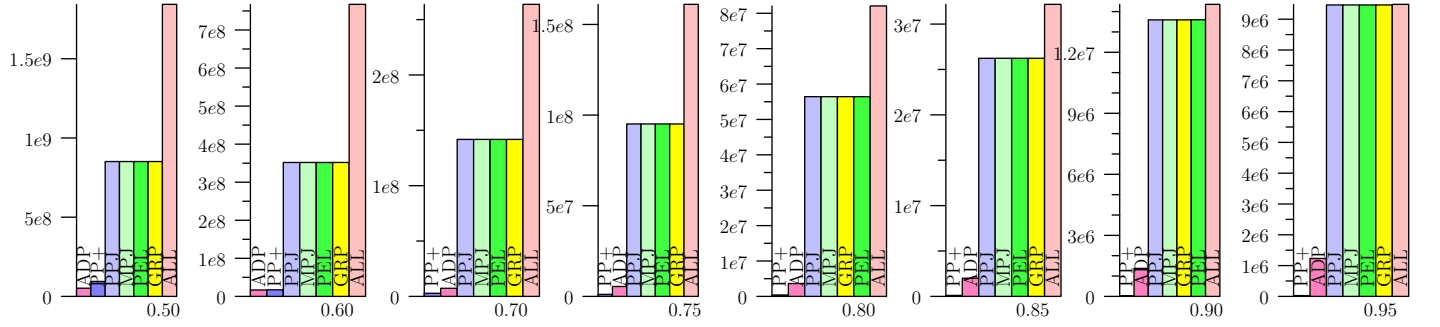


Figure 72: BMS-POS, Jaccard

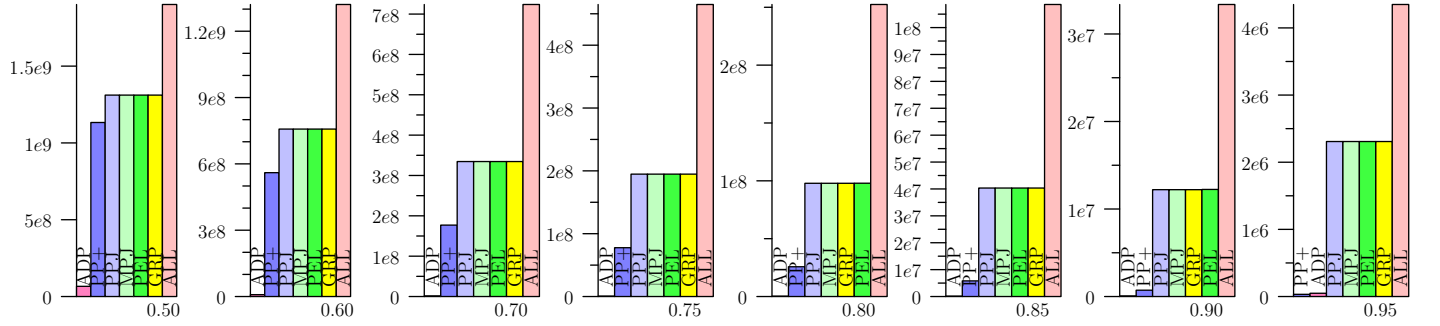


Figure 73: DBLP, Jaccard

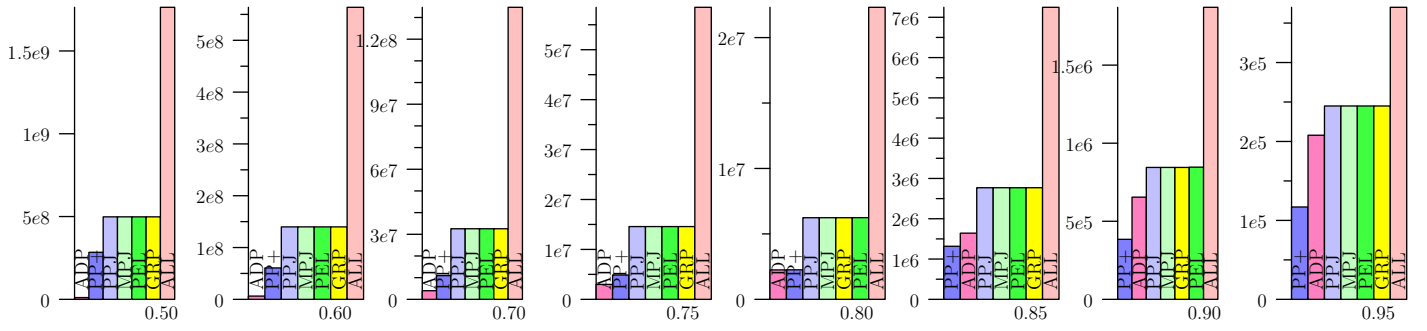


Figure 74: ENRON, Jaccard

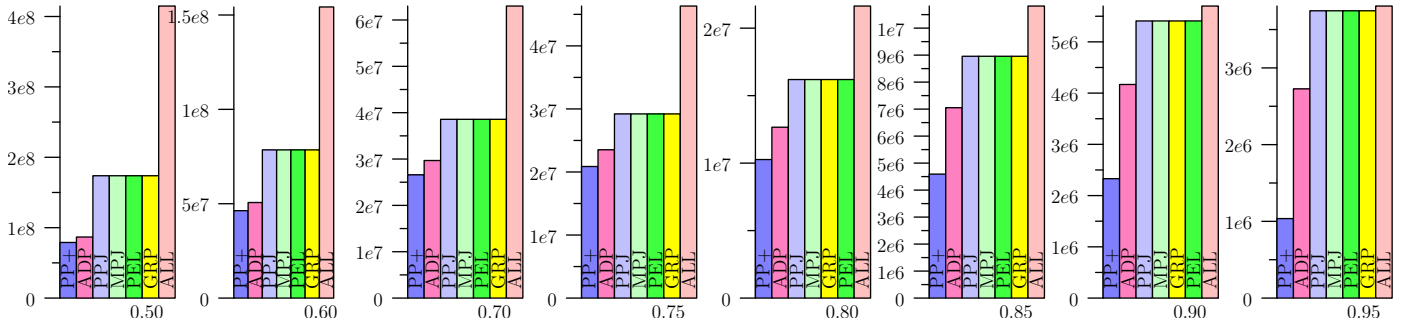


Figure 75: FLICKR, Jaccard

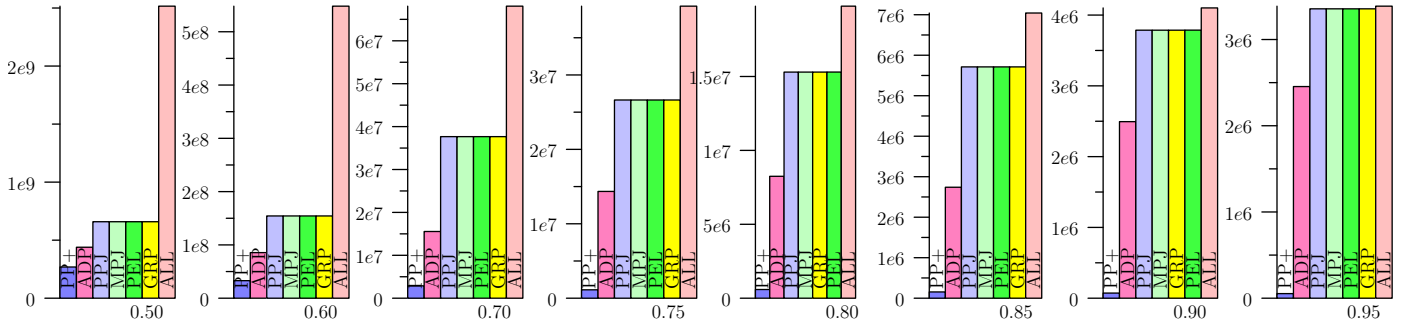


Figure 76: KOSARAK, Jaccard

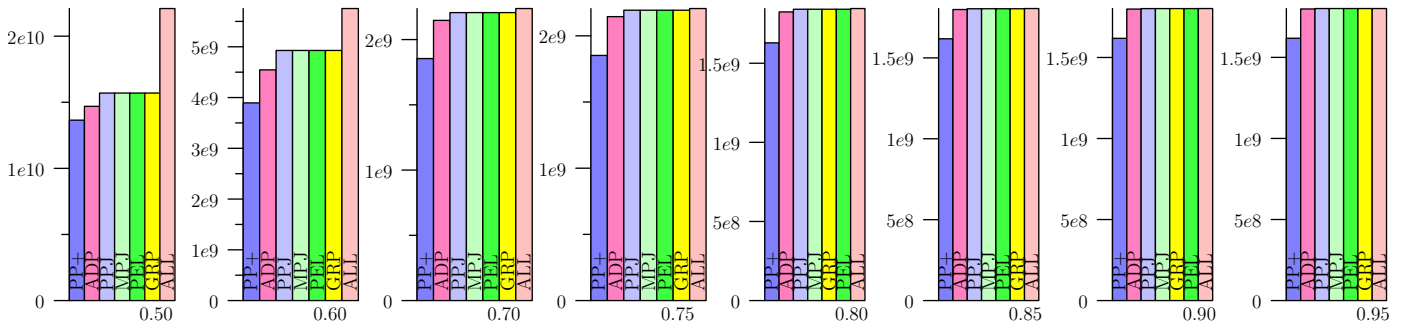


Figure 77: KOSARAK, no deduplication, Jaccard

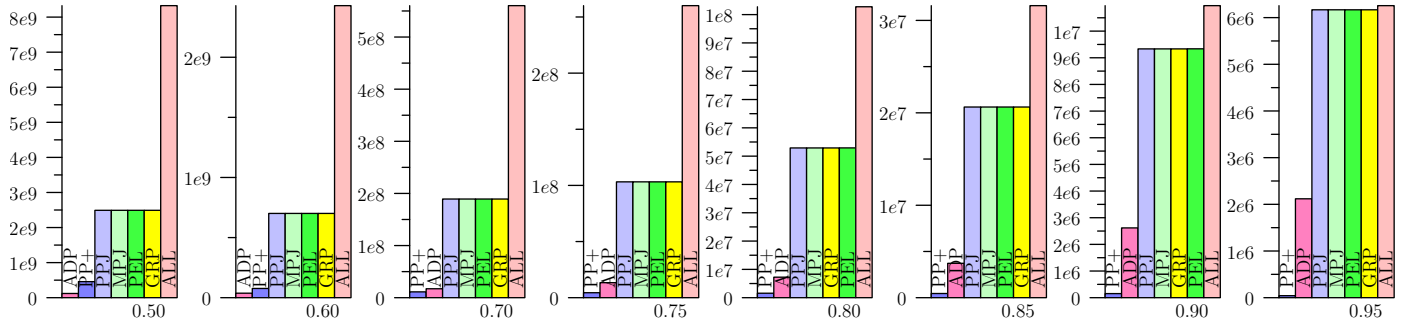


Figure 78: LIVEJ, Jaccard

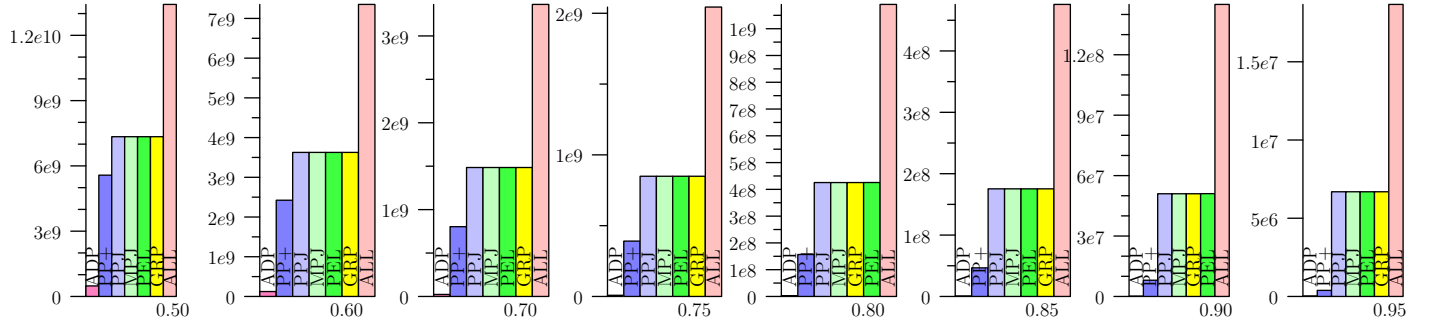


Figure 79: NETFLIX, Jaccard

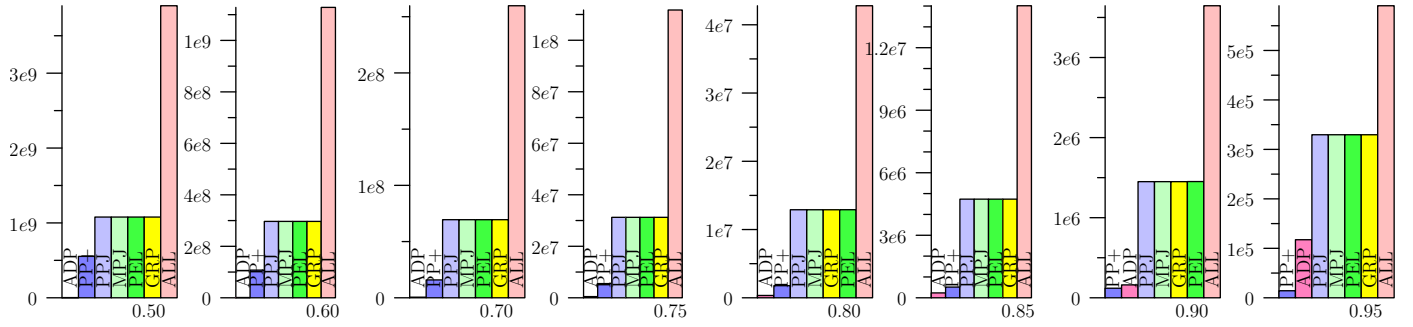


Figure 80: ORKUT, Jaccard

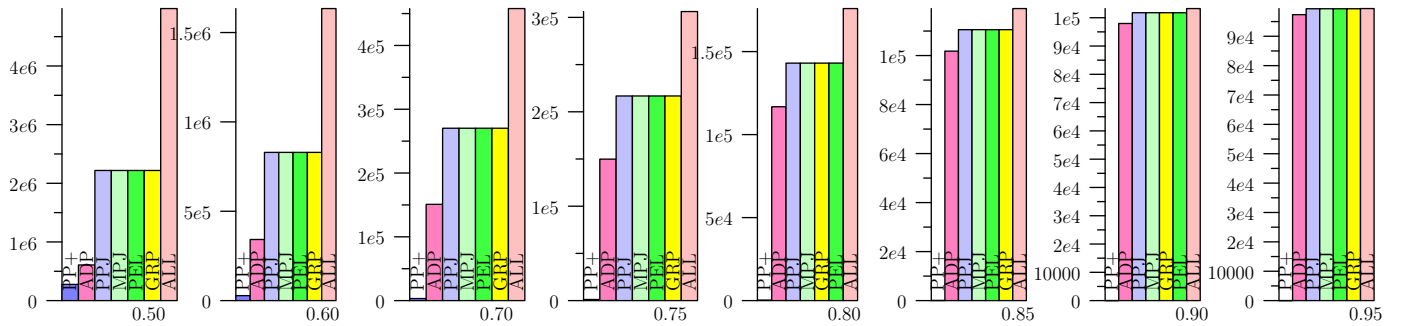


Figure 81: SPOT, Jaccard

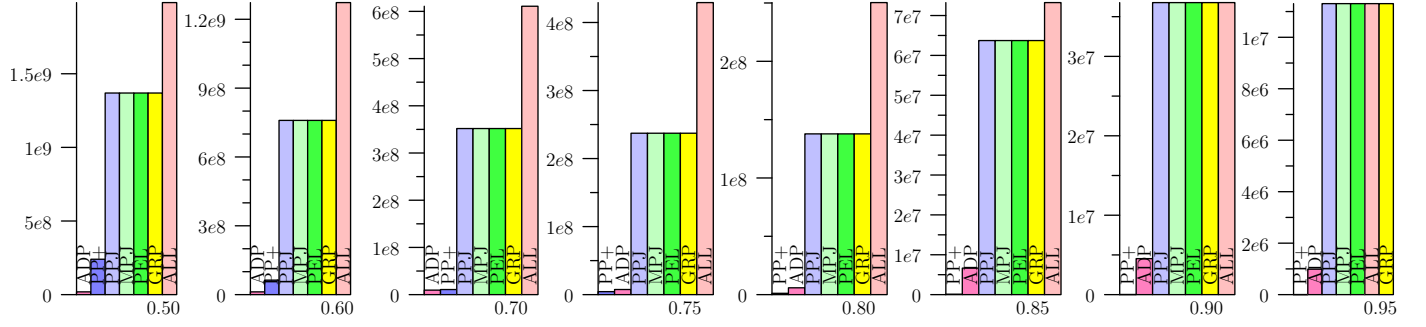


Figure 82: UNIFORM, Jaccard

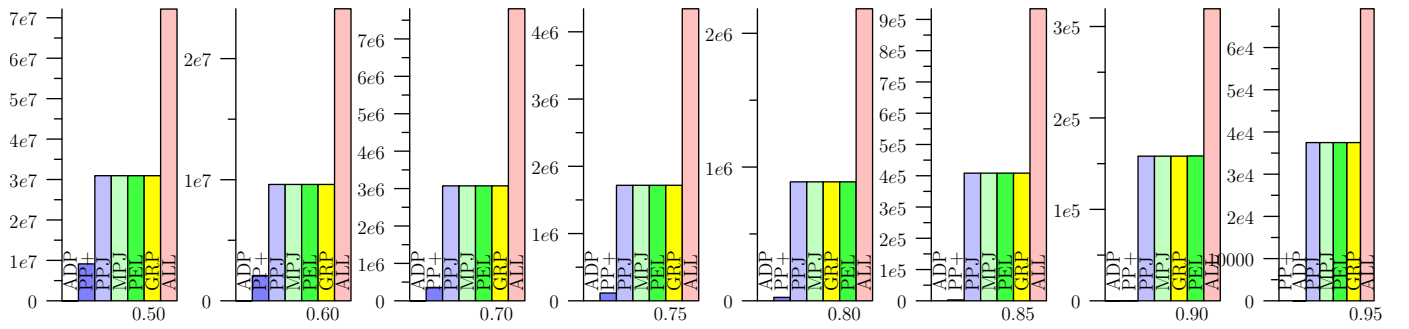


Figure 83: ZIPIF, Jaccard

F. SUMMARY TABLES

F.1 Jaccard

Table 10: Summary statistics of gap factors, Jaccard, self-join.

	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

Table 11: Fastest algorithms, Jaccard, Self-Join.

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

Table 12: Slowest algorithms, Jaccard, Self-Join.

Dataset	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	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
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

Table 13: Gap factor per dataset and threshold, Jaccard, self-join.

Dataset	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 14: Winner per threshold, Jaccard, self-join.

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

F.2 Dice

Table 15: Summary statistics of gap factors, Dice, self-join.

	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 16: Fastest algorithms, Dice, Self-Join.

Dataset	Threshold							
	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	PEL	PPJ	PEL	PEL	GRP	ALL	GRP	GRP
	1540	443	111	81.0	23.6	8.60	2.40	1.43
	PPJ	PEL	PPJ	PPJ	GRP	GRP		
BMS-POS	PPJ	PPJ	PPJ	PPJ	GRP	GRP	GRP	GRP
	196	76.2	27.1	15.6	7.50	2.78	0.852	0.140
				GRP	GRP	PPJ	ALL	
DBLP	PPJ	ADP	ADP	ADP	ADP	ADP	ADP	PPJ
	349	178	74.5	47.0	26.7	12.3	3.79	0.612
		PPJ						GRP
ENRON	PPJ	PPJ	PPJ	PPJ	PPJ	PEL	PEL	ALL
	333	119	33.4	16.1	7.20	2.96	1.25	MPJ
	GRP	GRP	GRP	GRP	GRP	PPJ	GRP	0.364
FLICKR	PEL	PEL	PPJ	PPJ	PPJ	ALL	ALL	ALL
	71.3	24.3	9.02	5.70	3.50	2.07	1.04	0.396
	PPJ	PPJ	PEL	PEL	ALL	PPJ	PPJ	GRP
KOSARAK	PEL	PPJ	PEL	PEL	PPJ	ALL	ALL	GRP
	384	82.1	15.3	9.26	2.85	0.956	0.344	ALL
	PPJ	PEL	PPJ	GRP	GRP	GRP	GRP	0.132
LIVEJ	PPJ	PPJ	PPJ	PEL	PEL	PEL	PEL	ALL
	2439	775	192	86.9	35.0	12.8	5.02	1.72
		PEL	PEL	PPJ	PPJ	PPJ	GRP	MPJ
NETFLIX	PPJ	ALL	ALL	ALL	ADP	ADP	ADP	PPJ
	5250	2375	870	496	217	86.7	27.2	4.12
	GRP	PPJ	PPJ	ADP				GRP
ORKUT	PPJ	PPJ	PPJ	PPJ	PPJ	GRP	GRP	MPJ
	1440	471	140	77.8	43.6	22.8	10.4	3.94
			GRP	GRP	GRP	PPJ	PPJ	ALL
SPOT	PPJ	ALL	ALL	ALL	ALL	ALL	ALL	ALL
	1.42	0.708	0.396	0.316	0.220	0.164	0.120	MPJ
	ALL	PPJ	MPJ	MPJ	MPJ	PEL	PEL	PEL
UNIFORM	ADP	ADP	ADP	ADP	GRP	GRP	GRP	GRP
	79.0	55.7	35.2	24.7	14.3	4.38	1.18	0.176
				PPJ	ADP			
ZIPF	ADP	ADP	PPJ	PPJ	PPJ	PPJ	ALL	ALL
	15.8	4.85	1.58	0.928	0.544	0.304	0.160	0.060
	PPJ	PPJ		ALL	ALL	ALL	PPJ	

Table 17: Slowest algorithms, Dice, Self-Join.

Dataset	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+	PP+	ADP
DBLP	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ENRON	PP+	PP+	PP+	PP+	PP+	ADP	ADP	ADP
FLICKR	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
KOSARAK	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
LIVEJ	PP+	PP+	ADP	ADP	ADP	ADP	ADP	ADP
NETFLIX	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ORKUT	PP+	PP+	PP+	PP+	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+	PP+	PP+	ADP	ADP	ADP

Table 18: Gap factor per dataset and threshold, Dice, self-join.

Dataset	Threshold							
	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	3.36	3.12	3.25	3.61	3.07	4.30	5.23	6.28
BMS-POS	1.94	1.78	1.64	1.58	1.52	1.50	1.64	2.66
DBLP	2.62	3.24	4.00	4.05	3.79	3.28	2.61	1.61
ENRON	3.34	2.99	2.51	2.22	1.88	1.61	1.93	2.40
FLICKR	2.33	2.05	1.95	1.94	1.87	1.99	2.38	3.52
KOSARAK	2.17	2.54	2.63	3.01	1.88	1.79	2.57	4.03
LIVEJ	2.22	2.03	1.88	1.99	2.06	2.29	2.62	3.50
NETFLIX	2.51	2.85	3.04	2.94	3.10	2.75	2.10	1.40
ORKUT	3.09	2.90	2.27	1.85	1.84	2.00	2.18	2.47
SPOT	1.71	2.02	2.53	2.70	3.04	3.32	3.70	4.23
UNIFORM	3.41	2.73	1.97	1.73	1.61	2.14	3.09	3.80
ZIPF	2.95	2.73	2.31	2.10	1.83	1.78	2.48	4.20

Table 19: Winner per threshold, Dice, self-join.

Threshold	Algorithm						
	ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
0.50	0	7	0	0	3	2	0
0.60	2	6	0	0	1	3	0
0.70	2	6	0	0	2	2	0
0.75	2	5	0	0	3	2	0
0.80	1	6	0	0	1	2	2
0.85	4	1	0	0	2	2	3
0.90	4	0	0	0	2	2	4
0.95	6	2	0	3	1	0	4
sum	21	33	0	3	15	15	13

F.3 Cosine

Table 20: Summary of gap factors, Cosine, self-join.

	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

Table 21: Fastest algorithms, Cosine, self-join.

Dataset	Threshold							
	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	PEL	PEL	PEL	PEL	PPJ	ALL	GRP	GRP
	1706	510	134	93.9	32.8	8.88	2.47	1.46
	PPJ	PPJ	PPJ	PPJ	PEL	GRP		
BMS-POS	PPJ	PPJ	PPJ	PPJ	PPJ	PPJ	ALL	GRP
	243	94.6	32.4	17.4	8.32	3.03	0.904	0.144
				GRP	GRP	GRP	GRP	
DBLP	PPJ	PPJ	ADP	ADP	ADP	ADP	ADP	PPJ
	378	196	83.9	52.0	29.4	13.4	4.06	0.624
		ADP						
ENRON	PPJ	PPJ	PPJ	PPJ	PEL	PEL	PEL	ALL
	425	145	39.0	18.3	7.89	3.10	1.28	0.372
	GRP	GRP	GRP	GRP	PPJ	PPJ	PPJ	MPJ
FLICKR	PEL	PEL	PEL	PPJ	PPJ	PPJ	ALL	ALL
	83.2	28.7	10.2	6.16	3.66	2.13	1.09	0.404
		PPJ	PPJ	PEL	PEL	ALL	PPJ	PPJ
KOSARAK	PEL	PEL	PEL	PEL	PPJ	ALL	ALL	GRP
	445	99.2	18.1	10.7	3.42	1.02	0.360	ALL
				GRP	PEL	PPJ	GRP	0.136
LIVEJ	PEL	PEL	PEL	PEL	PEL	PEL	PEL	ALL
	3235	971	227	96.8	38.0	13.6	5.20	1.76
	PPJ	PPJ	PPJ	PPJ	PPJ	PPJ	PPJ	MPJ
NETFLIX	PPJ	ALL	ALL	ALL	ADP	ADP	ADP	PPJ
	6687	2974	1055	586	258	97.1	29.2	4.24
	ALL	PPJ	PPJ	PPJ	GRP			GRP
ORKUT	PPJ	PPJ	PPJ	PPJ	GRP	GRP	GRP	ALL
	1859	587	163	87.8	48.5	24.2	10.7	4.02
		GRP	GRP	GRP	PPJ	PPJ	MPJ	MPJ
SPOT	PPJ	ALL	ALL	ALL	ALL	ALL	ALL	ALL
	1.76	0.868	0.476	0.344	0.252	0.172	0.124	0.084
	PEL	PEL	PPJ	PPJ	MPJ	MPJ	PEL	MPJ
UNIFORM	ADP	ADP	ADP	PPJ	PPJ	GRP	GRP	GRP
	90.4	61.8	40.1	27.7	15.9	5.46	1.29	0.184
			PPJ	ADP	GRP			
ZIPF	PPJ	ADP	PPJ	PPJ	PPJ	PPJ	ALL	ALL
	16.1	5.12	1.66	0.956	0.556	0.320	PPJ	PPJ
	ADP	PPJ		ALL	ALL	ALL	0.164	0.064

Table 22: Slowest algorithms, Cosine, self-join.

Dataset	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+	PP+	ADP
DBLP	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
				ADP				
ENRON	PP+	PP+	PP+	PP+	PP+	ADP	ADP	ADP
FLICKR	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
KOSARAK	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
LIVEJ	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
NETFLIX	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ORKUT	PP+	PP+	PP+	ADP	ADP	ADP	ADP	ADP
				PP+				
SPOT	ADP	ADP	ADP	ADP	ADP	ADP	ADP	ADP
UNIFORM	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ZIPF	PP+	PP+	PP+	PP+	PP+	ADP	ADP	ADP

Table 23: Gap factor per dataset and threshold, Cosine, self-join.

Dataset	Threshold							
	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
AOL	3.90	3.79	3.95	3.68	3.07	4.29	5.20	6.32
BMS-POS	1.99	1.83	1.66	1.59	1.53	1.50	1.58	2.58
DBLP	2.68	3.24	3.99	4.05	3.83	3.22	2.57	1.63
ENRON	3.36	3.02	2.52	2.20	1.88	1.68	2.01	2.43
FLICKR	3.20	2.60	2.26	2.11	2.03	2.13	2.39	3.50
KOSARAK	2.78	3.68	3.59	3.23	2.21	1.77	2.51	3.71
LIVEJ	2.42	2.36	2.68	2.31	2.26	2.37	2.63	3.44
NETFLIX	2.43	2.80	3.03	2.92	2.97	2.73	2.09	1.40
ORKUT	2.85	2.70	2.19	1.85	1.86	2.06	2.23	2.42
SPOT	1.85	1.97	2.36	2.65	2.90	3.30	3.61	4.48
UNIFORM	3.09	2.63	1.91	1.64	1.56	1.89	2.90	3.67
ZIPF	3.02	2.75	2.34	2.13	1.86	1.74	2.41	4.06

Table 24: Winner per threshold, Cosine, self-join.

Threshold	Algorithm						
	ALL	PPJ	PP+	MPJ	PEL	ADP	GRP
0.50	0	7	0	0	4	1	0
0.60	2	4	0	0	4	2	0
0.70	2	4	0	0	4	2	0
0.75	2	6	0	0	3	1	0
0.80	1	6	0	0	2	2	1
0.85	3	3	0	0	2	2	2
0.90	5	1	0	0	2	2	3
0.95	7	3	0	0	0	0	4
sum	22	34	0	0	21	12	10

G. TOKEN FREQUENCY DISTRIBUTIONS

In this section, we provide plots of token frequency distributions.

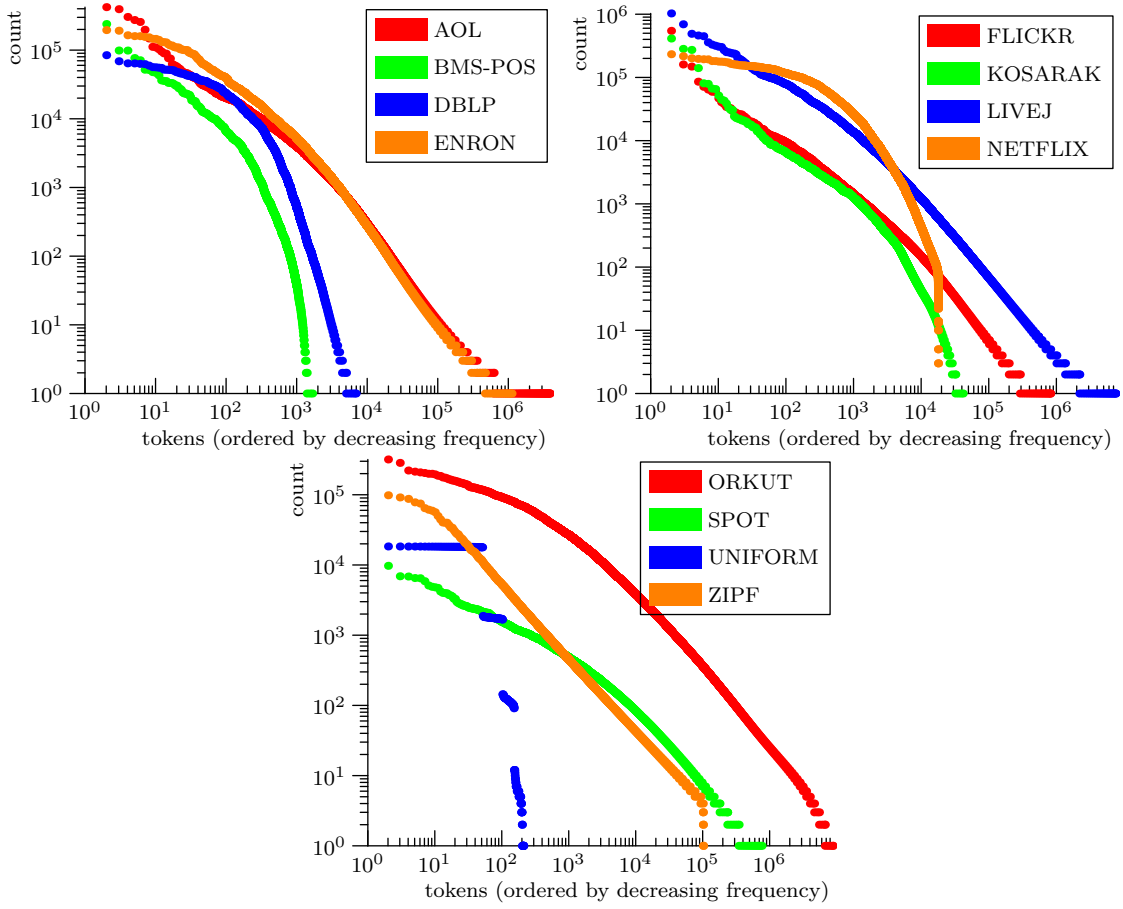


Figure 84: Token frequency distributions