Set Similarity Join Techniques On The Containment Similarity

Wei Zheng Yichen Zhang Jianing Guo Meng Zhang

Abstract

In this work, we investigate set similarity join techniques on the containment similarity function, finding all pairs of sets for which containment similarity score is above a given threshold. We implement existing set similarity join algorithms on the containment similarity function. We show the performances of all algorithms on datasets with different characteristics across a wide set of thresholds. The containment function gives different result compare to the Jaccard function due to unique characteristics of the containment function. Moreover, the containment similarity function produces a huge number candidate pairs and output pairs. The prefix filters and the length filters are the two fundamental filters, and they do not work well on the containment function. Algorithms with complex filters can relief the inefficiency of the two filters.

1 Introduction

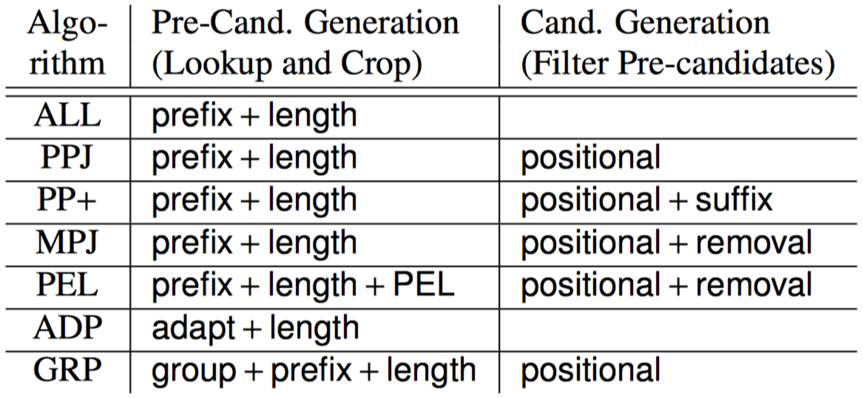
The set similarity join aims to find all similar set pairs from two collections of sets. Similarity join techniques are required in many real-world applications, such as near duplicate detection [XWL+11], data cleaning [CGK06], data integration [IFF+99] and machine learning [DDH01]. Given two collections of sets, R and S, a similarity function *sim* and a specified threshold t, the set similarity join is defined as finding and combining all pairs of sets R.r and S.s from R and S such that $sim(R.r, S.s) \geq t$.

Set size of s	Jaccard $\frac{\lvert r\cap s\rvert}{\lvert r\cup s\rvert}$	Containment $\frac{\lvert r\cap s\rvert}{\lvert r\rvert}$
10	0.5	1
100	0.09	1
1000	0.009	1

Table 1: Jaccard and Containment similarity score, given query set size r=10

Most of the existing similarity join techniques utilize Jaccard as the similarity function [MAB16]. Jaccard, however, has its limit solving set similarity join problems when the query set is the subset of the probing set. Table 1 compares the different values of Jaccard and Containment similarity score given the sizes of two sets r and s such that $r \subseteq s$. For example, two sets with significantly different set sizes may get a fairly small similarity score by Jaccard, while their Containment Similarity Score remains at 1.0. When implementing the set similarity join, two sets should be joined if one is a subset of the other. We see that these pairs are neglected by the Jaccard similarity function, while Containment can better alleviate the subset problems. Therefore it is important that we gain a better understanding of the set similarity join techniques on Containment similarity function.

The goal of this work is to study set similarity join techniques on the containment similarity function. To perform such a study, we focus on 7 existing set similarity algorithms, including AllPairs [BMS07], PPJoin and PPJoin+ [XWL+11], MPJoin [RH11], MPJoin-PEL [MA14], AdaptJoin [WLF12], and GroupJoin [BGM12]. All seven algorithms implement the filter-verification framework [MAB16], where the core technique is called prefix filter. The prefix filter first sorts all the tokens in each set in a given the global order and then inspects the first few tokens as the prefix. The lookup returns the inverted lists of sets that have at least one common token in their prefixes with the query set. Then length filter is applied to crop the inverted lists, and the set entries in the cropped inverted lists are called pre-candidates. Pre-candidates filtered through more sophisticated filters are the final candidates.



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

Figure 1: Algorithms with their filters [MAB16]

AllPairs implements pre-candidate generation with prefix filter and length filter, based on which the other six algorithms have a further implementation to generate candidates.

AllPairs (ALL): ALL [BMS07] uses prefix filtering to build the inverted lists. Then, length filtering is applied to crop the inverted lists and generate pre-candidates.

Based on ALL, PPJ, PP+, MPJ and PEL, GRP, ADP employ more sophisticated filters during candidate generation.

PPJoin (PPJ): Based on ALL, PPJ uses positional filtering to prune pre-candidates based on the position of the last common token that appears in the pair [XWL+11].

PPJoin+ (PP+): Based on PPJ, PP+ uses suffix filtering to recursively check the token overlap in two partitions of the pre-candidates [XWL+11] to further tighten the condition of being a candidate.

MPJoin (MPJ): Since sets in the inverted lists are sorted by increasing set size by the ALL algorithm, MPJ can safely remove an obsolete set entry from the inverted list if that set entry will be filtered by all positional filters throughout the candidate generation process [RH11]. The technique in MPJ is called suffix filter.

MPJoin-PEL (PEL): Based on MPJ, PEL uses the position-enhance length filter to tighten the upper bound of set size [MA14], which has better performance than the original length filter.

ADP and GRP make improvements on prefix filters.

AdaptJoin (ADP): To decide how many tokens should be indexed, standard prefix-filtering framework goes through all possible prefix lengths and selects the best one (the maximum prefix length) as the length of indexed tokens for all sets. It is fairly expensive. To address the issue, ADP proposes a variable-length prefix scheme, which selects an optimal prefix length based on the sum of the filter cost and verification cost. When the prefix length increases, the filtering time increases and the verification time decreases. The overall time for filter-verification framework first increases and then decreases with the increase of prefix length. ADP aims to find the "sweet spot" that has the least cost. Instead of indexing tokens by the maximum prefix length throughout all sets in ALL algorithm, ADP dynamically indexes tokens according to the overall cost [WLF12]. Therefore ADP has better performance when sets have long prefixes.

GroupJoin (GRP): Consider two collections of sets with a large number of common tokens, many of the sets share a common prefix. When simply applying prefix-based algorithms like PPJ, the algorithm would index the same prefix many times and compute the similarity score for the same overlaps many times as well. Based on PPJ, GRP proposes a "group" idea [BGM12]. GRP first groups together all sets that share the same prefix, then it applies the set similarity join technique over the groups of sets with the same prefix. Then the group can be viewed as a single set, and all sets in the group can be pruned in a batch at the same time. This technique effectively reduces the overlap of the same prefix in the inverted list and further reduces the number of sets that need to be processed during candidate generation. Figure 1 shows the filters used in all seven tested algorithms.

To perform our study, we implement Containment similarity function on tested algorithms mentioned above. We run the experiments on nine real-world datasets, Canadian open data, and Wikitable extracted from 2008 Wikipedia. We have the following findings in particular:

1. The prefix filter is not effective because all domain tokens must be indexed.

2. The length filter is not effective because there is no domain size upper bound.

3. GroupJoin performs well on large thresholds due to the grouping effect.

4. AdaptJoin is the winner in most cases thanks to the complex filter on pre-candidate set.

5. PPJoin, MPJoin, MPJoin-PEL have an average performance.

6. AllPairs and PPJoin+ have poor performance in all cases.

The rest of the report is structured as follows. We discuss related works in Section 2. Section 3 describes the implementation method and details for similarity join techniques on Containment. Section 4 describes the datasets used in our experiments. Experimental performance evaluation and analysis are presented in Section 5. Finally, we conclude in Section 6 and discuss future work in Section 7.

2 Related Works

$$Jaccard(Q, X) = \frac{Containment(Q, X)}{\frac{|X|}{|Q|} + 1 - Containment(Q, X)} \tag{1}$$

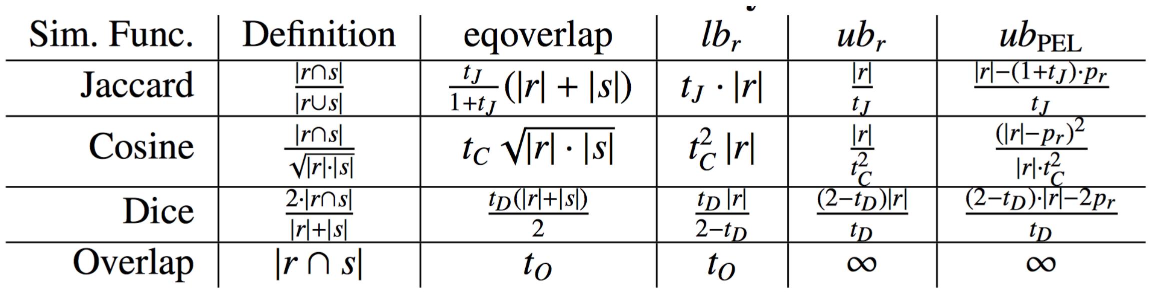
Several recent works use the approximate method to find similar sets. Containment threshold can be transformed to Jaccard threshold given domain size and query size (Equation 1). Asymmetric Minwise Hashing [SL15] applies a pre-processing step called the asymmetric transformation. This approach pads the domains with new values, making all domains in the index have the same size as the largest domain. The authors show that Jaccard similarity between an untransformed signature and a transformed signature is monotonic concerning set containment [SL15]. Thus, MinHash LSH can be used to index the transformed domains, such that the domains with higher set containment scores will have higher probabilities of becoming candidates. However, Asymmetric MinHash LSH is difficult to tune for containment threshold [ZNPM16]. Also, padding reduces min-hash accuracy especially when domain sizes have a skewed distribution. When the skew is high, the amount of padding required becomes very large, making the probability of qualifying domains becoming candidates very low and sometimes nearly zero [ZNPM16].

LSH Ensemble [ZNPM16] uses multiple MinHash LSH built on domain size partitions to approximate containment search and maintain accuracy. It finds an optimal partition and uses upper bound domain size for each partition to find the Jaccard threshold. The estimation would introduce false positives but not false negatives. The approximation method is much better than using the upper bound of the entire domain set and leads to higher precision. LSH Ensemble [ZNPM16] adds a step to remove any false positives introduced by the Jaccard threshold approximation. LSH Ensemble [ZNPM16] is also self-tunable at query time for the partitions, given any threshold. The partition also enables parallel execution of the join, significantly reducing the time required.

SSJoin [MAB16] uses the same exact search algorithms we discussed in the previous section on the Jaccard, Cosine, and Dice similarity functions. Mann found the PPJoin, GroupJoin, and AllPairs have the best performance. AllPairs wins on the largest number of cases [MAB16]. AdaptJoin and PPJoin+ apply sophisticated filters and generate small candidate set, but the candidate generation is too slow compared to verification time [MAB16]. Therefore, AdaptJoin and PPJoin+ have large performance difference with the best algorithms. Other algorithms except AdaptJoin and PPJoin+ have the similar performance.

3 Implementation

The original implementation uses Jaccard, Dice, and Cosine similarity functions and they have been normalized to equivalent overlap. Prefix filters and length filters may have different performance as similarity functions vary. Figure 2 gives equivalent overlap, lower bound length filter, and upper bound length filter for similarity functions. When using length filter, Jaccard can efficiently reduce the number of candidates with its tight upper and lower bound on size. For overlap, domain size does not have an upper bound. In this case, the overlap has to go through many more candidates than Jaccard does. Cosine has loosened the upper and lower bounds compared to Jaccard so that the runtime for Jaccard will be slower than Jaccard.



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

Figure 2: Similarity functions and set size bounds for set r and s [MAB16]

eqoverlap: equivalent overlap for normalized thresholds

lb: size lower bound on join partners for r

up: size upper bound on join partners for r

ubPEL: size upper bound on join partners for r at matching position pr (MPJoin-PEL)

	Containment1	Containment2	Jaccard
Definition	$\frac{\|r\cap s\|}{\|r\|}$	$\frac{\|r\cap s\|}{\|s\|}$	$\frac{\|r\cap s\|}{\|r\cup s\|}$
Equivalent Overlap	$t\|r\|$	$t\|s\|$	$\frac{t}{1+t}(\|r\| + \|s\|)$
Max Query Prefix Length	$\|r\| - t\|r\| + 1$	$\|r\|$	$\|r\| - t\|r\| + 1$
Max Domain Prefix Length	$\|s\|$	$\|s\| - t\|s\| + 1$	$\|s\| - t\|s\| + 1$
Domain Upper Bound	∞	$\frac{\|r\|}{t}$	$\frac{\|r\|}{t}$
Domain Lower Bound	$t\|r\|$	1	$t\|r\|$

Table 2: Containment normalization

We normalize containment to overlap and implement the changes in the original code. Summarized containment normalization can be found in Table 2. We noticed that the containment similarity function does not have the symmetric property like cosine, dice, and Jaccard. Containment is not symmetric such that Cont(Q, X)!=Cont(X, Q), so we need to consider the query and input domains separately. For Jaccard, the max prefix calculation is the same for the query and the input domain (Table 2). Let r be the query, s be the input domain, and t be the containment threshold. For a query, the max prefix size that needs to considered is equal to $|r| - t|r| + 1$ where the max prefix for domain input is $|s|$. This means that for the input domain we need to index all of the tokens without knowing the query size. Let r be token 10, s be $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$, and containment threshold be 1. If the index does not contain the last token, it will not find the matching token. This would decrease the efficiency of the prefix filter because every domain's token needs to be indexed. For Jaccard, given the same query, domain, and threshold, we only need to index the first token for both query and input domain. For comparison, we consider another containment score with the denominator being the domain input size. Its max query prefix length and max domain prefix length is the reverse of the original containment score (Table 2).

We also calculate the domain size upper bound and lower bound for the length filter (Table 2). For containment, there is no upper bound for domain size as the domain does not appear in the denominator of the containment calculation. However, there is a lower bound for domain size. The length filter has a huge impact on the join time as having no upper bound on the length filter would generate a huge pre-candidate set. Assume that we have a query with length 10 and there are domains with size 5,10,15,20,...,100. If we use Jaccard and the threshold is 0.5, then we only need to look into domains with size 5,10,15 and 20. However, if we use containment, we need to look into all of the domain sizes. This issue would be more severe if the domain size is much bigger than the query size.

Since the prefix filter and length filter are not very effective on the containment similarity function, we expect AllPairs to perform poorly. Performance of MPJoin and PPJoin should be very similar, and MPJoin would be slightly better in some cases with the additional removal filter. MPJoin and MPJoin-PEL should perform about the same because the upper bound of length filer is the domain size. Therefore, the positional enhanced length filter would be the same as the normal length filter. PPJoin+ should not perform very well as we do not have an upper bound on the length filter. Longer suffix would damage the performance of PPJoin+. GroupJoin should perform as described in the original paper [MAB16]. It would perform even better when there are a large number of sets with the same prefix. AdaptJoin applies a complex filter on the pre-candidate sets, extending the prefix of the query. Since input domains already have their maximum prefix which is the length, their prefixes would not be extended. Therefore, the extended prefixes of queries would be compared with the prefix of input domains. PPJoin+ and AdaptJoin have poor performance on Jaccard similarity function [MAB16]. In contrast, we expect that AdaptJoin's complex filters would perform well when the number of indexed tokens increases.

4 Datasets

4.1 Wikitable Data

We created a new dataset called Wikitable. Wikitable contains relational tables extracted from 2008 English Wikipedia HTML files [wik]. The size of the original compressed HTML data is approximately 209 GB, and it contains more than 14 million HTML files. We apply several filters to prune the files.

1. First, we remove the HTML files that are not regular Wikipedia pages which contain only images or conversations between editors or users. Below are examples of file names belonging to such pages:

   - image HTMLNAME.html
   - User_Talk HTMLNAME.html
   - Talk HTMLNAME.html

2. After the first filter, we still have over 7 million files left; we apply the second filter based on the file size. If the file size is less than 1 KB, the file contains a redirect link to another page. For example, the Wikipedia page for UK would redirect the user to the "United Kingdom" Wikipedia page. These redirected pages do not contain any relational tables, so we can safely remove them.

3. After the first two filters, we are left with HTML files that are likely to have relational tables. By manual inspection of the some of the HTML files we notice that most of the table is used for formatting purpose. We count the number of $< table >$ tags have the specific class. We found that "wikitable" and "sortable" are the two classes with relational tables. Below are examples of classes and number of tables has the class. We use BeautifulSoup to scrapes tables from HTML files into CSV files. After removing empty tables and tables with a single column, around 500,000 tables remain.

   - 'infobox': 1045293
   - 'collapsible': 960110
   - 'nowraplinks': 935434
   - 'navbox': 895603
   - 'toc': 812766
   - 'autocollapse': 741597
   - 'wikitable': 545206
   - ...
   - 'sortable': 56155

Dataset	Total number of sets in collection	Max set size	Average set size	Number of different tokens
AOL	10,054,184	245	3	3,900,000
BMS-POS	515,597	164	7	1,657
KOSARAK	990,002	2,497	8	41,000
UNIFORM	99,935	25	10	208
LIVEJ	3,201,203	300	35	7,500,000
ZIPF	3,201,203	300	35	7,500,000
DBLP	100,000	1,625	86	6,864
ORKUT	2,732,271	40,425	120	8,730,857
ENRON	517,422	3,162	134	1,100,000
Open data	3,367,520	22,075,531	466	612,819,386
Wikitable	1,613,775	8,886	18	8,291,434

Table 3: Characteristics of datasets

4.2 Other Datasets

We use nine datasets provided in the SSjoin paper [MAB16], open data, and the Wikitable dataset. ZIPF and UNIFORM are synthetic datasets with 100k sets each [MAB16]. Set size is randomly drawn from a Poisson distribution, and tokens are drawn from a Zipfian or uniform distribution. The other nine datasets are real-world datasets. We use these many datasets to avoid domain bias.

According to the paper on SSjoin [MAB16], 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. All tokens are numbered without gaps and are represented by their integer IDs. The token numbering is based on token frequencies: the lower the frequency of a token, the lower its ID. GroupJoin benefits from its sorting sets with the same size alphabetically because it can then identify duplicate prefixes in a single scan over the sets in the input collection.

Characteristics of datasets are summarized in Table 3. Each of the datasets has unique characteristics but is comparable to other datasets in one or more scales. For example, AOL and KOSARAK have small average set size but the relatively large number of different tokens. BMS-POS and Uniform have small average set size and a small number of different tokens. Both of these two sets of comparable datasets differentiate on the max set size and the total number of sets in the collection. Open data has the largest average set size and also the largest amount of different tokens. Wikitable data we extract from the web has the similar characteristic as AOL. Description of the datasets except for Open data and Wikitable data can be found in the SSJoin paper [MAB16].

5 Evaluation

We conducted our experiments on a machine with two Intel E5-2680 CPUs, 20MB L3 Cache, and 256 GB RAM. We executed one join at a time without interference from other processes. For each join, we executed three times to collect running time and other statistics separately. Collect statistics would increase the running time. In this evaluation section, we are going to go through join time which includes time for building the index, generating candidate sets and verifying candidates. Next, we are going to look into the number of lookups, size of pre-candidate sets, and size of candidates sets. Also, we want to observe the effect of duplication set and lexicographical shuffle on the input domains. Finally, we measure memory usage of the all the algorithms.

We randomly sample 10% of the data without replacement of each data set as the query, with the exception that for open data we select 10000 domains as query due to the size of the open data. We run each algorithm on each dataset with 8 thresholds: 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.6, 0.5. We did not include a threshold below 0.5. When containment similarity of a query and an input domain is below 0.5, a join between the query and domain will lose more than half of the information in the query. The original SSJoin paper [MAB16] also did not include any threshold below 0.5. Containment is more sensitive to threshold a small increase in threshold could increase the number of output a lot because the domain size is not in the denominator. Note that AdaptJoin is not working for open data due the memory management issue for the implementation when the file size exceeds some threshold.

To ensure our implementation is correct, we have implemented a brute force application that calculates containment score of any combination of query set and domain set. If the containment score is greater than the provided threshold, the combination of query set and domain set will be included in the output. We ran the brute force application on some datasets on the eight thresholds and compared the output with our implementation of containment on SSJoin [MAB16]. We had the same result for all seven algorithms and in all eight thresholds.

We have other results for Cosine, Dice, Jaccard and the containment that divides domains size. However, in this report, we want to focus on containment, and we do not want to mitigate the coherence of this paper by adding other similarity functions. If you are interested in the more detailed results, please contact Wei Zheng.

5.1 Join Time

Overall join time can be divided into index time, candidate time and verification time [MAB16]. Index time measures the time of building the index which is an array of pointers to the inverted list. Pre-candidate generation time includes the time of applying prefix filter and length filter [MAB16]. Candidate generation time includes the time of filtering the pre-candidate set. All pairs do not have a pre-candidate set filter so candidate generation time would be zero. On the other hand, PPJoin+ applies complex suffix filter on pre-candidate sets to reduce the size of candidate sets and thus reduce the verification time [MAB16].

Verification time includes time of verifying each candidate pair.

5.1.1 Jaccard Vs Containment

From the previous section, we learned that prefix filter and length filter are not effective for containment. Therefore, we expect a huge performance difference between Jaccard and containment. We run experiments on open data at threshold 0.8. Figure 3 shows the performance difference between Jaccard similarity function and containment similarity function. The average join time for Jaccard is around 25 seconds except ADP to be around 96 seconds. However, the minimum join time for containment is around 582 seconds and approximately 2000 seconds for AllPairs. That is around 20 times longer regarding runtime. For the AllPairs algorithm, the number of candidate pairs is 86 million for Jaccard and 1.8 billion for containment. That is around 21 times more candidates. The algorithm outputs 12,864 pairs with Jaccard threshold and 68 million pairs with containment threshold. That is 5400 times more output pairs.

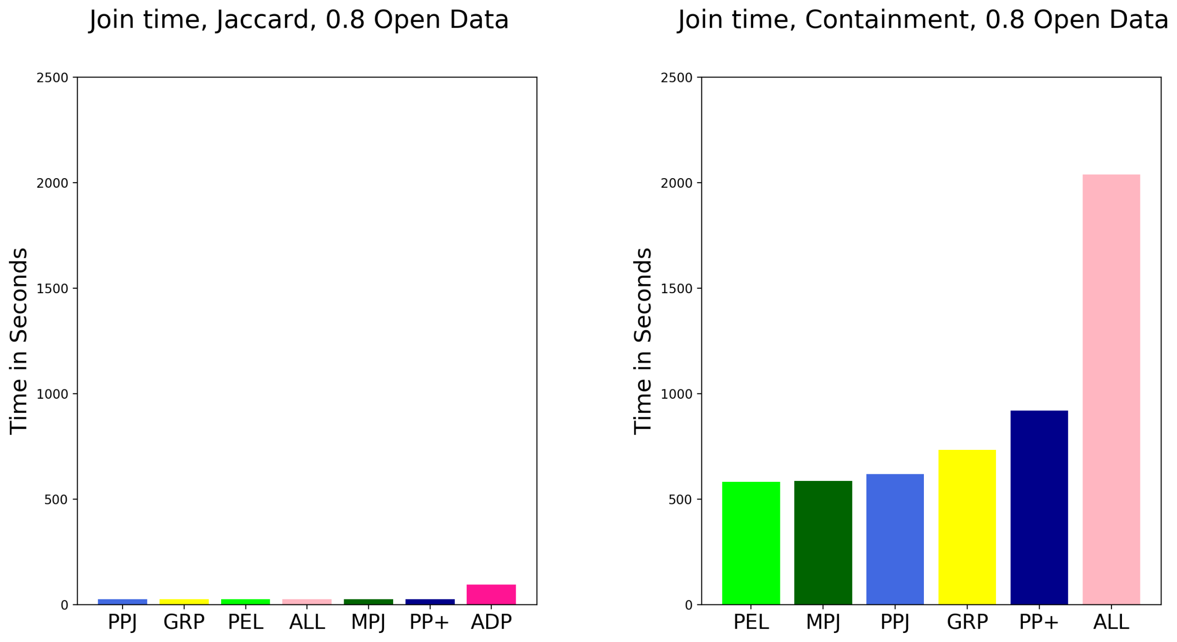


Figure 3: Jaccard Vs Containment

5.1.2 Join Time - Fastest

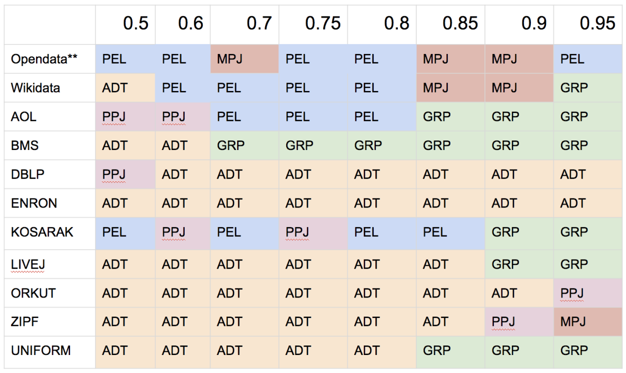
We run experiments using eight thresholds and 11 datasets, which leads to 88 cases in total. AdaptJoin wins on 42 cases, which is about half of the cases. GroupJoin wins in 17 Cases. Winners for the rest cases are PPJoin, MPJoin, and MPJoin-PEL. Performances of PPJoin, MPJoin, and MPJoin-PEL are similar. Figure 4 gives the summary of which algorithm is the fastest at given dataset and threshold. Please note that we do not have results for AdaptJoin on open data. Performance of MPJoin and MPJoin-PEL are very similar. Figure 6 and 7 gives more detail on join time statistics on four of the datasets. Please note AdaptJoin is not included in the Figure 7 for open data due to the memory issue. We expect AdaptJoin to be the fastest algorithm on open data. AdaptJoin performs very well on the ORKUT dataset. ORKUT has similar characteristics to open data in that it has a large number of different tokens and large set sizes. Therefore, we expect AdaptJoin to perform similarly.

The performance of prefix filter for GroupJoin is fairly effective at large thresholds, leading to pre-candidates with a small percentage of false positives. GroupJoin works especially well when there are less different numbers of tokens, which leads to greater probability of similar prefix [MAB16]. BMS-POS is an example of the dataset that has a small number of different tokens.

For small thresholds complex filter on pre-candidate sets pays off [MAB16]. For example, AdaptJoin inspects an extended prefix and maintains multiple inverted lists [MAB16]. AdaptJoin processes more pre-candidates into small candidates set into leads to short verification times. AdaptJoin is very effective at reducing the size of candidate set when the index size is large [MAB16]. However, for some of the datasets like AOL, candidate generation is more expensive than verification which leads to longer join time [MAB16]. When the dataset is similar to AOL and threshold is relatively low, MPJoin, MPJoin-PEL or PPJoin will be the fastest algorithm.

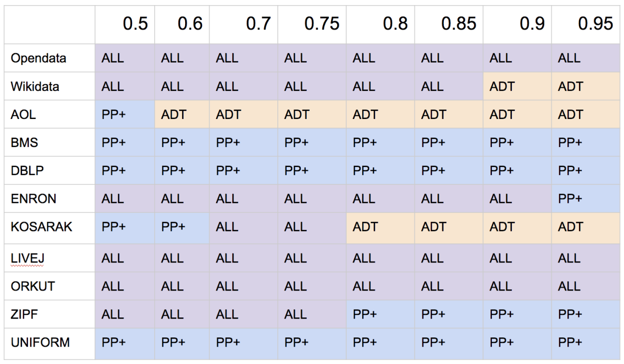
When using the containment similarity function, the number of pre-candidates is huge. In the previous section, we see that containment similarity function generates 21 times more candidates. Therefore, an efficient filter on the pre-candidate set can significantly reduce the size candidate set, verification time and join time.

When the threshold decreases, the conditions on the filters loosen which leads to more output results. All of the algorithms suffer a various level of performance degrading for a lower threshold. GroupJoin suffers the most because of lack of similar prefixes on low thresholds.



	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
Opendata**	PEL	PEL	MPJ	PEL	PEL	MPJ	MPJ	PEL
Wikidata	ADT	PEL	PEL	PEL	PEL	MPJ	MPJ	GRP
AOL	PPJ	PPJ	PEL	PEL	PEL	GRP	GRP	GRP
BMS	ADT	ADT	GRP	GRP	GRP	GRP	GRP	GRP
DBLP	PPJ	ADT	ADT	ADT	ADT	ADT	ADT	ADT
ENRON	ADT	ADT	ADT	ADT	ADT	ADT	ADT	ADT
KOSARAK	PEL	PPJ	PEL	PPJ	PEL	PEL	GRP	GRP
LIVEJ	ADT	ADT	ADT	ADT	ADT	ADT	GRP	GRP
ORKUT	ADT	ADT	ADT	ADT	ADT	ADT	ADT	PPJ
ZIPF	ADT	ADT	ADT	ADT	ADT	ADT	PPJ	MPJ
UNIFORM	ADT	ADT	ADT	ADT	ADT	GRP	GRP	GRP

Figure 4: Join time fastest



	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
Opendata	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
Wikidata	ALL	ALL	ALL	ALL	ALL	ALL	ADT	ADT
AOL	PP+	ADT	ADT	ADT	ADT	ADT	ADT	ADT
BMS	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
DBLP	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ENRON	ALL	ALL	ALL	ALL	ALL	ALL	ALL	PP+
KOSARAK	PP+	PP+	ALL	ALL	ADT	ADT	ADT	ADT
LIVEJ	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
ORKUT	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
ZIPF	ALL	ALL	ALL	ALL	PP+	PP+	PP+	PP+
UNIFORM	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+

Figure 5: Join time slowest

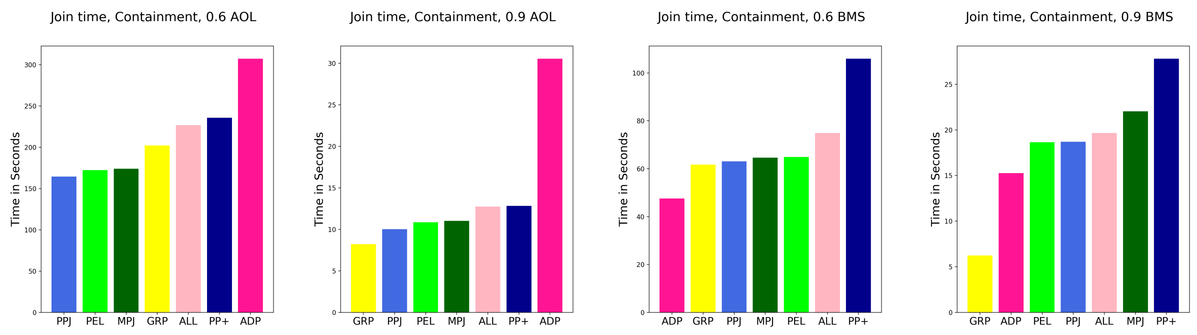


Figure 6: AOL and BMS Join time at 0.6 and 0.9 threshold

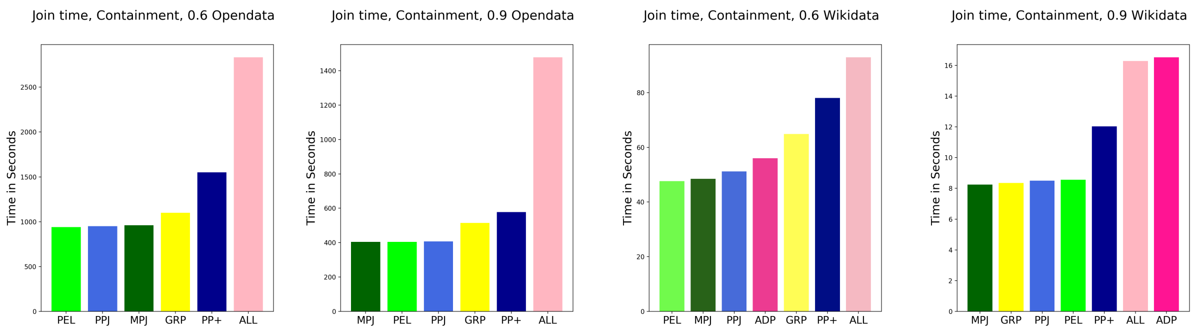


Figure 7: Open data and Wikidata Join time at 0.6 and 0.9 threshold

5.1.3 Join Time - Slowest

The length filter does not have an upper bound. Also, the prefix filter is also affected by the containment similarity score because for containment we have more tokens being indexes for each input domain. All Pairs lack a pre-candidate set filter and thus would generate the largest number of candidate to verify. Figure 5 gives the summary of slowest algorithms for each case.

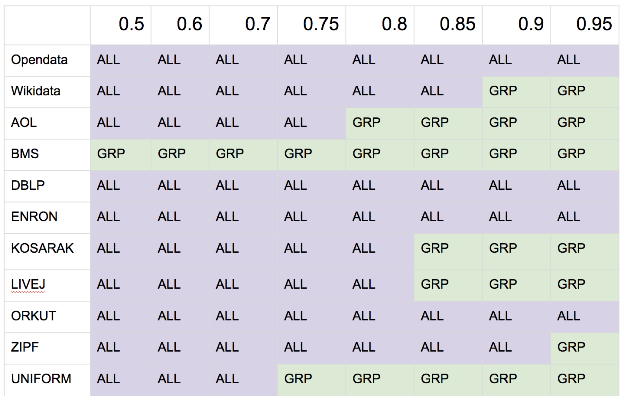
Adaption join does not work well for AOL and KOSARAK because those two datasets have small average set sizes compared to a dataset like ORKUT and open data. AOL has an average set size of 3 and Kosarak has an average set size of 12 while open data and ORKUT have average set sizes of over 100. When average set size is very small, the effect of dynamical prefix size is not huge. Also, AOL and KOSARAK have a huge number of different tokens which leads to longer candidate generation time which does not pay off in verification time. AdaptJoin maintains multiple inverted lists per tokens so if there are a large number of different tokens, the performance of AdaptJoin will decrease.

PP+ applies the suffix filter to pre-candidates. Unfortunately, the suffix filter is too slow compared to verification and never pays off [MAB16]. Length filter for containment does not have upper bound which also decreases the performance of PP+ because of the longer suffix.

In summary, for large thresholds, GroupJoin would usually be the good choice because large threshold leads to many similar prefixes. For most of the datasets, AdaptJoin would work well except the dataset that has many different tokens and short average set size. PPJoin, MPJoin, MPJoin-PEL have the similar performance, and they are a safe choice when the threshold and distribution of datasets are not determined. PPJoin+ and Allparis are the worst choices.

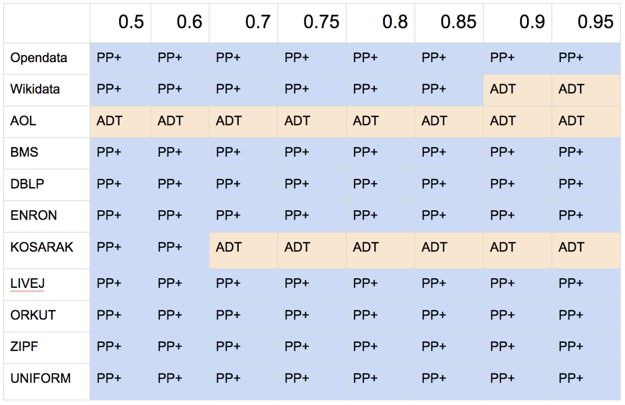
5.2 Candidate Time

AllParis and GroupJoin are the fastest at candidate generation. Figure 8 and Figure 9 gives the summary of the fastest and slowest candidate generation algorithm in each case. GroupJoin processes the sets with the same prefix together, resulting in shorter candidate generation time [MAB16]. AdaptJoin is the slowest on some datasets because those datasets have small average set sizes and a large number of different tokens. Building additional extended prefix indexes on these datasets are slow [MAB16]. PPJoin+ is slow at candidate generation because the suffix filter is too complex and not effective when the length filter does not have an upper bound.



	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
Opendata	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
Wikidata	ALL	ALL	ALL	ALL	ALL	ALL	GRP	GRP
AOL	ALL	ALL	ALL	ALL	GRP	GRP	GRP	GRP
BMS	GRP	GRP	GRP	GRP	GRP	GRP	GRP	GRP
DBLP	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
ENRON	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
KOSARAK	ALL	ALL	ALL	ALL	ALL	GRP	GRP	GRP
LIVEJ	ALL	ALL	ALL	ALL	ALL	GRP	GRP	GRP
ORKUT	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
ZIPF	ALL	ALL	ALL	ALL	ALL	ALL	ALL	GRP
UNIFORM	ALL	ALL	ALL	GRP	GRP	GRP	GRP	GRP

Figure 8: Candidate time fastest



	0.5	0.6	0.7	0.75	0.8	0.85	0.9	0.95
Opendata	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
Wikidata	PP+	PP+	PP+	PP+	PP+	PP+	ADT	ADT
AOL	ADT	ADT	ADT	ADT	ADT	ADT	ADT	ADT
BMS	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
DBLP	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ENRON	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
KOSARAK	PP+	PP+	ADT	ADT	ADT	ADT	ADT	ADT
LIVEJ	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ORKUT	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
ZIPF	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+
UNIFORM	PP+	PP+	PP+	PP+	PP+	PP+	PP+	PP+

Figure 9: Candidate time slowest

5.3 Lookups, Pre-Candidates and Candidates

The number of lookups is identical for AllPairs, PPJoin, PPJoin+, MPJoin, and MPJoin-PEL [MAB16]. The first chart on Figure 10 shows the number of lookups for BMS dataset at threshold equals to 0.8. GroupJoin has fewer lookups because sets with identical prefixes are grouped and treated as a single set during candidate generation [MAB16]. AdaptJoin has the largest number of lookups because the adaptive prefix index of ADP supports different prefix sizes. Instead of one inverted list per token, multiple inverted lists (one for each prefix size) are stored in the index, so it requires additional index lookups [MAB16].

Allpairs, PPJoin, PPJoin+, MPJoin and MPJoin-PEL produce different numbers of pre-candidates, but the following relationship holds: $PEL \subseteq MPJ \subseteq PPJ = PP+ = ALL$ [MAB16]. The second chart on Figure 10 shows the number of pre-candidates for the BMS dataset at threshold equals to 0.8. MPJoin and MPJoin-PEL reduce the number of pre-candidates by removing hopeless inverted list entries from the index [MAB16]. MPJoin-PEL, also, leverages the matching position to crop the lists better, but it is not effective on containment because there is no upper bound for the length filter. GroupJoin generates fewer pre-candidates due to the grouping effect at a high threshold. However, when the threshold decreases, grouping is not that effective [MAB16]. In the absence of duplicate prefixes, GroupJoin behaves like PPJoin [MAB16]. AdaptJoin does more lookups and thus processes more pre-candidates.

The number of candidates is always smaller than (or equal to) the number of pre-candidates, and the following relationship holds: $GRP = PEL = MPJ = PPJ \subseteq ALL$ [MAB16]. The last two charts in Figure 10 shows the number of candidates for Wikidata and the BMS dataset at threshold equals to 0.8. The complex filters for AdaptJoin and PPJoin+ lead to small candidate sets [MAB16]. Lack of filters for AllPairs leads to large candidate size, therefore increasing the verification time.

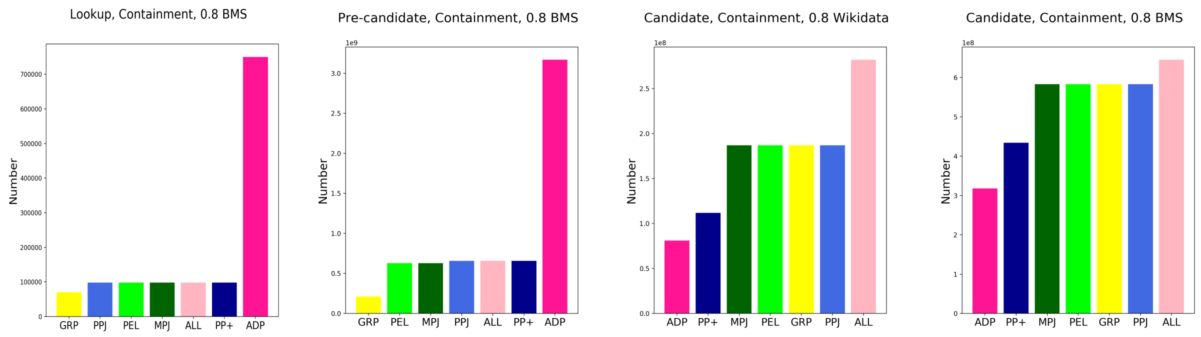


Figure 10: BMS Lookups, BMS Pre-Candidate and Wikidata and BMS Candidates

5.4 Effect of Duplication

Duplicate sets in the dataset lead to more lookups, pre-candidates, candidates and output pairs. Thus, duplicate sets would increase the join time. For the Karsarak dataset, removing duplicate sets reduces the dataset size by 10%. Duplicate sets do not have a big impact on GroupJoin thanks to the grouping [MAB16]. Figure 11 shows the join time and candidate time for Kosarak dataset with duplicates and without duplicates. However, the other six algorithms suffer a decrease in performance from duplicate sets.

5.5 Length Shuffled

We compare the join time for collections that are lexicographically sorted vs. length shuffled in Figure 12. Our original data has already been sorted. Therefore we have the data length shuffled to measure the effect of sorting. Data with the same length could be sorted differently because of length shuffle [MAB16], and all algorithms benefit from sorting. The join times are largely reduced with sorting compared with length shuffled, especially for GroupJoin. The improvement in sorting compared to length shuffling is larger for wikidata than opendata. Sorting is more beneficial for wikidata because the average size is only 18, and there are a significant amount of sets with the same length. However, for open data, the size distribution

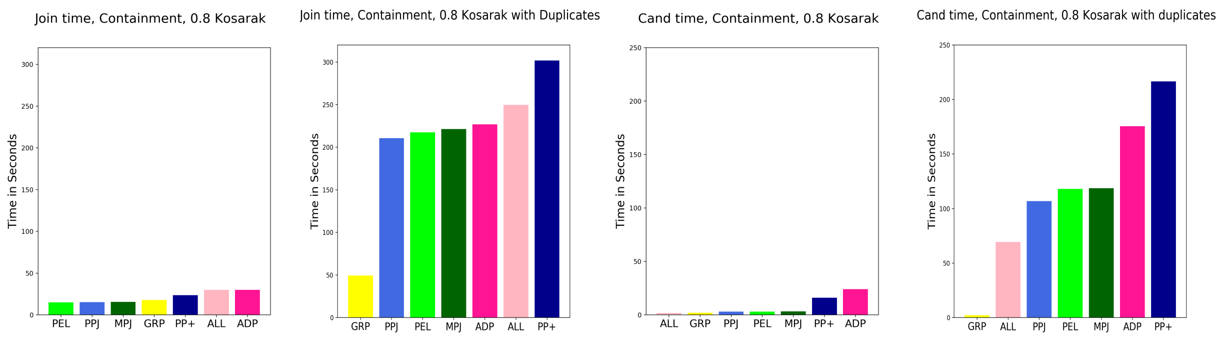


Figure 11: Kosarak dataset join time and cand time with and without duplicates

has a large standard deviation. Row lengths are less likely to be the same, so there are fewer changes after length shuffle.

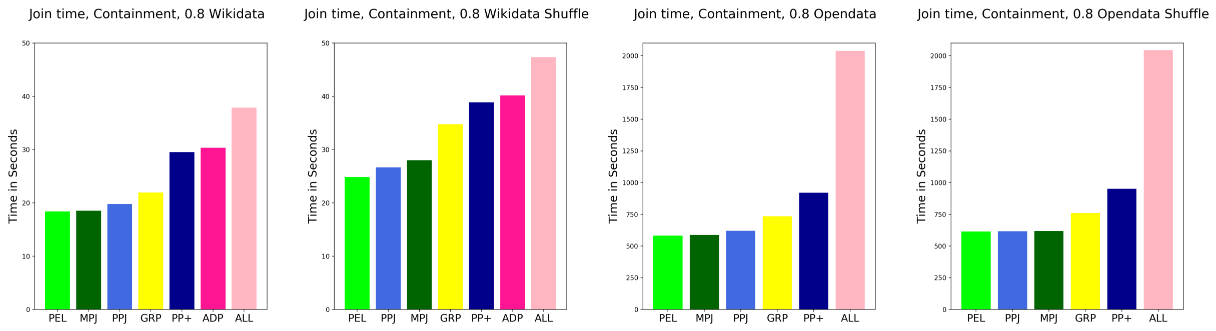


Figure 12: Open data and Wikidata Join time with and without duplicates

5.6 Memory Usage

Besides runtime, memory usage is another important factor for algorithms. Figure 13 gives memory usage for ENRO, wIkidata, and open data at threshold equals to 0.8. AllPairs join uses the smallest memory because it doesnt need position data [MAB16]. AdaptJoin uses the largest memory due to its multiple inverted lists per token [MAB16]. Please note that ADP join did not finish running on open data due to memory error. The memory usage we recorded when running ADP join on open data is roughly 175 GB right after finishing indexing in building the inverted lists.

When we considered real-world applications, memory usage and runtime are both important in evaluating an algorithm. For example, AdaptJoin outperforms other algorithms in runtime for most datasets and thresholds, but it uses much larger memory. In some practical cases, users may not care about runtime but have limitations in hardware. Then it is reasonable to choose other algorithms.

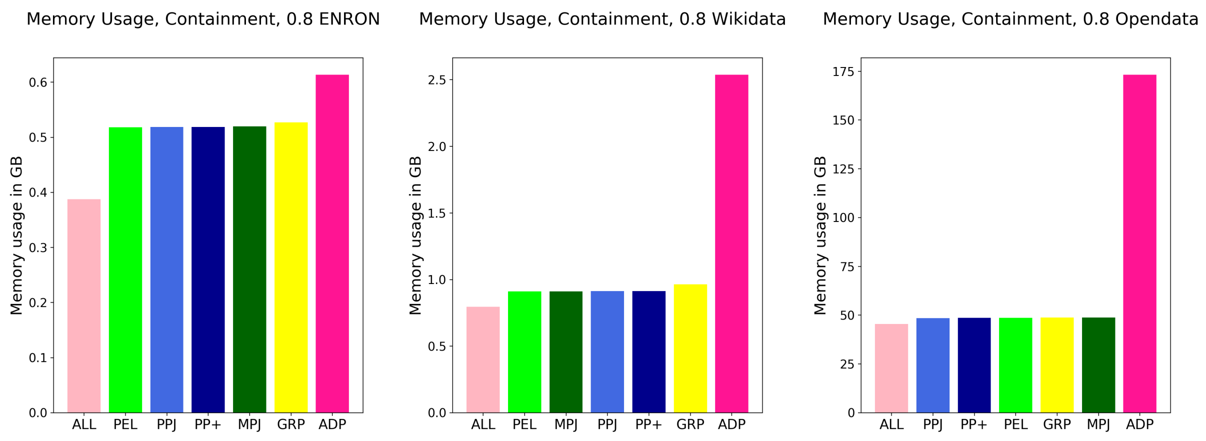


Figure 13: Open data and Wikidata Join time

6 Conclusion

We have studied seven recent set similarity join algorithms, whose core techniques are based on the filter-verification framework. We have re-implemented all seven tested algorithms on the containment similarity function. Experiments on different datasets showed AllPairs to have the terrible performance because the prefix filter and length filter are not effective on containment threshold. PPJoin+ also has the terrible performance due to the inefficient suffix filter. ADP has the best performance thanks to its complex extended prefix filter. GroupJoin has the best performance when there are many sets with a similar prefix. GroupJoin has the similar performance as using the Jaccard similarity function. MPJoin, PPJoin, MPJoin-PEL have the average performance. The result presented in this paper is different from the original work [MAB16] where Allpairs used to be the one of the fastest algorithms and AdaptJoin was the slowest algorithm in many cases. This paper presents the opportunities to improve the exact search algorithms for the containment similarity function.

7 Future Works

For future work, we would like to fix the memory issue for AdaptJoin for large input. Also, we would like to add functionality to the scraper to handle row spans. Additionally, we wish to use the wikitable available in JSON format on the Wikimedia page. We may want to use the dataset and compare it with the wikitables we extracted from HTML pages. In this paper, a query has roughly same distribution as the input domains, but this is not often the case. We could consider the impact of query distribution on the performance. Currently, the implementation of algorithms only makes use of one core. Distributed computing and parallel computing are the most obvious ways to decrease runtime.

The original SSJoin paper [MAB16] was designed for similarity function like Jaccard, Cosine, and Dice that are symmetric. Containment index is not symmetric, so the performance is not optimized. For optimizing containment on exact set similarity methods, we could take account of the distribution of query size for the indexing purpose. For example, if the min query size is 10 and containment threshold is one then we don't have to index all of the tokens if the input domain size is 20, indexing first 11 would be sufficient. We could also use the techniques introduced in Asymmetric Minwise hashing [SL15] that pads values to the domain to make them have the same size. After that, we could transform the containment to the Jaccard similarity function, and proceed as Jaccard similarity search.

References

[BGM12] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment*, 6(1):1–12, 2012.

[BMS07] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Data Engineering, 2006. ICDE'06. proceedings of the 22nd International Conference on*, pages 5–5. IEEE, 2006.

[DDH01] AnHai Doan, Pedro Domingos, and Alon Y Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *ACM Sigmod Record*, volume 30, pages 509–520. ACM, 2001.

[IFF$^+$99] Zachary G Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S Weld. An adaptive query execution system for data integration. In *ACM SIGMOD Record*, volume 28, pages 299–310. ACM, 1999.

[MA14] Willi Mann and Nikolaus Augsten. Pel: Position-enhanced length filter for set similarity joins. In *Grundlagen von Datenbanken*, pages 89–94, 2014.

[MAB16] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647, 2016.

[RH11] Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62–78, 2011.

[SL15] Anshumali Shrivastava and Ping Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *Proceedings of the 24th International Conference on World Wide Web*, pages 981–991. International World Wide Web Conferences Steering Committee, 2015.

[wik] Wikimedia downloads.

[WLF12] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 2012.

[XWL$^+$11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.

[ZNPM16] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. Lsh ensemble: internet-scale domain search. *Proceedings of the VLDB Endowment*, 9(12):1185–1196, 2016.