1 Evaluations

As shown in Table 1, approximate matching is theoretically faster (and consumes less memory) than exact matching based approaches. However, the impact of this improvement on the final (end-to-end) execution time is determined by the amount of time spent in matching within the compute intensive *verification phase*, which performs the crucial task of calculating the similarity between two sets.

Table 1: Matching complexity per set pair of the comparing approaches. n is the number of vertices of the bipartite graph generated from the pair (R, S). The memory complexity of AJ–PS assumes the pairs of sets are generated one by one.

	Variants	Approx.	Time	Memory
TokenJoin (TJPJ) [7]	Hungarian (TJPJ–HG)	1	$O(n^3)$	$O(n^2)$
	Efficient Verification (TJPJ–EV)			
ApproxJoin (AJ) [this work]	Locally Dominant (AJ–LD)	0.5	$O(n^2 \log n)$	$O(n^2)$
	Greedy (AJ–GD)			
	Paz Schwartzman (AJ–PS)	0.5	$O(n^2)$	$O(n \log n)$

We have extended the open-source Python codebase of a state-of-the-art implementation of set similarity join (that uses exact matching), TokenJoin [7].[1] Our code is available and provided. For our experiments, we use the TokenJoin methods for the candidate generation and refinement with the Positional and Joint filters (TJPJ) with a similarity threshold of $\delta = 0.7$. We note that all of our experimental runs are self joins on the respective dataset. In instances where $|\mathcal{R}| \neq 100\%$, a random sample at the given size is generated and used for all methods at that size. Our tests focus on Jaccard similarity (JAC) with an additional test using Normalized Edit similarity (NEDS) to accommodate direct string comparison. Depicted execution times are averaged over several runs with exclusive allocation on the testbed platform.

In the rest of this section, we empirically compare approximate matching (our work) with exact matching (existing state-of-the-art, TokenJoin) based approaches for set similarity join, after briefly discussing the datasets and experimental platform. §1.1 sets the expectation in terms of the achievable performance improvements across varied datasets. In §1.2, we discuss baseline performances in terms of the overall execution and verification times. We discuss the accuracy of our approximate matching based implementations in §1.3, and establish trade-offs to improve the overall accuracy. We perform extra experiments in §2.1 and §2.2, showing that our approaches consume less memory (29% on average), alongside the parallelism potential of our approaches at a baseline of 8× improvement.

[1]https://github.com/alexZeakis/TokenJoin/tree/main

Datasets. We experiment on 10 different datasets, whose characteristics are highlighted in Table 2. Each token is generated by splitting words into q-grams ($q = 3$) and substituting an integer for each q-gram. For the purpose of results classification, we refer to *dense* instances as datasets with >100 element per set averages or >1,000 maximum element per sets. We use Jaccard (JAC) and/or Normalized Edit (NEDS) similarity measures.

Table 2: Experimental datasets and their characteristics.

Datasets	Elements ∈ Set	#Sets	Ele/Set	
			Avg	Max
LIVEJ [4][2]	interests ∈ user	3.1M	36	300
AOL [5][3]	keywords ∈ search	1.8M	3	73
KOSARAK [1][4]	links ∈ user	610K	12	2.5K
ENRON [7][5]	words ∈ email	518K	134	3.2K
DBLP [7] [5]	authors ∈ work	500K	13	189
FLICKR [7] [5]	words ∈ photo	500K	9	361
GDELT [7] [5]	topics ∈ article	500K	19	396
BMS-POS [3][6]	items ∈ sale	320K	6	164
YELP [7] [5]	types ∈ business	160K	6	47
MIND [7] [5]	words ∈ article	123K	32	357

Platforms. Our baseline results are collected using a single thread on a system with 2TB of DDR4 RAM and dual-socket NUMA AMD EPYC 7742 2.2GHz 64-core processors and 2 threads/core with Ubuntu version 20.04.4 LTS O/S. We use Python version 3.12.9 and TJPJ–HG uses NetworkX version 3.4.2. For the preliminary parallel experiments (Appendix 2.2), GNU C++ compiler (v13.3) with OpenMP 5.1 were used.

1.1 Timing Breakdown

Using 50K samples/dataset, we show in Fig. 1, the percentage breakdown of the different components from the TJPJ workflow, comprising of initialization, candidate pairing generation, refinement and verification (deduplication, graph building and subsequent matching combined). For Fig. 1, we consider the datasets for which the verification phase requires more than 5% of the overall execution time. We observe high variations in the verification related computations (between <1–94% of the overall execution times). Within the verification phase, matching is typically the most computationally expensive component. Generally larger datasets with higher number of elements/set tend to spend more time on matching, with the exception of the MIND and GDELT datasets. This disparity can be explained by examining the average number of elements/set of the candidate pairings that proceed through the verification phase. For instance, GDELT with $|\mathcal{R}| = 5\text{K}$ produces up to 2M candidate pairings, out of which only 11K proceed through the verification following refinement. Within the 11K pairings, average elements/set is about 4; the resultant graph will have low-degree edges, significantly reducing the matching overheads.

[2]http://socialnetworks.mpi-sws.org/data-imc2007.html
[3]https://www.cim.mcgill.ca/ dudek/206/Logs/AOL-user-ct-collection/
[4]http://fimi.uantwerpen.be/data/
[5]https://github.com/alexZeakis/TokenJoin/tree/main
[6]https://www.kdd.org/kdd-cup/view/kdd-cup-2000

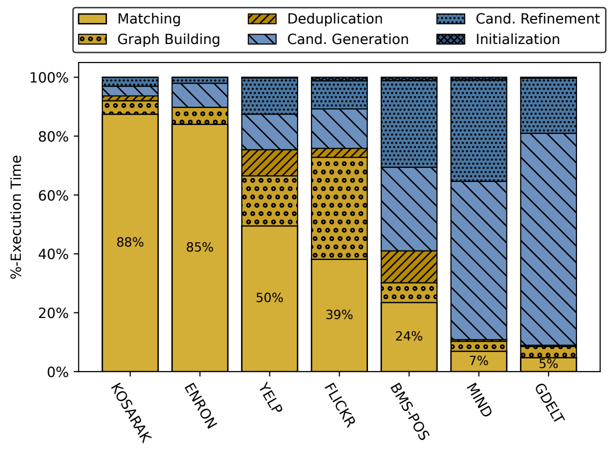

Figure 1: Execution time distribution for runs of TJPJ–HG at $|\mathcal{R}| = 50K$. The annotated percentage is the exact value for matching, rounded up. Verification includes Matching, Deduplication and Graph Building steps.

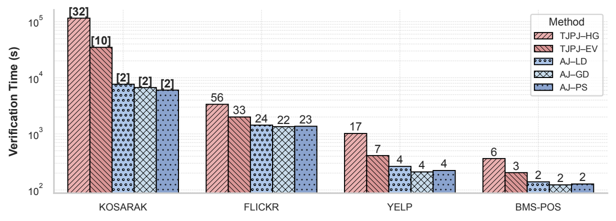

Figure 2: JAC verification execution time comparison per matching method, $|\mathcal{R}| = 100\%$. The [bold] annotations depict time in hours, while the rest depict time in minutes, rounded up to the nearest whole number.

1.2 Baseline Performance

In this section, we discuss performance results of the various approximate matching methods compared to exact Hungarian and Efficient-Verification in original TokenJoin (see Table 1), in terms of the total execution and verification times using Jaccard (JAC) and NEDS similarity measures (in §1.2.1, §1.2.2 and §1.2.4). We also discuss scalability (§1.2.3), effect of threshold variation (§1.2.5) and finally summarize the performance aspects in §1.2.6.

1.2.1 ***JAC verification time:*** Fig. 2 exhibits the execution time of the verification phase using Jaccard similarity within TJPJ ($\delta = 0.7$) at $|\mathcal{R}| = 100\%$. In all instances, the approximate methods yield faster verification than the original TJPJ–HG, with the most improvements in KOSARAK and YELP, with 19.1× improvement using AJ–PS and 4.8× using AJ–GD, respectively. Across all datasets, we see a geometric mean improvement over TJPJ–HG of 4.5× for AJ–PS and AJ–GD, with 3.9× for AJ–LD. The arithmetic mean for each is 6× for AJ–PS, 5.7× for AJ–GD and 5× for AJ–LD. Compared to TJPJ–EV, we continue to observe faster execution times for the approximation methods. The biggest improvement is seen using AJ–PS on KOSARAK, with 5.8× performance improvement relative to TJPJ–EV, with the next best being DBLP with 2.3× improvement with AJ–PS. On an average, AJ–GD depicts 2.4× improvement, AJ–LD demonstrates 2.1× improvement and AJ–PS shows 2.5× improvement, relative to TJPJ–EV. In terms of the verification times, AJ–GD performs the best in 3/6 cases, while AJ–PS performs equally better for the rest. Individual differences between the approximation methods is marginal in most cases, except for KOSARAK (where we observed the best improvement using AJ–PS, about 19× relative to TJPJ–HG), where the most time difference between AJ–GD and AJ–PS is about a quarter of an hour. Otherwise, the geometric mean percentage difference of maximum and minimum verification times between proposed approximation methods is about 16%. Thus, we see a wide range (1.9–19.1×) of performance improvements in the verification process when comparing the approximation methods to TJPJ–HG and to TJPJ–EV.

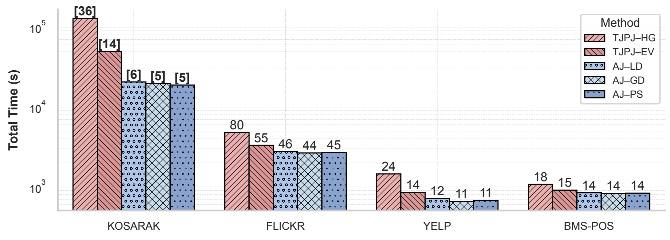

Figure 3: JAC total execution time comparison per matching method, $|\mathcal{R}| = 100\%$. The [bold] annotations depict time in hours, while the rest depict time in minutes, rounded up to the nearest whole number.

1.2.2 ***JAC total execution time:*** We show the total execution time of the fuzzy set similarity join workflow in Fig. 3. In every case, approximation methods outperform TJPJ–HG and TJPJ–EV, with performance improvements being 3.78× compared to TJPJ–HG and 2.18× vs. TJPJ–EV on average. Not surprisingly, cases where the verification time is relatively high, we observe substantial impact on the total execution time (refer to §1.1). Specifically, KOSARAK (94% of the total time spent in verification) sees the highest improvements relative to TJPJ–HG: 6.5× with AJ–GD, 6.2× with AJ–LD and 6.7× with AJ–PS, respectively. In terms of actual execution times, this equates to a difference of more than a day's worth of computing (about 30 hours)! Depending on the dataset density and the verification times, the improvements are more impactful for the larger instances. YELP exhibits the second highest performance, depicting 2.1× improvement in the execution times across the approximate methods. In general, we observe about 2.2× improvement in the (total) execution times across the datasets against TJPJ–HG. On the other hand, compared to TJPJ–EV, there is a 2.5× improvement for KOSARAK and an average 1.4× improvement for the approximation methods considering rest of the datasets. Thus, we see a range of 1.4–6.7× performance improvement with the approximate methods (AJ–LD, AJ–GD and AJ–PS) as compared to exact TJPJ–HG and TJPJ–EV.

1.2.3 ***JAC execution time scaling:*** In Fig. 4, we perform scaling experiments on a subset of our datasets to assess the performance as the data sizes increase, taking random samples of each dataset at 20% size intervals. Across every instance, we observe improved scaling trends with reduced execution times for the approximation methods at each size interval. Best performance is achieved at 80%–100% for FLICKR, 4.9× better performance using AJ–GD than TJPJ–HG, and, 2.5× using AJ–PS as compared to TJPJ–EV. On average, the approximation methods shows 3.4× improvement relative to TJPJ–HG and 1.8× improvement against TJPJ–EV. We see AJ–GD outperforming the rest in 16/25 instances, with AJ–PS exhibiting better performance for the remaining 9/25, with an average %-difference of 14% in terms of execution times among the approximation methods.

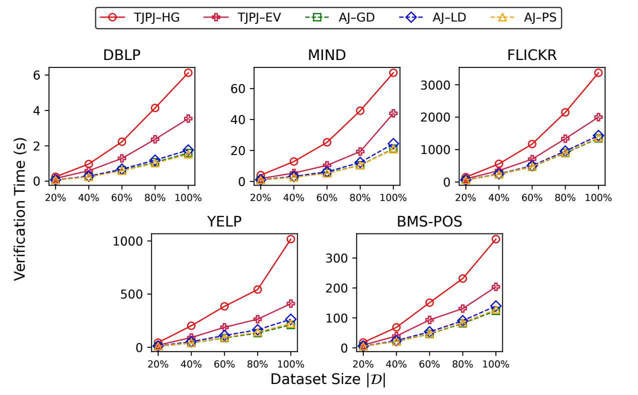

Figure 4: JAC verification execution time scaling per set size (lower is better). Average improvement of 3.4× vs. TJPJ–HG and 1.8× vs. TJPJ–EV.

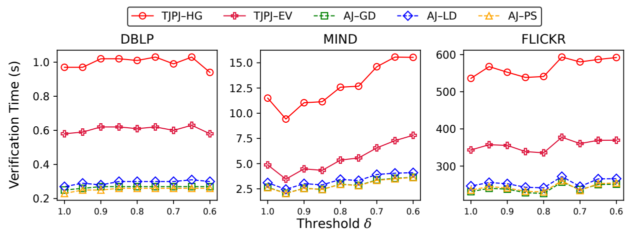

Figure 5: JAC threshold variation verification execution time scaling (lower is better). Average performance improvement of 3.4× vs. TJPJ–HG and 1.8× vs. TJPJ–EV.

1.2.4 ***NEDS verification time:*** We also test the performance using NEDS on a subset of our datasets due to significantly higher execution times as compared to JAC. For this scenario, we report the verification times on $|\mathcal{R}| = 20\%$ using TJPJ ($\delta = 0.7$) as our default. The results are reported in Fig. 6. We observe approximate matching cases outperforming the default across every instance, with KOSARAK seeing the most improvement of 10.2× using AJ–GD compared with TJPJ–HG. Consequently, AJ–GD depicts the highest performance improvement of 2.7× against TJPJ–EV. Across every instance, we observe an arithmetic mean improvement of 3.7× against TJPJ–HG and 1.8× as compared to TJPJ–EV. In general, AJ–GD performs the best across all the 5 instances of the approximation methods (as shown in Fig. 6), depicting a performance variation of up to 14% among the approximation methods considered.

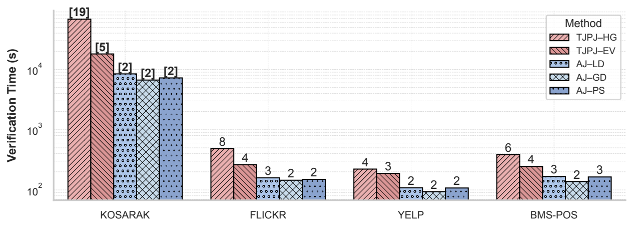

Figure 6: Verification time using NEDS at $|\mathcal{R}| = 20\%$. 3.7× improvement on average against TJPJ–HG and 1.8× against TJPJ–EV. The [bold] annotations depict time in hours, while the rest depict time in minutes, rounded up to the nearest whole number.

1.2.5 ***JAC threshold variation:*** We also experiment with varying δ to ensure execution time improvements are consistent relative to candidate set sizes. We present only the significant results in Fig. 5. Throughout the datasets, we observe consistent execution time trends with constant performance improvement gaps between the TJPJ–HG and TJPJ–EV methods, and the approximate methods. The highest performance improvement on average is observed at the lower values of δ, as there are more candidate pairings for verification improvement. Considering the approximate methods, we observe on an average 3.4× improvement upon TJPJ–HG across all δ values and 1.8× improvement upon TJPJ–EV. Among the 45 instances considered, AJ–GD performed the best in 27 while AJ–PS performed the best in the other 18.

Table 3: Summary of relative performance improvements (approx. vs. exact) as geometric mean: higher is better.

Method	Verification Time		Total Time	
	HG	**EV**	**HG**	**EV**
AJ–LD	4.27×	2.00×	2.43×	1.46×
AJ–GD	4.70×	2.20×	2.49×	1.51×
AJ–PS	4.88×	2.29×	2.56×	1.55×

1.2.6 ***Performance summary:*** The overall performance improvements observed by the approximation methods relative to the exact state-of-the-art are summarized in Table 3 and Fig. 7. Table 3 shows the geometric mean of performance improvements of approximate methods relative to exact counterparts. We observe an average of 2× improvement in favor of approximate methods against both TJPJ–HG and TJPJ–EV for the total execution times, with 2–4× improvement for the verification times. The relative differences between verification and total time improvements relies on the

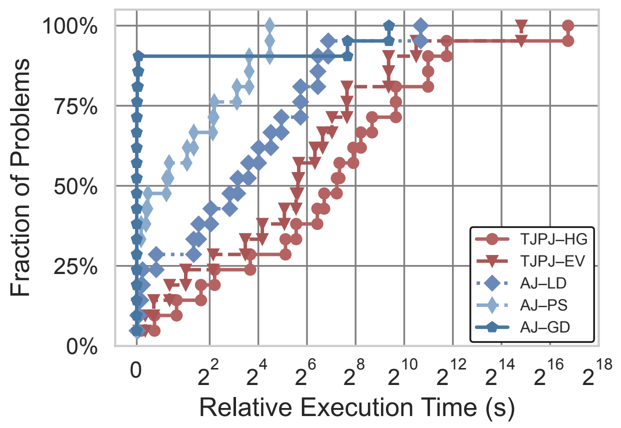

Figure 7: A performance profile to summarize the execution time differences between ApproxJoin methods compared to TokenJoin. The X-axis (log scale) represents the execution time in seconds, normalized by the best performing algorithm for a given problem. We normalize the time by subtracting the time taken by the best performing method from all the methods. The Y-axis represents the fraction of the problems. The closer to the left, the better performance.

overall distribution of the verification times, as discussed in §1.1 and §1.2.2.

Fig. 7 shows the compiled relative performance of all of our test instances, relative to the best method as a difference of execution time. For each test instance, the best algorithm (i.e., the algorithm with the least execution time) is set to zero, and all others' execution time is offset from the best one. Approximate methods outperform the exact methods in every instance. In general, we see the strongest performance results being competitive between AJ–PS and AJ–GD, with a slight performance advantage on average for AJ–PS despite AJ–GD having the best execution times in a majority of cases. We expand on the accuracy benefits of the approaches in the forthcoming section, §1.3.

1.3 Approximate Matching Accuracy

Minor variations in the matching weights are expected for the approximate methods; we consider the exact Hungarian method as our ground truth in assessing the accuracy of approximate matching based verification. Since both TJPJ–HG and TJPJ–EV produces the same matching, their qualities are exactly similar. We calculate the resultant set size of the approximate matching verification and compare that to the default, checking for discrepancies in the set sizes (Δ#Sets) to calculate recall and precision metrics. We define recall and precision relative to our ground truth based on standard true positives (TP), false positives (FP) and false negatives (FN) formulations:

$$Recall = \frac{TP}{TP + FN},\ Precision = \frac{TP}{TP + FP}.$$

We denote TP as instances in the resultant set for both the Hungarian and approximate matching verifications, FP as the extra instances not found in the original (resultant) set, and FN as the missing instances.

1.3.1 ***Matching weight based accuracy:*** We present the default results in the first half of Table 4, using the *lowest* values from 3 runs of random samples of $|\mathcal{R}| = 50$K. From these runs, we conclude that the usage of approximate matching yields recall values >0.9 across all instances, with all datasets having at least 0.99 recall besides ENRON. We note that we have excluded the precision as all datasets yielded a precision of 1.0. This implies that our datasets are subsets of the dataset yielded by the TJPJ–HG method with an average of approximately 570 pairings missing for AJ–GD and AJ–LD, and 120 pairings for AJ–PS. AJ–PS also shows the highest recalls among all methods, having the highest value when not tied by the other approximation methods for all 10 datasets. By geometric mean, we see an average recall of **0.9824, 0.9822** and **0.9974** for AJ–GD, AJ–LD and AJ–PS, respectively. We attribute the exception of ENRON statistics to its high matching usage (about 87%) alongside a large amount of candidate sets (about 70K) and relatively high dataset density (average 133 elements/set). Nevertheless, we can obtain significant performance improvements of up to **6**× while maintaining a majority of the resultant pairings with **99%** recall.

1.3.2 ***Upper bound based accuracy:*** To improve the recall metric using approximate matching, we experiment with trade-offs regarding accuracy by using upper bounds generated from the approximate matching for verification. Since AJ–GD and AJ–LD are half-approximate solutions, we simply double the matching weights, whereas for the AJ–PS method, we use the sum of ϕ from the primal-dual basis. We report the lowest values from 3 runs of random samples considering $|\mathcal{R}| = 50$K in the second half of Table 4. We omit the associated recall values, as every dataset instance led to a 100% recall, implying that the resultant sets include the default/exact method's pairings, in addition to new pairings. This inclusion of the extra pairings is represented by the precision metric for each method (in Table 4); geometric mean of precision across the approximate matching methods are **0.9978, 0.9978** and **0.9980** for AJ–GD, AJ–LD and AJ–PS, respectively. We observe that shifting the bounds leads up to a thousand extra pairings in the resultant dataset while including the ones missed in the first portion of Table 4. On an average, AJ–GD and AJ–LD produces 210 extra pairings while AJ–PS yields 191 across the datasets considered. Nevertheless, we observe high precision values (about 0.98) using proposed approximate matching methods within verification. Based on these accuracy assessments, we can consider adapting the approximate matching methods to yield more or less pairings in the resultant dataset. Specifically, if retaining the pairings generated by an exact method such as TJPJ is essential, then shifting the bounds can induce the necessary pairings without affecting the overall performance (experiments indicate less than 2% performance loss upon expanding the approximation bounds). However, this type of adaptation would not be necessary if the application can withstand minor loss of pairings, i.e., taking a mild hit at the accuracy. Regardless of the choice, we demonstrate significant performance improvements of **1.2–6.1**× with high and acceptable accuracy values.

Table 4: Accuracy assessment of our approximate matching based approach, $|\mathcal{D}| = 50K$; Recall/Precision closer or equal to 1 is ideal.

Dataset	Matching Weight based (ApproxJoin)						Upper bounds based (ApproxJoin)					
	Δ#Sets			Recall			Δ#Sets			Precision		
	GD	LD	PS	GD	LD	PS	GD	LD	PS	GD	LD	PS
LIVEJ	0	0	0	1.0000	1.0000	1.0000	+2	+2	+1	0.9998	0.9998	0.9999
AOL	0	0	0	1.0000	1.0000	1.0000	0	0	0	1.0000	1.0000	1.0000
KOSARAK	-1	-7	-1	0.9999	0.9999	0.9999	+4	+4	+4	0.9999	0.9999	0.9999
ENRON	-4515	-4637	-593	0.9293	0.9274	0.9757	+835	+835	+778	0.9871	0.9871	0.9879
DBLP	0	0	0	1.0000	1.0000	1.0000	+1	+1	+1	0.9870	0.9870	0.9870
FLICKR	-115	-115	-91	0.9994	0.9994	0.9995	+114	+114	+109	0.9994	0.9994	0.9995
GDELT	-937	-937	-481	0.9992	0.9992	0.9996	+843	+843	+806	0.9992	0.9992	0.9993
BMS-POS	0	0	0	1.0000	1.0000	1.0000	0	0	0	1.0000	1.0000	1.0000
YELP	-47	-47	-80	0.9999	0.9999	0.9998	+293	+293	+199	0.9999	0.9999	0.9999
MIND	-38	-38	-8	0.9995	0.9995	0.9999	+14	+14	+13	0.9998	0.9998	0.9998

Appendix

2 Memory and Parallelization Experiments

2.1 Memory Usage Analysis

To analyze the memory usage patterns of the approximation and exact methods, we assess the average and peak memory consumption, and examine memory consumption throughout the program execution, in upcoming §2.1.1 and §2.1.2.

2.1.1 ***Peak memory usage:*** Given that AJ–PS does not require the entire graph being built in advance, we compare its peak memory usage relative to the exact TJPJ–HG and TJPJ–EV methods. For peak memory analysis depicted in Fig. 8, we run each method on a 20% sample of a subset of datasets using JAC. We use the built in `tracemalloc` snapshots[7] to obtain peak memory usages. Notably, we observe the highest improvement compared to TJPJ–HG for the denser instances, with up to a 93% and 72% reduction in memory usage on ENRON and KOSARAK, respectively. On average, AJ–PS yields a 34% reduction compared to TJPJ–HG and a 12% reduction against TJPJ–EV, improving the memory usage for every instance considered.

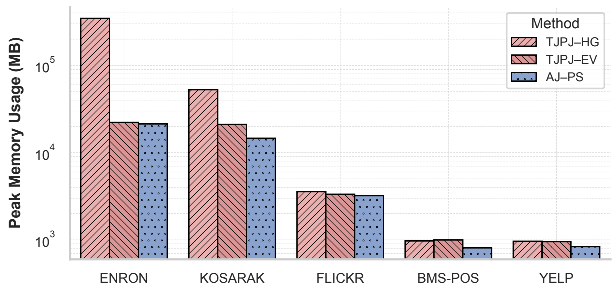

Figure 8: Comparison of memory usage using the integrated AJ–PS verification with the default TJPJ–HG and TJPJ–EV cases at $|\mathcal{R}| = 20\%$ (lower is better). Average memory usage reduction is 23% in AJ–PS compared to TJPJ–HG /TJPJ–EV.

[7] https://docs.python.org/3/library/tracemalloc.html

2.1.2 ***Memory usage across program lifetime:*** In Fig. 9, we compare the lifetime memory consumption (from beginning to end of a particular program run) of the exact and approximate methods on an instance of $|\mathcal{R}| = 10\%$ for KOSARAK (one of the denser instances). We collect up to 500 distinct samples of instantaneous memory usage for TJPJ–HG, TJPJ–EV and AJ–PS using the `valgrind Massif` heap profiling tool[8]. Verification happens in

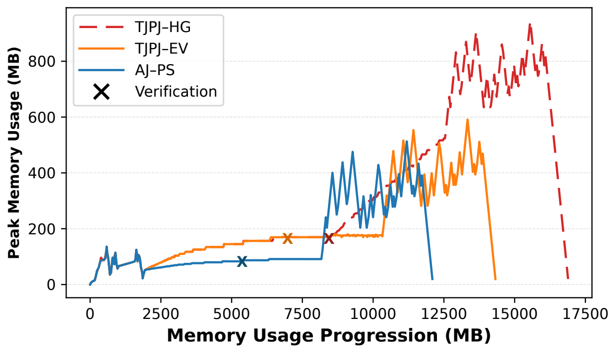

Figure 9: Memory usage progression relative to peak memory usage during a run of TJPJ–HG /TJPJ–EV and AJ–PS using KOSARAK $|\mathcal{R}| = 10\%$. Verification center is marked by "$\times$".

the middle portion of the computation (denoted by "x" in Fig. 9), where AJ–PS shows under 100MB of peak memory usage while TJPJ–HG /TJPJ–EV can consume up to 40% extra memory in this phase. For KOSARAK (best performing dataset for the approximate methods), TJPJ–HG, TJPJ–EV and AJ–PS depicts peak memory usage of 944MB, 591MB and 512MB, respectively; AJ–PS exhibits 14/ 45% better memory footprint.

2.2 Preliminary parallelization of AJ–PS

Designing highly concurrent approximation algorithms is an active research area [6]. As such, there is parallelism potential in the approximate matching methods used in verification. Since AJ–PS is the best performing algorithm in this scenario (both in terms of performance and memory), we develop a "proxy" verification pipeline comprising of a preliminary parallel version of AJ–PS using OpenMP thread-parallelism [2], leveraging randomly generated data to analyze the parallel efficiency across varied data ranges. This "proxy" pipeline simulates the graph building (i.e., edge streaming, in the context of AJ–PS) and matching phases of verification on a set of 500 randomly generated pairings of $\{R, S\}$. Within these randomly generated pairings, we vary the elements/set between 10^2–10^6, imitating diverse dataset densities prevalent on real world datasets.

In terms of parallel implementation, we augment the streaming process of Lines 3 to 8 in Algorithm 2 by performing set similarity calculations (JAC) in parallel and storing the edge similarity information in a buffer. This buffer acts as our stream, where we have a single thread pull the edges sequentially and commit to the stack. Given the potential for hazards while accessing the buffer in

[8] https://valgrind.org/docs/manual/ms-manual.html

Algorithm 1 Set Verification

```
Input: Candidate Sets: $R, S$, Threshold: $\theta_R$
Output: Score: $sim_\phi(R, S)$
1:  ov = $|R \cap S|$                                   ▷ Phase 1: Deduplication
2:  $R_d, S_d$ = deduplication$(R, S)$
3:  if $R_d = \emptyset$
4:      return $\frac{\text{ov}}{(|R|+|S|-\text{ov})}$
5:  $G(V, E, w) = \emptyset$, UB = $|R|$                ▷ Phase 2: Graph Building
6:  for all $r \in R_d$ do
7:      $max_s = 0$
8:      for all $s \in S_d$ do
9:          $score$ = sim$(r, s)$
10:         $max_s$ = max$(max_s, score)$
11:         $V = V \cup r \cup s$
12:         $E = E \cup \{r, s, score\}$                ▷ Add edge to graph
13:     UB = UB $- (1 - max_s)$
14:     if $\theta_R >$ UB
15:         return $\frac{\text{UB}}{(|R|+|S|-\text{UB})}$
16: $W_M$ = ov + matching$(G)$                          ▷ Phase 3: Bipartite Matching
17: return $\frac{W_M}{(|R|+|S|-W_M)}$
```

Algorithm 2 Verification with PS Matching

```
Input: Candidate Sets: $R, S$, Threshold $\theta_R$, constant $\epsilon$
Output: Score: $sim_\phi(R, S)$
1:  $R_d, S_d$, ov = deduplication$(R, S)$
2:  UB = $|R|$; $M \leftarrow \emptyset$; Stk $\leftarrow \emptyset$; $\forall v \in V : \phi(v) = 0$
3:  for all $r \in R_d$ do
4:      $max_s = 0$
5:      for all $s \in S_d$ do
6:          $w(e)$ = sim$(r, s)$                         ▷ $e = \{r, s\}$
7:          $max_s$ = max$(max_s, w(e))$
8:          if $w(e) > (1 + \epsilon)(\phi(r) + \phi(s))$   ▷ Streaming Process
9:              $w'(e) = w(e) - (\phi(r) + \phi(s))$
10:             $\phi(r) = \phi(r) + w'(e); \phi(s) = \phi(s) + w'(e)$
11:             Stk.push$(e)$
12:     UB = UB $- (1 - max_s)$
13:     if $\theta_R >$ UB
14:         return $\frac{\text{UB}}{(|R|+|S|-\text{UB})}$
15: while Stk $\neq \emptyset$ do                        ▷ Post Processing
16:     $e(r, s) \leftarrow$ Stk.pop()
17:     if $(V(M) \cap \{r, s\}) = \emptyset$           ▷ $V(M)$: vertex set covered by $M$.
18:         $M \leftarrow M \cup \{e\}$
19: return $\frac{w(M)+\text{ov}}{(|R|+|S|-(w(M)+\text{ov}))}$
```

multithreaded environments, we implement thread locks for task synchronization.

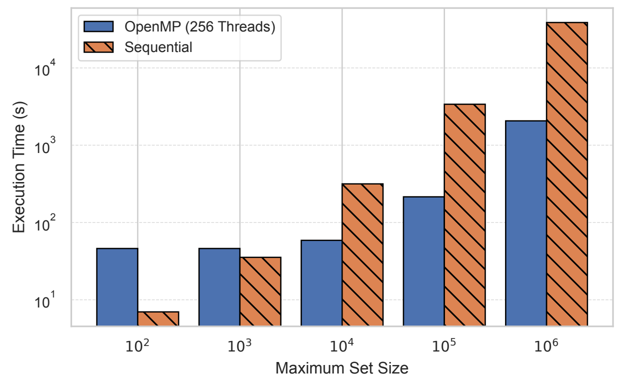

Figure 10: Preliminary execution time results (lower is better) of the proxy OpenMP-based AJ–PS verification (on 256 threads) vs. the sequential, varying the set density.

As shown in Fig. 10, for smaller instances (e.g., set sizes less than 1K), threads are starved (of work), and parallelism overheads outweigh any potential benefits. However, when the set sizes increase, parallel efficiency improves, enhancing the performance by 8× on average as compared to the default serial version on 256 threads (observed maximum speedup of 18× at set size 10^6). Using a single thread for stream update and subsequent lock-based synchronization in this version limits ideal scalability, which could be mitigated by an enhanced parallel implementation, maintaining a stack per thread/vertex, at the expense of extra memory.

References

[1] Ferenc Bodon. 2003. A fast APRIORI implementation. In *Workshop on Frequent Itemset Mining Implementations*. https://api.semanticscholar.org/CorpusID:14392931

[2] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[3] Ron Kohavi, Carla E. Brodley, Brian Frasca, Llew Mason, and Zijian Zheng. 2000. KDD-Cup 2000 organizers' report: peeling the onion. *SIGKDD Explor. Newsl.* 2, 2 (Dec. 2000), 86–93. https://doi.org/10.1145/380995.381033

[4] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*. San Diego, CA.

[5] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search, Vol. 152. 1. https://doi.org/10.1145/1146847.1146848

[6] Alex Pothen, SM Ferdous, and Fredrik Manne. 2019. Approximation algorithms in combinatorial scientific computing. *Acta Numerica* 28 (2019), 541–633.

[7] Alexandros Zeakis, Dimitrios Skoutas, Dimitris Sacharidis, Odysseas Papapetrou, and Manolis Koubarakis. 2022. TokenJoin: Efficient Filtering for Set Similarity Join with Maximum Weighted Bipartite Matching. *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 790–802. https://doi.org/10.14778/3574245.3574263