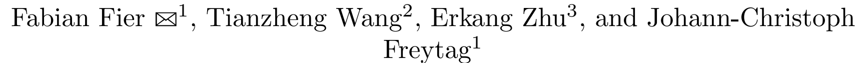
Parallelizing Filter-Verification based Exact Set Similarity Joins on Multicores


Fabian Fier ✉[1], Tianzheng Wang[2], Erkang Zhu[3], and Johann-Christoph Freytag[1]

[1] Humboldt-Universität zu Berlin, Germany
{fier,freytag}@informatik.hu-berlin.de
[2] Simon Fraser University, Canada tzwang@sfu.ca
[3] Microsoft Research, USA ekzhu@microsoft.com

Abstract. Set similarity join (SSJ) is a well studied problem with many algorithms proposed to speed up its performance. However, its scalability and performance are rarely discussed in modern multicore environments. Existing algorithms assume a single-threaded execution that wastes the abundant parallelism provided by modern machines, or use distributed setups that may not yield efficient runtimes and speedups that are proportional to the amount of hardware resources (e.g., CPU cores). In this paper, we focus on a widely-used family of SSJ algorithms that are based on the filter-verification paradigm, and study the potential of speeding them up in the context of multicore machines. We adapt state-of-the-art SSJ algorithms including PPJoin and AllPairs. Our experiments using 12 real-world data sets highlight important findings: (1) Using the exact number of hardware-provided hyperthreads leads to optimal runtimes for most experiments, (2) hand-crafted data structures do not always lead to better performance, and (3) PPJoin's position filter is more effective in the multithreaded case compared to the single-threaded execution.

1 Introduction

The set similarity join (SSJ) operation takes two collections (or a single collection) of records and finds all pairs of records with similarities greater than a user-defined threshold. Many data management problems are modeled as SSJ, such as fuzzy join of two tables on a pair of text columns, record deduplication to remove highly similar near-duplicates, and plagiarism detection to find similar sentences or paragraphs.

In particular, a fruitful line of research work has contributed to speeding up filter-verification based SSJ algorithms [1,3,4,12]. The basic idea is to generate candidate pairs of records, which are a superset of the result set. Computationally cheap *filters* are used to keep the number of candidate pairs far below the size of the cross product of the input collection(s). Then, a *verification* step computes the similarity of each candidate pair. However, filter-verification approaches either use single-threaded or shared-nothing, distributed computing paradigms (e.g., MapReduce [6,11]). Neither approach fully exploits the parallelization potential provided by modern multi-socket multicore machines. Single-threaded

solutions waste the available parallelism on modern hardware. Distributed solutions can use local parallelism by running multiple executors on one node, but assume a shared-nothing architecture that replicates data structures such as inverted indexes multiple times on the same machine, wasting memory and introducing cache inefficiency [6]. Surprisingly, they often cannot compete with single-threaded algorithms in terms of runtime and data set sizes [8].

In this paper, we explore the potential of parallelizing such filter-verification based SSJ algorithms on modern multicore machines. These machines often feature high core counts over multiple processors and exhibit non-uniform memory access (NUMA) in which remote memory access is much slower than local accesses. In this context, we adapt existing single-core SSJ algorithms to become parallel algorithms and discuss the performance impact of various design decisions such as thread placement, filtering and record inlining to improve locality.[4] Experimentally, we show how local parallelization significantly speeds up existing single-threaded approaches, without data replication and the high complexity and cost of managing a cluster of machines. Furthermore, we find some optimizations such as position filters can work even better in parallel SSJ algorithms than in sequential algorithms.

We provide the necessary background in Section 2. We then describe our approach to parallelizing SSJ algorithms in Section 3, followed by experimental evaluation in Section 4. Section 5 concludes this paper.

2 Background

In this section, we first define the exact SSJ problem formally and survey state-of-the-art algorithms and related work that parallelizes SSJ on different hardware platforms. We then give background on the hardware platform we target at, i.e., multi-socket, multicore servers with large main memory.

2.1 Exact Set Similarity Join

There are two categories of SSJ problems: approximate SSJ and exact SSJ. For approximate SSJ problems, it is acceptable to output pairs that are below the similarity threshold, and miss pairs that are above the threshold. We focus on the exact SSJ problem: the output pairs must be correct and there should be no missing correct pairs.

Given two collections (sets), S and R, formed over the same universe U of tokens (set elements), and a similarity function between two sets, $sim : \mathscr{P}(U) \times \mathscr{P}(U) \rightarrow [0, 1]$, the SSJ between S and R computes all pairs of sets $(s, r) \in S \times R$ whose similarity exceeds a user-defined threshold t, where $0 < t \leq 1$. That is, the output is the set of all pairs (s, r) with $sim(s, r) \geq t$. Following previous work [8,1,12] on SSJ algorithms, we hereafter exemplarily focus on all-pairs self-joins using the inverse Jaccard distance as a similarity function.

[4] Our implementation is available at `https://github.com/fabiyon/ssj-sisap`.

Algorithm 1: Sequential `AllPairs` algorithm.

Data: R, $invertedIndex$, t
Result: $\{(r,s)|(r,s) \in R \times R, r \neq s, sim(r,s) \geq t\}$
1 **foreach** $r \in R$ **do**
2 $candidates \leftarrow \{\}$
3 **foreach** $token \in$ GetPrefix(r,t) **do**
4 **foreach** $s \in$ GetList$(invertedIndex, token)$ **do**
5 $candidates \leftarrow candidates \cup \{s\}$
6 **foreach** $s \in candidates$ **do**
7 Verify(r,s,t)

However, all subsequently described approaches can be adapted to the $R \times S$ join and are applicable to other set similarity measures such as Cosine or Dice. All our datasets are a single collection R of sets consisting of sorted tokens. In the following, we use the terms *set* and *record* interchangeably.

2.2 State-of-the-Art Approaches

The exact SSJ computation can be expensive: to compute the SSJ over R, $|R| \cdot |R|$ set comparisons need to be performed in the worst case. To speed up SSJ, a line of prior work focused on minimizing the number of candidates generated. Efficient techniques for SSJ use filters to avoid comparing hopeless record pairs, i.e., pairs that provably cannot pass the threshold [1,4,12]. We distinguish two classes of filters. *Filter-verification* techniques use set prefixes or signatures followed by an explicit verification of candidate pairs (e.g., [1,12]). *Metric-based* approaches regard each record as a point in space with each token as dimension. It partitions the space such that similar records fall into the same or nearby partitions (e.g. [7]). The study in [6] suggests that this approach is not efficient. Thus, we focus on the filter-verification approach.

To generate candidate pairs, for each set r in the input collection R, the filter-verification approach aims to find other sets s in R which contain tokens (set elements) from r. Inverted indexes are used to speed up the process. For a given similarity threshold t and a set R, we only need to probe the inverted index for a subset of tokens in R (i.e., the *prefix*) to discover all possible candidates. For a Jaccard similarity threshold t, the size of the prefix can be computed as $|R| - \lceil |R| \cdot t \rceil + 1$ (referred to as *prefix filter* [4]). Any prefix-sized subset of tokens in R can be used as prefix. Thus, choosing the subset of the least frequent tokens would be the most efficient and likely yield the least number of candidates. As a result, most exact SSJ algorithms sort tokens in every set using inverse global token frequency, so that the prefix can be obtained by reading the inverted lists of the tokens starting from the beginning.

Several SSJ algorithms use prefix filtering, such as `AllPairs` [1], `PPJoin` [12] and `GroupJoin` [3]. Algorithm 1 shows the major steps of `AllPairs`. In lines

2–5, the candidates of set R are found from the inverted lists of the prefix of R; at lines 6–7 the exact similarity of each pair (r, s) is computed. `PPJoin` extends `AllPairs` by using a *position filter* which helps remove candidates based on the position of the first intersecting tokens in R and S [12]. `GroupJoin` further extends `PPJoin` by merging identical prefixes over multiple sets; this avoids the re-computation of the same overlaps [3]. `MPJoin` [10] introduces a removal filter. It disregards entries in the inverted index that do not pass future applications of the position filter. It uses the observation that records are indexed and probed in ascending order in length such that the required overlap increases monotonically.

Besides the CPU-based algorithms described previously, some recent work focuses on speeding up SSJ using different hardware platforms, notably GPUs. Quirino et al. proposed a standalone GPU algorithm that runs both candidate generation and verification within GPU using a block-based probing approach [9]. Bellas et al. proposed a different framework that uses GPU for candidate verification, while keeping candidate generation a CPU task [2]: working under a much-limited GPU memory, candidate pairs are verified by GPU in chunks. The experimental result in [2] indicates that the CPU-GPU solution out-performs the standalone GPU algorithm in [9]. It batches candidate pairs for verification when the number of candidates are large, which is the case for low similarity thresholds. As noted by the authors of [2], the bottleneck in SSJ is often the candidate generation rather than candidate verification, thus the acceleration provided by GPU is limited. In comparison, our work exclusively focuses on the parallelization potential of multi-core hardware in combination with existing filtering approaches and implementation optimizations – it is orthogonal to the recent work on GPU-based SSJ.

2.3 Modern Multicore Systems

We target single-node shared memory systems with multiple processors and a high core count. All processors in such a system are connected through an interconnect (e.g., Intel QPI) that implements a coherence protocol. Each processor can access it's local memory through an integrated memory controller. Local memory access is fast, while accessing remote memory attached to other processors on the same machine, comes with additional latency. This is referred to as "NUMA effect."

Modern processors use caches for performance. There are usually three levels of caches with a total size of tens of MBs. The last level cache (LLC) is usually shared among all cores in a processor. The first-level cache L1 is typically small and core-local. Modern processors usually provide hyperthreading. The idea is to better utilize a processor by allowing two processes to concurrently access different resources of one core, i. e. the arithmetic logic unit (ALU) or the floating point unit. Another important technique for performance is prefetching. Processors probabilistically read data from main memory which is likely to be used subsequently by a running program. Prefetching can hide memory stalling, i.e., a core waiting for data to arrive from main memory. Prefetching is done automatically or explicitly as instructed by software.

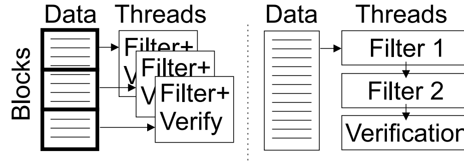


Fig. 1. Data-parallel (left) vs. pipelined (right) execution models.

3 Parallelizing Filter-Verification based Set Similarity Join

We choose the `AllPairs` algorithm [1] as the basis filter-verification algorithm. First, we discuss the execution model (i.e., how to assign threads tasks to run SSJ algorithms), and then discuss the design considerations in the context of multicores. We show the impact of different design decisions experimentally in Section 4.

3.1 Execution Model

An SSJ algorithm can be parallelized using *data parallelization* or *pipelining*. Figure 1 shows the basic idea of each design choice. In data parallelization, the input data is partitioned into disjoint batches consist of a tunable number of records. Each thread then runs the `AllPairs` algorithm (or `PPJoin` when the position filter is activated) on a different batch. Multiple threads can proceed in parallel without conflicts. In practical implementations, a pool of threads can be created upon system start. After a batch is processed, the thread continues with the next batch, avoiding the cost of creating and destroying threads at runtime.

Another possible parallelization model is *pipelined* execution. The SSJ task can be subdivided into sub-tasks, each of which can be executed on a dedicated thread. The entire join algorithm is finished cooperatively by multiple threads which communicate through message passing.

Compared to data parallelization, pipelining requires frequent inter-thread communication and synchronization using message queues. We experimentally verified that such overheads were too high to make the parallel algorithm efficient. Therefore, in the rest of the paper we focus on data parallelization.

3.2 Design Considerations

Under the data-paralell execution mode, we identify four important issues in designing parallel SSJ algorithms.

Filters. It was not clear how filter techniques used in single-core algorithms may behave on multicores. Prior study [8] has shown that the prefix filter in `AllPairs` is the most effective filter technique. Besides the prefix filter, `AllPairs` also includes the important length filter. Furthermore, we explore the use of `PPJoin`'s position filter. Our parallel SSJ algorithm ignores by default the entries

in the inverted index which cannot be similar due to length differences. This is comparable to the deletion filter of `MPJoin`, which deletes such entries in the inverted index which cannot pass the position filter for following probe records. Since it has a significant positive impact in all our experimental cases, we decided to use this optimization by default.

Record Inlining. Records consist of a record ID (integer) and a variable number of integer tokens. A straightforward record implementation is to use a `struct` which contains the id and a pointer to an array of integer tokens. In order to access the tokens, the pointer has to be dereferenced first which often incurs expensive cache misses and CPU stalls as the processor waits for data to be fetched from memory to CPU caches. By inlining, we co-locate the tokens with the record ID without such extra layer of indirection. We expect this to be more efficient as it avoids pointer-chasing during record access. `AllPairs` reads the records including their tokens one-by-one in the filter phase, so we expect a positive effect on runtime. On the other hand, inlining introduces overhead when accessing records randomly due to the variable-length tokens. For random reads, we introduce a pointer array that maps token IDs to the location of the corresponding record in the record array. `AllPairs` accesses records randomly in the probe phase. As a result, in the probe phase, both variants (with and without inlining) do pointer chasing once per record.

Thread Affinity. Threads running SSJ tasks may get migrated among cores by the OS scheduler because of various events if they are not pinned to specified hardware threads or cores. This can degrade performance due to the NUMA effect in case a thread is migrated to a socket but the data it is accessing is on another socket.

Batching. In the data-parallel execution model, each thread runs the SSJ algorithm by batches. The batch size controls the number of records that are joined on one thread without synchronizing with other threads. Thus, we expect the batch size to influence the runtime.

4 Experiments

In this section, we empirically quantify the impact of the design considerations discussed in the previous section.

4.1 Setup

Our implementation uses C++11 and allows tuning of various parameters as described in the previous section. We run experiments on a server with two Intel(R) Xeon(R) CPU E5-2620 processors clocked at 2.0GHz and 32 GB of DRAM. Each CPU has six cores (12 hyperthreads) and 32KB/256KB/15MB of L1/L2/L3 caches. We use the machine exclusively for the experiments. Since system processes and hardware events (network etc.) can influence the measurements, we repeat each experiment three times and report the average to even out such effects.

Datasets. We use 10 real-world and two synthetic datasets from a prior non-distributed experimental survey [8]. Table 1 summarizes the characteristics of these datasets. We omit more detailed descriptions of the datasets available elsewhere [8]. Similar to prior work [8], we assume that records in the input collection R are sorted by ascending lengths. This is important for applying length filter and reducing index accesses. Tokens within each record are sorted by global token frequency. Using the least frequent tokens for the prefix reduces the number of candidates. There are no exact duplicates in the datasets; finding exact duplicates is an important but orthogonal problem, and does not impact our design. In order to analyse the runtime behavior on larger datasets we scale each dataset 5 and 10 times using the method from [11]. We copy each record n times, and in the copied records, each contained token is replaced with the next token in the global token frequency. Token lengths and distributions remain unchanged. Note that the approach does not introduce duplicates. Similar to prior work on sequential SSJ algorithms, we assume the input datasets fit in main memory: modern multicore servers often feature 100s of GBs or even TBs of main memory. Even if the dataset does not fully fit in memory, we expect for a majority of workloads, the working set will fit in memory such that during SSJ execution no disk I/O is involved on the critical path.

Metrics. We vary the Jaccard similarity threshold among 0.6, 0.75, and 0.9. We assume these are sensible values for many SSJ applications. We measure the runtime within our program from including the index build until the join computation is completed. We do not store the join result itself, only its size. We run our code with all combinations of variables described in the previous section and report on the results in the following. In each run, we also profile the execution using `perf` to gather metrics such as cache misses. This adds a small and constant amount of overhead, however, it does not affect relative runtimes, which are important in our discussion.

Table 1. Characteristics of the experimental datasets.

Dataset	# recs $\cdot 10^5$	Record length		Universe $\cdot 10^3$		Size (B)
		max	avg	size	maxFreq	
AOL	100	245	3	3900	420	396M
BPOS	3.2	164	9	1.7	240	17M
DBLP	1.0	869	83	6.9	84	41M
ENRO	2.5	3162	135	1100	200	254M
FLIC	12	102	10	810	550	92M
KOSA	6.1	2497	12	41	410	46M
LIVE	31	300	36	7500	1000	873M
NETF	4.8	18000	210	18	230	576M
ORKU	27	40000	120	8700	320	2.5G
SPOT	4.4	12000	13	760	9.7	41M
UNI	1.0	25	10	0.21	18	4.5M
ZIPF	4.4	84	50	100	98	33M

4.2 Speedups and Scalability

We first investigate how the number of threads affects runtime.

Speedup over Single-Core Execution. Using multithreading is benefitial for the SSJ runtime under all combinations of our input datasets and thresholds. We observed speedups of roughly 2-10 times on our hardware. We omit the detailed results for brevity. The absolute runtimes of the multithreaded version vary between rougly 0.2 and 262 seconds for all datasets and thresholds. For each combination of input and threshold, we evaluated the parameter combination of number of threads, core affinity, position filter, inlining, and batch size leading to the lowest runtime. Overall, the best runtimes were achieved by using 24 or 32 threads. 70% of the best runtimes were achieved for a batch size of 125 or 250. The position filter is effective for most (70%) of the cases. However, we did not find an optimal parameter configuration for all the combinations of dataset and threshold. In the following, we discuss the influence and interdependencies of and between the variables and draw conclusions under which conditions which variable values are favorable.

Scalability. Figure 2 shows the speedup of our experiments relative to the number of threads. Without loss of generality we only consider results with the following parameters: no inlining, batch size 500, no CPU affinity, and no position filter. Other parameter combinations show a similar behavior, so we omit them here. For the majority of results, the speedup increases linearly up to 12 threads (number of physical cores). Starting with 16 threads, we record decreasing speedups. The optimal runtime is achieved at 24 threads for all datasets except ORKU and LIVE (0.75 and 0.9 similarity thresholds), and ENRO (0.9 similarity threshold). Since the machine has 12 physical cores, this result shows that SSJ algorithms generally benefit from hyperthreading which can hide memory access latency caused by cache misses. This is not a trivial result, as hyperthreading was shown to be only benefitial in a limited range of cases [5].

The results show that the scalability varies depending on the input dataset, the threshold, and the number of threads. Note that SPOT is an exception showing a hard limit at a speedup of 2, independent of the threshold and the number of threads. We found that the scability behavior is related to index lengths (the number of record IDs for each token). For SPOT, the average index length varies between 2.19 and 0.82 (for thresholds 0.6 and 0.9, respectively). The index lengths of other datasets and thresholds varies between roughly 1500 (for UNI and threshold 0.6) and 2.6 (AOL, threshold 0.9). Only ENRO, FLIC, LIVE and ORKU reveal comparably short index lengths for a threshold of 0.9. The low speedup of SPOT can be explained by data access patterns which can improve or prevent prefetching. Our implementation probes the inverted index for each prefix token in each record. If there are sufficiently many entries in the postings list, the CPU can guess that they are needed subsequently. If there is only a small number contained, prefetching does not apply and wait can occur. Longer postings lists in the inverted index give better scaleups as we show in the following section.

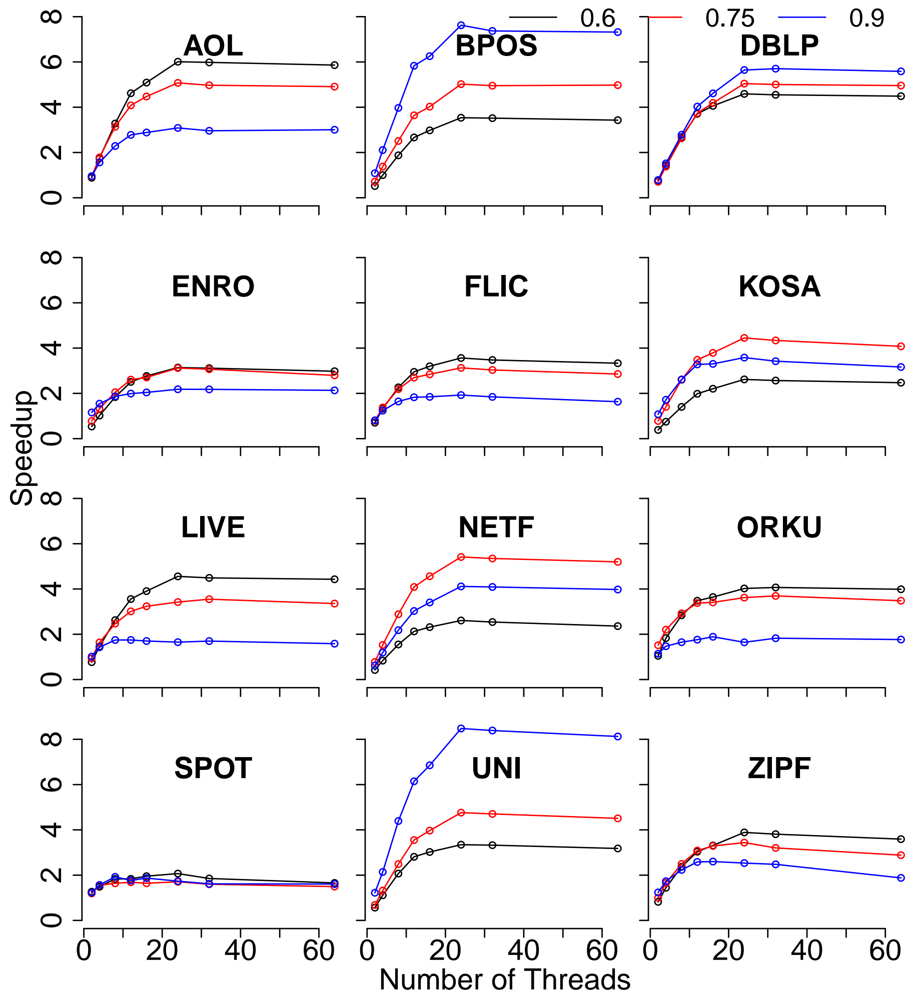


Fig. 2. Speedup under various datasets and similarity thresholds.

4.3 Impact of Data Size

We enlarge our datasets synthetically as described in Section 4.1. With 5× larger data, the runtime increases between 3.6× and 44×; the numbers for 10× larger data are 6.1× and 182×. In most cases, the runtime does not increase linearly with respect to data size. This is expected because SSJ is a quadratic operation. The filter-verification framework only optimizes the operation depending on favorable data characteristics.

Only ENRO, FLIC, LIVE, ORKU, SPOT, and ZIPF show roughly a linear runtime increase for a threshold of 0.9; for SPOT we observe linear runtime increases under thresholds 0.75 and 0.6. As we have shown in the previous section with the original datasets, SPOT was not well parallelizable. The relative runtime increase for 5/10× larger data is below 5/10× for all thresholds, hence the scalability is better for the enlarged datasets. With larger datasets, the postings list lengths in the inverted index also increase. We attribute the reason to be that this makes it easier for the CPU to prefetch the larger SPOT datasets.

4.4 Impact of Inlining

Inlining only has a positive impact on runtime for a minority of our experiments. It has a generally positive impact on experiments with the AOL dataset. Figures 3-4 show the runtime gain of AOL compared to the non-inlined version relative to the parameters method (single-threaded [allp], multithreaded [allph], multithreaded with CPU affinity [allps]) and threshold. There are no clear interdependencies to the other parameters number of threads, batching and position filter. The figures show that the biggest runtime gain occurs for a threshold of 0.6. Furthermore, only the multicore implementations profit from inlining. BPOS shows a similar behavior like AOL. We omit the figures for brevity. DBLP shows only small positive effects using inlining. The biggest runtime gain occurs for a threshold of 0.9. For KOSA, there is only a positive effect at 0.6. SPOT only shows a positive effect for 0.9. Inlining has a generally neutral or negative effect on the runtimes of ENRO, FLIC, LIVE, NETF, ORKU, UNI, and ZIPF.

We expected inlining to have a positive effect on the filter phase, because it saves pointer chasing to obtain the prefix tokens. It helps the CPUs to perform prefetching. However, if prefixes are much shorter than the complete records, many tokens must be skipped to read the next record. As shown in Table 1, AOL has the smallest average record size of 3. For such short lengths, the prefix is usually not much shorter than the record. Thus, prefetching may increase the runtime if there are many short records in the input dataset.

4.5 Impact of Batching

We grouped the experimental results by all variables except the batch size and computed the percentaged difference between the lowest and the highest runtime. It varies between 0.7% and 1%. Thus, we consider the impact of batching on the runtime as rather low. Our runtime experiments suggest that the batch size of

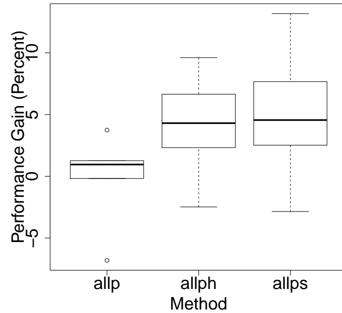


Fig. 3. AOL: Runtime gain of inlining relative to single-threaded (allp), multi-threaded without (allph), and with CPU affinity (allps).

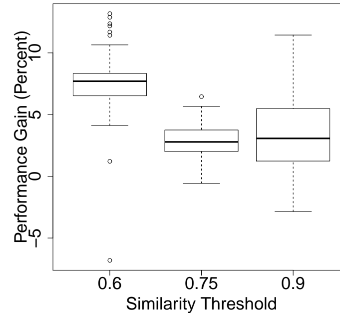


Fig. 4. AOL: Runtime gain of inlining relative to thresholds.

125 is the best in most cases (23 times) and 250 is the second best (10 times). We could not find a pattern that tells us when which batch size is optimal. It seems to be a complex relation with other variables and with the data characteristics.

4.6 Position Filter

Using position filter is benefitial in most cases with thresholds of 0.6 and 0.75. Figure 5 shows the relative runtime gains using the position filter grouped by threshold. For a threshold of 0.6 the runtime gain varies between 20-50% except for SPOT, where the median is close above zero. The position filter only has a small impact on SPOT for all thresholds. This can be explained by the number of candidates. For SPOT 0.6 the position filter saves roughly 8% of candidates, or in absolute numbers 50 000. The verification of this number of additional candidates is cheaper than to filter them out before. On other datasets this filter saves 28% of candidates on average. Only for AOL, the savings with the position filter are equally low with 7%. However, the absolute number of saved candidates is orders of magnitude higher with 86 600 000, so the position filter pays off for AOL. The `maxFreq` of tokens (cf. Table 1) gives a hint on the effectiveness of the prefix filter. The most frequent token in SPOT occurs roughly 9 700 times, which is very low compared to all other datasets. This implies that the prefix filter generates few candidates. The position filter only pays off if the prefix filter is less effective, which is the case for all other datasets and thresholds below 0.9. For a threshold of 0.75, the gain varies between 5-50% for all datasets except for SPOT (as discussed before) and AOL. For AOL the number of saved candidates relative to the number of candidates without position filter is 0.3% and thus comparably low. For a threshold of 0.9, all gains are close to zero except for DBLP and NETF where there is still a gain of 40% to 50%. One explanation is also the number of saved candidates.

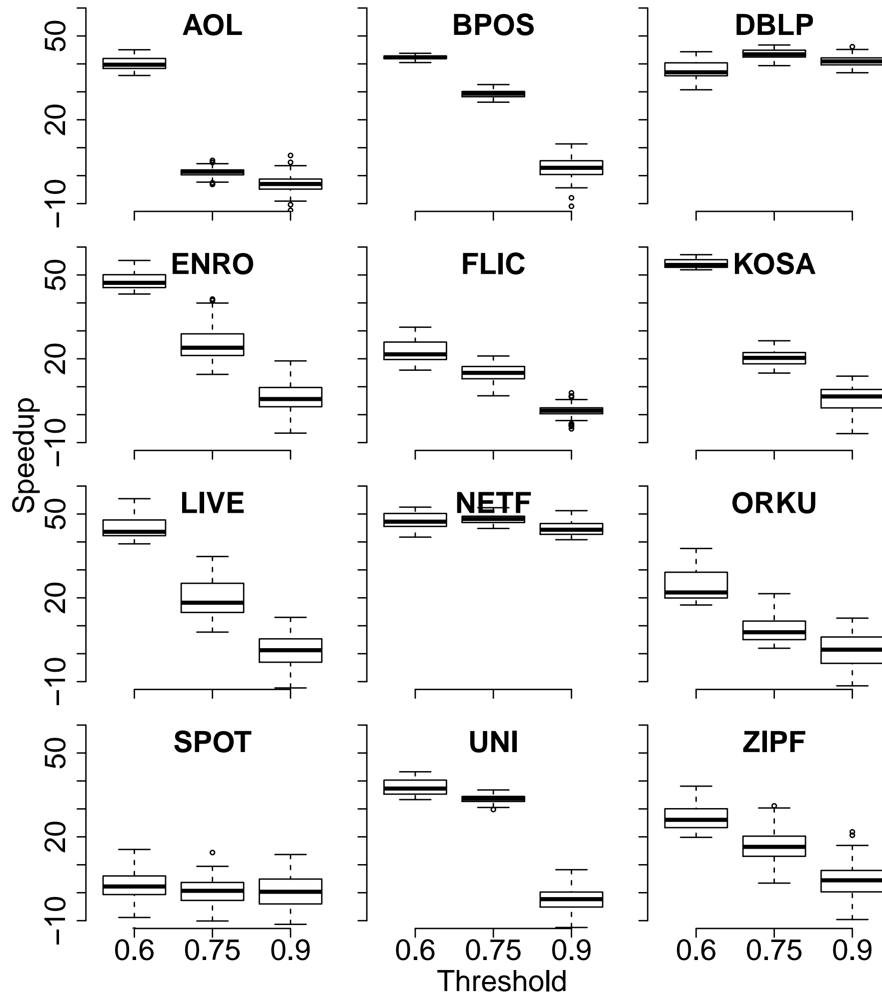


Fig. 5. Runtime gain/loss of using the position filter grouped by similarity threshold.

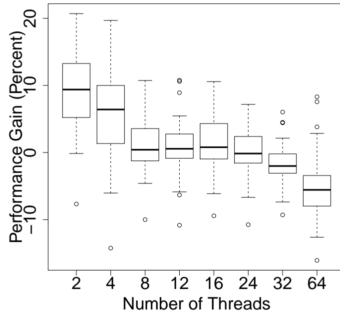


Fig. 6. Runtime gain/loss of using CPU affinity on the ENRO dataset.

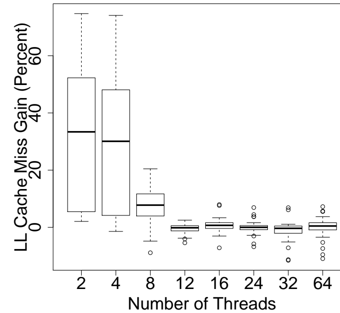


Fig. 7. Reduction of LLC misses using CPU affinity on the ENRO dataset.

We compared the effect of the position filter between the single-threaded SSJ vs. the multithreaded (using average runtimes). For brevity, we omit the detailed results. The effect is the same for the majority of cases. However, for AOL 0.75, FLIC 0.9, KOSA 0.9, ORKU 0.9, and SPOT 0.6 and 0.75 the position filter has a positive effect in the multicore case, while it does not have a positive effect in the single-core case. This observation suggests that the overhead of the position filter pays off more often in the multicore case. There is no obvious relationship between the runtimes using the position filter and the remaining parameters inlining, batching, the number of threads, and CPU affinity.

4.7 Impact of Thread Placement

By statically assigning the CPU affinity we expected a more optimal use of the cores and prevent thread migrations. However, our experiments reveal that statically assigning the CPU affinity is only benefitial for the runtime in a minority of cases. Figure 6 shows the performance gain using CPU affinity for ENRO exemplarily. There is a performance gain for 2 to 4 threads. This gain decreases down to 12 threads, stays nearly the same up to 24 threads, and decreases for more threads. This can be explained by the saved cache misses. Figure 7 shows the percentage of saved LLC misses with CPU affinity. The runtime is generally the best from a number of threads starting from 24. Our results show that manually setting CPU affinity is not very helpful for optimizing SSJ algorithms.

5 Conclusion

Filter-verification based SSJ algorithms were either single-threaded or distributed, wasting much computing capability provided by multicore processors. In this paper, we fill the gap to explore the potential of parallelizing SSJ on multicores. We propose a data-parallelization execution model along with various design considerations, including the use of filters, CPU affinity, record inlining and batching.

Experiments using real-world datasets revealed several important insights. Using multithreading improves SSJ runtime by 2–10× on a 12-core machine; the optimal number of threads is often the number of hardware threads (hyperthreads). Surprisingly, unlike in many other workloads, using hand-crafting data structures (e.g., using inlining) or CPU affinity do not always lead to significantly higher performance. We also find that the position filter is more effective than in the single-core scenario and should generally be used for parallel SSJ. One interesting direction of future work is to use a multithreaded CPU and GPU parallelization for the computation of the SSJ and find the optimal point (i. e., number of candidates) from where the usage of the GPU is benefitial.

Acknowledgements This work was partially supported by a LexisNexis research grant. We thank Panagiotis Bouros for the sequential SSJ source code.

References

1. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th international conference on World Wide Web. pp. 131–140. ACM (2007)
2. Bellas, C., Gounaris, A.: Exact set similarity joins for large datasets in the GPGPU paradigm. In: Neumann, T., Salem, K. (eds.) Proceedings of the 15th International Workshop on Data Management on New Hardware. pp. 5:1–5:10. ACM (2019)
3. Bouros, P., Ge, S., Mamoulis, N.: Spatio-textual similarity joins. Proceedings of the VLDB Endowment **6**(1), 1–12 (2012)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: Proceedings of the 22nd International Conference on Data Engineering (ICDE). pp. 5–5. IEEE (2006)
5. Drepper, U.: What every programmer should know about memory. Red Hat, Inc **11**, 2007 (2007)
6. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.C.: Set similarity joins on MapReduce: an experimental survey. PVLDB **11**(10), 1110–1122 (2018)
7. Jacox, E.H., Samet, H.: Metric space similarity joins. ACM Transactions on Database Systems (TODS) **33**(2), 7 (2008)
8. Mann, W., Augsten, N., Bouros, P.: An empirical evaluation of set similarity join techniques. PVLDB **9**(9), 636–647 (2016)
9. Quirino, R.D., Ribeiro-Júnior, S., Ribeiro, L.A., Martins, W.S.: Efficient filter-based algorithms for exact set similarity join on gpus. In: Hammoudi, S., Smialek, M., Camp, O., Filipe, J. (eds.) Enterprise Information Systems - 19th International Conference, ICEIS. Lecture Notes in Business Information Processing, vol. 321, pp. 74–95. Springer (2017)
10. Ribeiro, L.A., Härder, T.: Generalizing prefix filtering to improve set similarity joins. Information Systems **36**(1), 62–78 (2011)
11. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 495–506. ACM (2010)
12. Xiao, C., Wang, W., Lin, X., Yu, J.X., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Transactions on Database Systems (TODS) **36**(3), 15 (2011)