Bwjoin: A Blockwise GPU-based Algorithm for Set Similarity Joins

Rafael D. Quirino[1], André M. Quirino[2], Leonardo A. Ribeiro[1], Wellington S. Martins[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

rafaelquirino@discente.ufg.br, {wellington,laribeiro}@inf.ufg.br

[2]Visiona Tecnologia Espacial
São José dos Campos – SP – Brazil

andre.moreira.quirino@gmail.com

Abstract. *Set similarity joins play a pivotal role in diverse fields, ranging from modern database management systems to near-duplicate detection and even galaxy cluster analysis in cosmology. However, due to their quadratic nature, these operations have been associated with substantial computational costs. To tackle this challenge, parallel solutions have been developed in recent years, spanning algorithms for distributed and shared memory architectures, as well as massively parallel systems like GPU accelerators. In this paper, we propose a new GPU-based algorithm, using the prefix-filtering technique, that harnesses the power of blockwise parallelism, achieving better performance than its competitors, especially for high threshold similarity joins in big datasets.*

1. Introduction

Set similarity join retrieves all pairs of similar sets from a dataset, where two sets are considered similar if the value returned by a set similarity function for them exceeds a given threshold. It serves as a fundamental operation and a crucial step in various advanced data processing tasks like integration, cleaning, and data mining. Moreover, its significance extends to diverse fields, encompassing modern database management systems, near-duplicate detection, and even galaxy cluster analysis in cosmology. This broad applicability has fueled a growing interest in efficiently processing set similarity join queries [Sarawagi and Kirpal 2004, Chaudhuri et al. 2006, Bayardo et al. 2007, Vernica et al. 2010, Xiao et al. 2011, Ribeiro and Härder 2011, Cruz et al. 2015].

Evaluating the exact similarity between complex objects can be computationally expensive, even for state-of-the-art algorithms. By adopting this set-based approach, similarity functions can be utilized to compare data pairs efficiently. Consequently, predicates involving these functions can be expressed as set overlap constraints, reducing the set similarity join task to the identification of set pairs with significant overlap.

The quadratic nature of the set similarity join, poses challenges in obtaining an efficient solution. To address this, parallel approaches have been developed in recent years, encompassing algorithms for distributed and shared memory architectures, as well as massively parallel systems like GPU accelerators [Cruz et al. 2015, Cruz et al. 2016, Ribeiro-Júnior et al. 2017, Quirino et al. 2017,

Bellas and Gounaris 2020, Fier and Freytag 2022]. The objective is to leverage parallelism to accelerate the filtering of pairs with low similarity, leaving only the similar pairs (candidates) for subsequent verification. The key challenge lies in achieving fast processing and optimal memory usage to ensure the scalability of the solution.

In this paper, we propose a new GPU-based algorithm, using the prefix-filtering technique, that harnesses the power of blockwise parallelism, achieving better performance than its competitors, especially for high threshold similarity joins in big datasets. The key idea behind our proposal involves assigning each GPU multiprocessor to process a query set and loading its associated inverted list into the shared memory. This optimization aims to enhance memory access patterns and reuse while eliminating the necessity for partial intersection matrix compression and memory reconfiguration during each filtering-verification cycle. The contributions of this paper are a blockwise parallel algorithm for the set similarity join problem, a GPU-based implementation and extensive experimental work with standard datasets with speedup gains over state-of-art competitors.

2. Related Work

The efficient evaluation of Set Similarity Join queries has been an active research topic over the years. Most solutions are designed for CPU and assume memory-resident data [Sarawagi and Kirpal 2004, Chaudhuri et al. 2006, Bayardo et al. 2007, Xiao et al. 2011, Ribeiro and Härder 2011, Li et al. 2021]; some works also proposed extensions for disk-resident data [Bayardo et al. 2007, Ribeiro and Härder 2011]. [Mann et al. 2016] presented an empirical evaluation of several CPU-based algorithms. More recently, [Wang et al. 2017] proposed a new inverted list organization to improve the filtering step and exploited the fact that similar sets produce similar results to improve the verification step. [Li et al. 2021] introduced a bitmap-like indexing structure and proposed a learning-based partitioning approach to increase pruning power. [Fier and Freytag 2022] adapted existing SSJ algorithms to exploit multi-threading parallelism on multicore CPUs.

[Ribeiro-Júnior et al. 2016] introduced `gSSJoin`, the first exact SSJ algorithm designed for GPUs. Later on, the same authors presented `sf-gSSJoin`, which we used in this article, and `fgssjoin`, which apply various filters to reduce the number of candidates. In all three algorithms, SSJ is entirely executed on the GPU. [Bellas and Gounaris 2019] presented an SSJ algorithm based on CPU-GPU coprocessing: in a multi-threading scheme, filtering is performed on GPU while the whole verification is delegated to the GPU. The same authors later presented a comprehensive evaluation of GPU-accelerated SSJ algorithms, which identified data characteristics and threshold values influencing their performance. Finally, the authors proposed HySet [Bellas and Gounaris 2021], a framework to execute SSJ concurrently on CPU and GPU in a single-machine setting. The framework explores two strategies for distributing the workload between these processors, with the most effective one allocating fixed fractions of the workload to the CPU and GPU.

3. Background

Definition 1. (Set Similarity Join). *Let U be a universe of elements, C be a set collection where every set consists of a number of elements from U, $Sim(x, y)$ be a similarity function that maps two sets from C to a number in $[0, 1]$ and γ be a similarity threshold*

in $[0, 1]$. *Set similarity join is the operation of defining the set* S *of all pairs of sets from* C*, for which* $Sim(x, y) \geq \gamma$.

We focus on a general class of set similarity functions, for which the similarity predicate can be equivalently represented as a set overlap constraint. Specifically, we express the original similarity predicate in terms of an overlap lower bound [Chaudhuri et al. 2006].

Function	Definition	minoverlap(x,y)	[minsize(x),maxsize(x)]
Jaccard	$\frac{\|x \cap y\|}{\|x \cup y\|}$	$\frac{\gamma}{1+\gamma}(\|x\| + \|y\|)$	$\left[\gamma \|x\|, \frac{\|x\|}{\gamma}\right]$
Dice	$\frac{2\|x \cap y\|}{\|x\| + \|y\|}$	$\frac{\gamma(\|x\| + \|y\|)}{2}$	$\left[\frac{\gamma \|x\|}{2-\gamma}, \frac{(2-\gamma)\|x\|}{\gamma}\right]$
Cosine	$\frac{\|x \cap y\|}{\sqrt{\|x\| \|y\|}}$	$\gamma\sqrt{\|x\| \|y\|}$	$\left[\gamma^2 \|x\|, \frac{\|x\|}{\gamma^2}\right]$

Table 1. Set Similarity Functions

Definition 2 (Overlap Bound): *Let* x *and* y *be sets,* Sim *be a set similarity function, and* γ *be a similarity threshold. The overlap bound between* x *and* y *relative to* Sim*, denoted by* $minoverlap(x, y)$*, is a function that maps* γ *and the sizes of* x *and* y *to a real value, s.t.* $Sim(x, y) \geq \gamma \Leftrightarrow |x \cap y| \geq minoverlap(x, y)$.

Framing the problem in this way enables us to reduce the set similarity join to a set overlap constraint, in which we need to obtain all pairs (x, y) whose overlap is not less than $minoverlap(x, y)$. The set overlap formulation enables the derivation of size bounds. Intuitively, observe that $|x \cap y| \leq |x|$ whenever $|y| \geq |x|$, i.e., set overlap and similarity are trivially bounded by $|x|$. Exploiting the similarity function definition, it is possible to derive tighter bounds allowing immediate pruning of candidate pairs whose sizes are incompatible according to the given threshold.

Definition 3 (Size Bounds): *Let* x *be a set of features,* Sim *be a set similarity function, and* γ *be a similarity threshold. The size bounds of* x *relative to* Sim *are functions, denoted by* $minsize(x)$ *and* $maxsize(x)$*, that maps* γ *and the size of* x *to a real value, s.t.* $\forall y$*, if* $Sim(x, y) \geq \gamma$ *then* $minsize(x) \leq |y| \leq maxsize(x)$.

Therefore, given a set x we can safely ignore all sets whose sizes do not fall within the interval $[minsize(x), maxsize(x)]$, for they cannot be similar to x according to the given threshold. Table 1 shows the overlap and size bounds of three of the most widely used similarity functions [Ribeiro and Härder 2011, Xiao et al. 2011]: Jaccard, Dice, and Cosine.

If we ensure that all the sets in the collection have its elements under the same total order O, we can combine overlap and size bounds to prune even more the comparison space through the *prefix filtering* technique [Chaudhuri et al. 2006,

Sarawagi and Kirpal 2004]. The idea is to derive a new overlap constraint to be applied in only subsets of the original sets. For any two sets x and y, under the order O, if $|x \cap y| \geq \alpha$ then the subsets consisting of the first $|x| - \alpha + 1$ elements of x and the first $|y| - \alpha + 1$ elements of y must share at least one element [Chaudhuri et al. 2006, Sarawagi and Kirpal 2004]. A careful observation reveals that if $\alpha = \lceil minoverlap(x, y) \rceil$ then the set of all pairs (x, y) sharing a common prefix element must contain the result set S. The exact prefix size is determined by $minoverlap(x, y)$, but it depends on each matching pair. Given a set x, the question is how to determine $|pref(x)|$ such that it suffices to identify all matches of x. Clearly, we have to take the largest prefix in relation to all y. The prefix formulation given above tell us that the prefix size is inversely proportional to $minoverlap(x, y)$, and the former increases monotonically with y. Therefore, $|pref(x)|$ is largest when $|y|$ is smallest. The smallest possible size of y, such that the overlap constraint can be satisfied, is $minsize(x)$.

4. Bwjoin

4.1. The Algorithm's Framework

Prefix-filtering algorithms of the *allpairs* family [Bayardo et al. 2007] are organized in an index-filter-verify cycle. The same is true for its GPU versions. The main difference lies in the *a priori* computation of the inverted index since we need the full index to run the process independently for all processing units (whether they are single threads or thread blocks). The `fgssjoin` and `sf-gssjoin` algorithms [Quirino et al. 2017, Ribeiro-Júnior et al. 2017], include a splitting of the dataset in partitions in such a way as to enable both the processing of datasets bigger than the size of the device memory and the leveraging of the dataset order (sorted reversely by set size) to skip partition pairs using the size-filtering technique (cases in which the last set in the first partition is bigger than the *maxsize* of the first set in the second partition).

Assuming we have a dataset of size N, the partitioning works as follows. We divide the dataset into p partitions of size N/p, which must fit into the GPU's global memory. Then, we iterate over the partitions. For each partition p_i we build its inverted index and, again, iterate, this time over the partitions preceding p_i, including itself. For each partition p_j in the second iteration, we probe p_j against p_i index, i.e. we search p_i index with the prefix elements from p_j sets. This process, which is the filter phase in the cycle, yields the candidate pairs that are later verified by calculating each pair's full intersection. Those pairs whose intersections are not less than the pair's *minoverlap* are finally added to the result. The top half of Figure 1 illustrates the partitioning scheme for three partitions.

`Bwjoin` uses a similar framework, i.e. the same block partitioning scheme and indexing/verifying algorithms (as described in [Quirino et al. 2017]), but innovates by proposing a new filtering algorithm, which demands some modifications in the framework. In `fgssjoin`, a quadratic matrix is created for keeping all possible candidate pairs from the index/probe partitions (even the ones that will not be generated by the filtering algorithm). `Bwjoin` uses a fixed-sized buffer, which can overflow. If it happens, we need to save the state of the filtering algorithm, output the buffer's result, and call the algorithm again to resume processing the probing sets that were not finished.

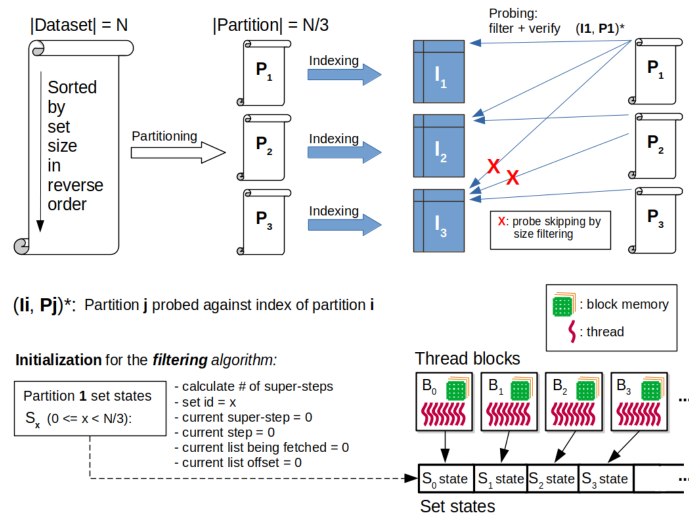


Figure 1. Block partitioning and filtering initialization.

4.2. Blockwise Parallelism and the New Filtering Algorithm

One way to parallelize the filtering phase of prefix-filtering algorithms for set similarity joins is to assign one thread to each probing set, as in `fgssjoin`. This results in each thread being responsible for fetching its set's prefixes and each of its inverted index lists, performing the searches, applying the filters, and computing the partial intersections. This approach is simple but has a key limitation, especially for GPUs: it produces random accesses to the global memory space. Another problem with `fgssjoin` is the quadratic nature of the partial scores matrix (as described in [Bellas and Gounaris 2020]).

Our new algorithm addresses both problems while introducing a new blockwise parallelization strategy. We use a thread block to filter a probing set's candidates instead of a single thread. Each block has an amount of local fast memory (shared memory and registers in CUDA). The idea is to allocate "tiles" of this local memory for each thread block, which are to be filled with the data from the set's prefixes and its inverted index lists. We also use a fixed-sized output buffer for the candidate pairs, which depends on the GPU memory size. The threads in the block fetch the data in batches through coalesced memory accesses and store it in the tiles. Then, they collaboratively perform both the filtering and the calculations to determine where to write the candidate pairs in the output buffer, i.e. the shuffling phase. It can take more than one step to fetch all the set lists from the index into the block tiles [1], and the output buffer can get full, in which case we would need to stop the algorithm and save its set fetching state for a further execution.

The bottom half of Figure 1 shows the initialization of probing set states for the new filtering algorithm. Figure 2 illustrates the memory configuration for thread block 0 processing probing set 0. We define blocks with T threads (four in the example in Figure 2), each one capable of processing I data values (items per thread, also four). Each block will process a probing set S (S_0 in the example).

[1] In the third super-step, in Figure 2 (prefix elements $[e_8, e_9, e_{10}, e_{11}]$), there are more than 16 items in the lists, which is more than the tile capacity.

4.3. Coalesced Memory Access Steps

Achieving coalesced memory access for irregular data (with variable data point sizes) is tricky and has been a common problem in GPU algorithm design. In our case, the irregularity is present in both the set's prefixes and the inverted index lists, which have variable sizes. To tackle this challenge, we designed a hierarchy of steps, each one a coalesced memory access performed by the threads as a block. We chose to name the levels in the hierarchy as "super-steps", "steps", and "micro-steps".

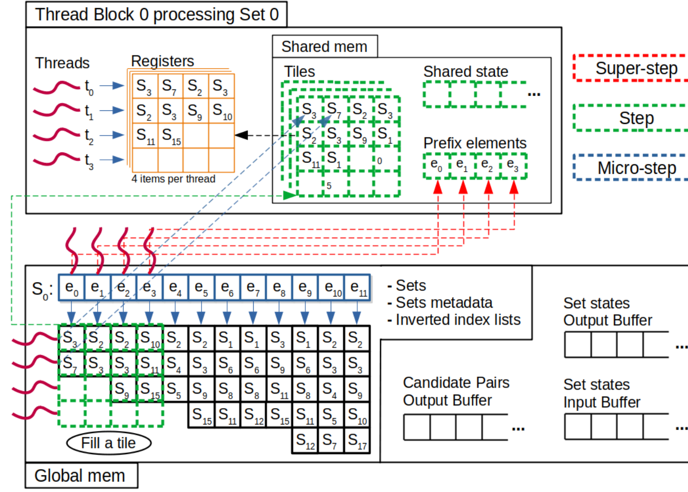


Figure 2. Illustration of the global memory access steps.

Figure 2 illustrates these steps. First, in dashed red line arrows, we have the super-step that loads the first four elements from S_0's prefix. Then, in a dashed green line arrow, we represent the data transfers that will take place in one step. To achieve one step, i.e. the filling of a whole tile with the inverted index list's elements associated with the prefix elements of the current super-step, I micro-steps will be needed, which are the actual coalesced memory accesses performed by the threads, of which the first one is represented in dashed blue line arrows (only the first two threads are working in the example). After the threads finished fetching a full tile, or all lists (pointed by blue arrows in the figure), the global memory accesses are over for this step. Still, each thread will get a slice of the tile into its local registers in order to execute the blockwise algorithms necessary for the filtering and shuffling phases.

In the example in Figure 2, one step was enough to conclude the first super-step, but if the inverted index lists were bigger, we would need further steps. Nevertheless, two more super-steps will be needed. In each change of super-step/step/micro-step, all threads in the block need to know the state of the data fetching process, such as what is the current super-step, the current step, the current inverted index list being fetched, the current offset of this list, etc. That is when the shared state variables come into play. At the beginning of each step in the hierarchy, a controller thread (t_0) will update these shared state variables, and all threads must synchronize to get to the same point in the steps hierarchy.

4.4. Parallel Filtering and Outputting of Candidate Pairs

At the end of a step we have a tile filled with set IDs fetched from the inverted index lists, which are sets that share at least one prefix element with the probing set, and it is time to

apply the filtering. The filtering is done by running a few blockwise parallel algorithms over the values from the tile. We need to keep in mind that when a partition is indexed and probed against itself, we may have the same candidate pairs considered twice. Moreover, there may be duplicates in the tile, i.e. a set may share more than one prefix element with the probing set. The first problem can be dealt with by ensuring that a probing set S_i will only consider set IDs greater than its own, i.e. S_j with $j > i$. The second problem can be solved by sorting the values (grouping duplicates) and a simple parallel algorithm in which each thread looks at the preceding value in its registers (or the last value from the preceding thread). If it is equal to the current value, since the values are sorted, the current value is filtered out. First, the tile is sorted using a blockwise parallel radix sort algorithm. Then, we do the filtering, in parallel, each thread processing its items in its registers. To improve performance, we express the two constraints mentioned earlier and the size filter as predicates, and apply them in a branchless fashion, generating a binary mask, with ones corresponding to the sets that passed the filter, making a candidate pair with the probing set.

Figure 3 illustrates the filtering and shuffling phases. The mask tile is important for the coalesced outputting of the candidate pairs. Its sum gives us the number of candidate pairs obtained in this step, and each of its positions prefix sums gives the thread the position where to write in the shuffle tile. To this end, threads in the block perform a parallel exclusive prefix sum over the mask. Then, each thread outputs its result to a shuffling tile, using its corresponding prefix sum as index. Now, thread t_0 will atomically update a global offset to the output buffer, effectively allocating memory space. Finally, the threads output the shuffling tile to the output buffer in a coalesced fashion. In case of output buffer overflow, the controller thread must update and save the shared state variables to a set state output buffer in the global memory, to resume the process in a further filtering kernel call, after the candidate pairs of the current get verified and sent to the final output.

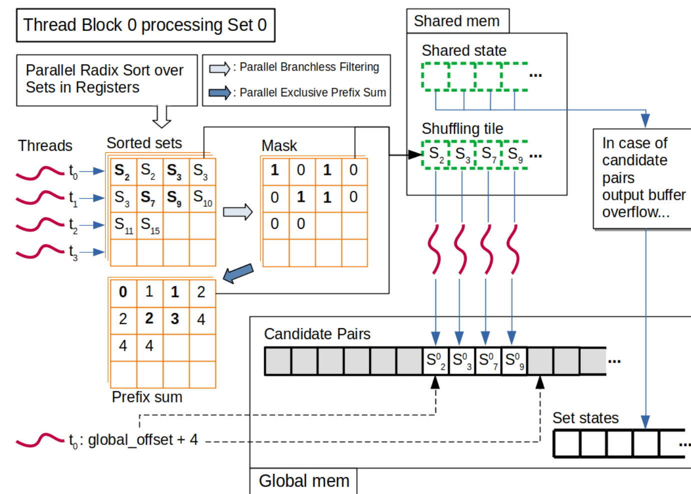


Figure 3. Illustration of the filtering and outputting phases of one step.

Algorithm 1 shows a simplified pseudo-code for `Bwjoin`'s filtering phase. Thread synchronization through barriers is indicated with “(Synchronize)”. The number

Algorithm 1 Bwjoin Filtering Algorithm (set state position in input buffer)

```
1: Thread 0 fetch the block's set state (Synchronize)
2: for all super-steps do
3:     Fetch a set tile from the maxprefix, in parallel (Synchronize)
4:     Thread 0 fetch the lists metadata for the current super-step (Synchronize)
5:     for all steps do
6:         for all micro-steps do
7:             Fetch inverted index list entries to the tile, in parallel (Synchronize)
8:             Thread 0 updates the fetching state (Synchronize)
9:         end for
10:        Transfer tile data to registers, in parallel (Synchronize)
11:        Perform a parallel radix sort (Synchronize)
12:        Perform a parallel filtering, generating a binary mask (Synchronize)
13:        Perform a parallel prefix sum over the binary mask (Synchronize)
14:        Perform a parallel shuffling of selected candidate pairs (Synchronize)
15:        Thread 0 increments the output buffer global offset (Synchronize)
16:        for all threads, in parallel do
17:            if not overflowing the output buffer then
18:                Write candidate pairs to the output buffer (Synchronize)
19:            else
20:                Thread 0 saves the fetching state in the global memory (Synchronize)
21:                Exit current kernel launch
22:            end if
23:        end for
24:    end for
25: end for
```

of super-steps, steps, and micro-steps, as well as other shared state variables, are obtained and managed by a controlling thread, t_0.

5. Experiments

5.1. Datasets

We used six standard datasets in our experiments [Bellas and Gounaris 2020]:

AOL: query records from the AOL search engine. Each set represents a record and contains an anonymous user id, the search string, the query time, and may contain an integer rank and a clicked link. We tokenized the records as strings, using 3-grams.

BMS: e-commerce purchased data. Each set represents an item, and its tokens are product categories.

DBLP: paper titles from the DBLP library of computer science academic work. Each set represents an article title, and its tokens are 3-grams.

ENRON: email data. Each set represents an email, and its tokens are words from the subject and body.

KOSARAK: click data from a news page. Each set represents user clicking behavior, and its tokens are clicked links.

ORKUT: social media data from ORKUT. Each set represents a user's interests, and its tokens are the user's groups.

Dataset	Cardinality	Avg set size	Distinct tokens
AOL	10.8×10^6	66.5402	138497
BMS	3.3×10^6	6.53033	3367020
DBLP	5.2×10^6	74.7267	227395
ENRON	2.4×10^5	135.198	1113220
KOSARAK	6×10^5	11.9336	41270
ORKUT	2.5×10^6	57.3032	7348298

Table 2. Dataset Information

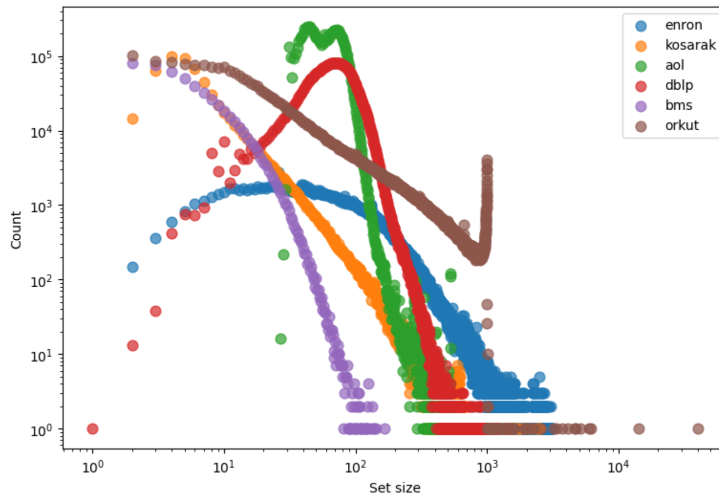


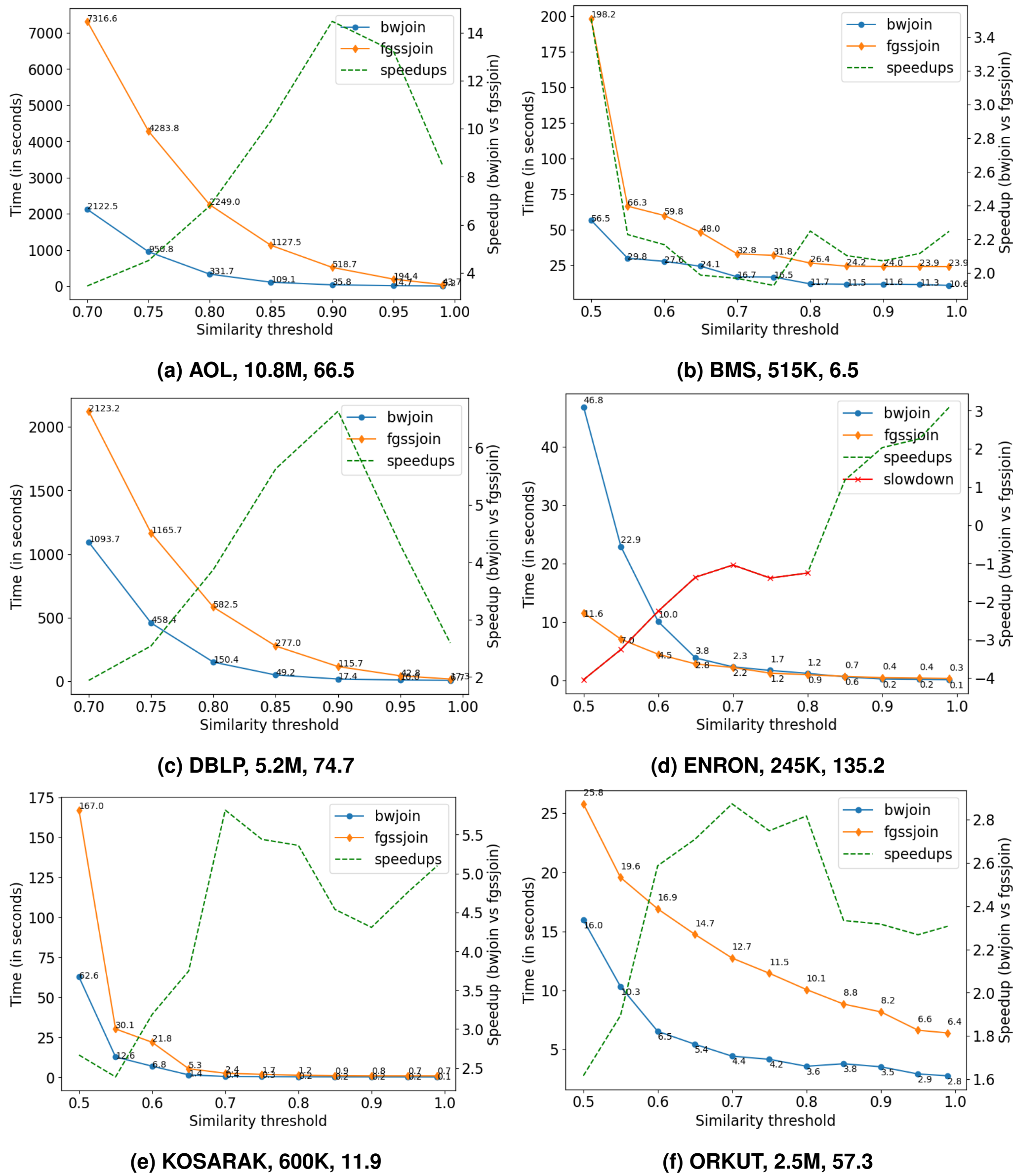
Figure 4. Set size distribution for all datasets.

5.2. Experimental Setup and Evaluation

Our experiments were executed on a system equipped with two Intel(R) Xeon(R) CPU E5-2620, 32GB of RAM, and two NVIDIA GeForce RTX 3060 GPUs with 12GB of memory and 28 SMs with 64 CUDA cores each. All results are averages of 5 independent executions, considering both processing and memory transferring times. Moreover, significance tests were run to test for differences with 95% confidence.

To execute `Bwjoin`, three parameters must be specified: the partition size, the number of threads per block, and the number of items per thread. The partition size, like in the original `fgssjoin` algorithm, relies on available device memory. However, there is a distinction in that, for `fgssjoin`, it is constrained by the quadratic nature of the partial scores matrix. In other words, the square of the partition size, multiplied by the size, in bytes, of one of its entries must fit within the GPU memory. For `Bwjoin`, we do not have this particular constraint since its buffer does not exhibit a quadratic relationship with the partition size. Nevertheless, we still need to consider the efficiency of the size filter. Larger partitions result in fewer size filter skips. Empirically, we opted for using partitions of 50,000 sets for `Bwjoin`, although similar results were obtained with partitions of 100,000 sets. For `fgssjoin`, we opted for partitions of 30,000 sets, as the square of

the partition size multiplied by the size of each of its entries comes close to the GPU memory size. The selection of the number of threads per block and items per thread in the `Bwjoin` algorithm is bounded by the shared memory available for GPU block execution. This decision was also based on empirical testing, resulting in 32 threads per block and 4 items per thread (8 also proved effective). It should be noted that the extensive use of shared memory can be considered a limitation of the algorithm.



(a) AOL, 10.8M, 66.5

(b) BMS, 515K, 6.5

(c) DBLP, 5.2M, 74.7

(d) ENRON, 245K, 135.2

(e) KOSARAK, 600K, 11.9

(f) ORKUT, 2.5M, 57.3

Figure 5. Execution times and speedups for Bwjoin vs fgssjoin, for all datasets.

Figure 5 shows the execution times, in seconds, and the speedups for `Bwjoin` vs `fgssjoin`, for all the datasets, with varying similarity thresholds. The datasets are indicated by their names, sizes, and average set size. We notice that the bigger speedups come from the bigger datasets, suggesting that `Bwjoin` scales better, and also for high thresholds. This is because those are the cases where the prefixes and the lists are the smallest, which means that the probability of the block needing fewer steps to process the

whole set prefix is the highest.

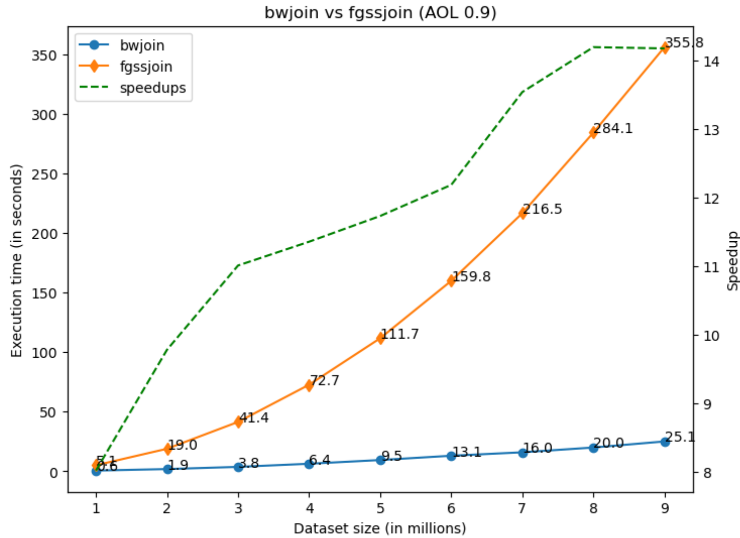


Figure 6. Execution times and speedups for partitions with growing cardinality, from the AOL dataset, with a threshold of 0.9.

To emphasize the difference in scalability between algorithms, we experimented with partitions of the AOL dataset with growing cardinalities, fixing a high threshold of 0.9, running both `Bwjoin` and `fgssjoin`. Figure 6 shows the execution times and speedups for these partitions, ranging from 1 to 9 million sets. We notice the higher speedups for bigger partitions, clearly indicating a better scaling of `Bwjoin` in relation to `fgssjoin`. Space (in the paper) and time constraints prevented us from experimenting with other algorithms. We refer to [Bellas and Gounaris 2020], which has extensive experimentation work with other GPU set similarity algorithms, including `fgssjoin`, and can be used as a reference together with this work.

6. Conclusions and Future Work

The experiments showed that `Bwjoin` performs better for big datasets than `fgssjoin`, and the best speedups were observed for higher thresholds, reaching a maximum at 0.9, for the biggest datasets (aol and dblp). The new filtering algorithm achieves better memory access and does not use a quadratic matrix for the resulting candidate pairs, using a fixed-sized buffer, dependent on the device memory size. The trade-off is a more complex algorithm including additional operations, like sorting, prefix sum, and synchronizations, and the possibility of having to run the algorithm multiple times, saving and restoring the data fetching state in between executions. We could try to use the same blockwise scheme for the verifying algorithm. Further experiments with even bigger datasets would shed more light on the algorithm's behavior. A multi-GPU version would demonstrate the power of the algorithms in multi-devices environments.

References

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *Proc. of the 16th Intl. Conf. on World Wide Web*, pages 131–140.

Bellas, C. and Gounaris, A. (2019). Exact Set Similarity Joins for Large Datasets in the GPGPU Paradigm. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 5:1–5:10.

Bellas, C. and Gounaris, A. (2020). An empirical evaluation of exact set similarity join techniques using gpus. *Information Systems*, 89:101485.

Bellas, C. and Gounaris, A. (2021). HySet: A Hybrid Framework for Exact Set Similarity Join using a GPU. *Parallel Computing*, 104-105:102790.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. of the 22nd IEEE Intl. Conf. on Data Engineering*, page 5.

Cruz, M. S. H., Kozawa, Y., Amagasa, T., and Kitagawa, H. (2015). GPU Acceleration of Set Similarity Joins. In *DEXA*, pages 384–398.

Cruz, M. S. H., Kozawa, Y., Amagasa, T., and Kitagawa, H. (2016). Accelerating Set Similarity Joins Using GPUs. *T. Large-Scale Data- and Knowledge-Centered Systems*, 28:1–22.

Fier, F. and Freytag, J. (2022). Parallelizing Filter-and-verification based Exact Set Similarity Joins on Multicores. *Information Systems*, 108:101912.

Li, Y., Yu, X., and Koudas, N. (2021). LES^3: Learning-based Exact Set Similarity Search. *Proceedings of the VLDB Endowment*, 14(11):2073–2086.

Mann, W., Augsten, N., and Bouros, P. (2016). An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB*, 9(9):636–647.

Quirino, R. D., Ribeiro-Júnior, S., Ribeiro, L. A., and Martins, W. S. (2017). fgssjoin: A GPU-based Algorithm for Set Similarity Joins. In *Proc. of the 19th Intl. Conf. on Enterprise Information System*, pages 152–161.

Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2016). gSSJoin: a GPU-based Set Similarity Join Algorithm. In *Proc. of the 31st Brazilian Symposium on Databases*, pages 64–75.

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2017). Fast parallel set similarity joins on many-core architectures. *J. Inf. Data Manag.*, 8(3):255–270.

Sarawagi, S. and Kirpal, A. (2004). Efficient Set Joins on Similarity Predicates. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 743–754.

Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set-similarity Joins using MapReduce. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 495–506.

Wang, X., Qin, L., Lin, X., Zhang, Y., and Chang, L. (2017). Leveraging Set Relations in Exact Set Similarity Join. *Proceedings of the VLDB Endowment*, 10(9):925–936.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-duplicate Detection. *ACM Trans. Database Syst.*, 36(3):15.