An Empirical Evaluation of Exact Set Similarity Join Techniques Using GPUs

Christos Bellas[a], Anastasios Gounaris[a]

[a]*Department of Informatics, Aristotle University of Thessaloniki, Greece*
{chribell,gounaria}@csd.auth.gr

Abstract

- “Exact set similarity join is a notoriously expensive operation”
- “There have been studies that present a comparative analysis using MapReduce or a non-parallel setting”
- “… we complement these works through […] a thorough evaluation of the state-of-the-art GPU-enabled techniques”
  - Key strengths (and weak points) of each one
- “… in real-life applications there is no dominant solution. Depending on specific dataset and query characteristics, each solution, even not using the GPU at all, has its own sweet spot”

Introduction

- “Due to inherent quadratically complexity, a set similarity join […] can take hours to complete on a single machine”
  - High dimensionality
  - Sparsity
  - Unknow data distribution
- Two main complementary approaches
  - Filtering techniques
  - Massive parallelism

Introduction

- In GPU accelerated implementations, two alternatives:
  - Transferring the whole workload onto the GPU
  - Splitting the workload between CPU and GPU

Introduction

- Contributions:
  - “… the first comprehensive presentation and comparative evaluation of GPU accelerated set similarity joins...”
  - “… extensive performance analysis using eight real world datasets”
    - “We identify the conditions under which each solution becomes the dominant one...”
  - “We provide a repository with all techniques...”
    - https://github.com/chribell/gpgpu_ssjoin_eval

Background

- Set Similarity joins
  - Avoid comparing all possible set pairs by applying filtering techniques on preprocessed data to prune as much candidate pairs as possible
  - Then to proceed to the actual verification of the remaining candidates

Background

- Data layout
  - Every dataset is a collection of multiple sets
  - Each set consists of elements called tokens
    - The data preprocessing phase involves a tokenization technique and deduplication of tokens if required
  - The input data tokens are represented by integers and are sorted by their frequency in increasing order
  - The sets in a collection are sorted first by their size and then lexicographically within each block of sets of equal size

Background

- Set Similarity functions
  - Jaccard, Dice, Cosine
  - The given normalized threshold t_n is translated to an equivalent overlap t

Background

- Filters
  - *prefix-filter* exploits the given threshold and similarity function by examining only two subsets called *prefixes*, one from each sorted set, and discards the pair if there is no overlap between the prefixes.
  - A *p*-prefix is formed by the *p* first tokens of the set, i.e. for set *r*, $p_r = |r| - t + 1$ and for set *s*, $p_s = |s| - t + 1$.

Similarity function	Definition	Equivalent Overlap
Jaccard	$\frac{\lvert r \cap s \rvert}{\lvert r \cup s \rvert}$	$\lceil \frac{\tau_n}{1+\tau_n}(\lvert r \rvert + \lvert s \rvert) \rceil$
Cosine	$\frac{\lvert r \cap s \rvert}{\sqrt{\lvert r \rvert \lvert s \rvert}}$	$\lceil \tau_n \sqrt{\lvert r \rvert \lvert s \rvert} \rceil$
Dice	$\frac{2\lvert r \cap s \rvert}{\lvert r \rvert + \lvert s \rvert}$	$\lceil \frac{\tau_n(\lvert r \rvert + \lvert s \rvert)}{2} \rceil$
Overlap	$\lvert r \cap s \rvert$	τ

Table 1: Similarity Functions (adapted from [1])

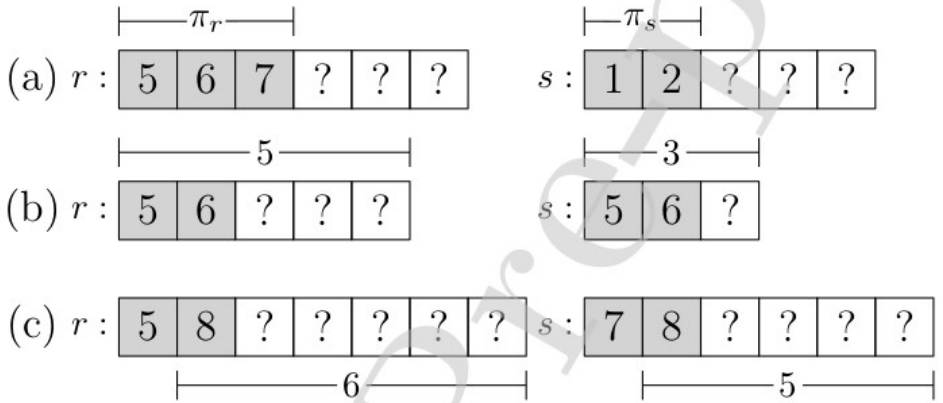


Figure 1: Filters used for candidate pruning: (a) prefix, (b) length, (c) positional

Background

- Algorithm outline
  - “Index nested loop join consisting of three steps”
    - Index lookup
    - Filtering pre-candidates
    - Verification
  - In self-joins
    - Incremental index building leads to more pairs to be pruned earlier

Algorithm 1 Filter - Verification Framework

Input: Sorted set collections R, S, a threshold τ
Output: A set *pairs* containing all similar pairs

```
1:  I ← construct_inverted_index(S)
2:  for each set r ∈ R do
3:      C ← {}
4:      for each token t ∈ prefix(r, τ) do
5:          for each set s ∈ I[t] do
6:              if not length/positional filter(r, s, τ) then
7:                  C ← C ∪ {s}
8:              end if
9:          end for
10:     end for
11:     for each set s ∈ C do
12:         if verify(r, s, τ) then
13:             pairs ← pairs ∪ (r, s)
14:         end if
15:     end for
16: end for
17: return pairs
```

R, S	Collections of sets to be joined
r_i (resp.s_j)	a token set from R (resp. S)
τ_n	Normalized similarity threshold
τ	Equivalent overlap
$C \subseteq R \times S$	Set of candidate pairs

CUDA Overview

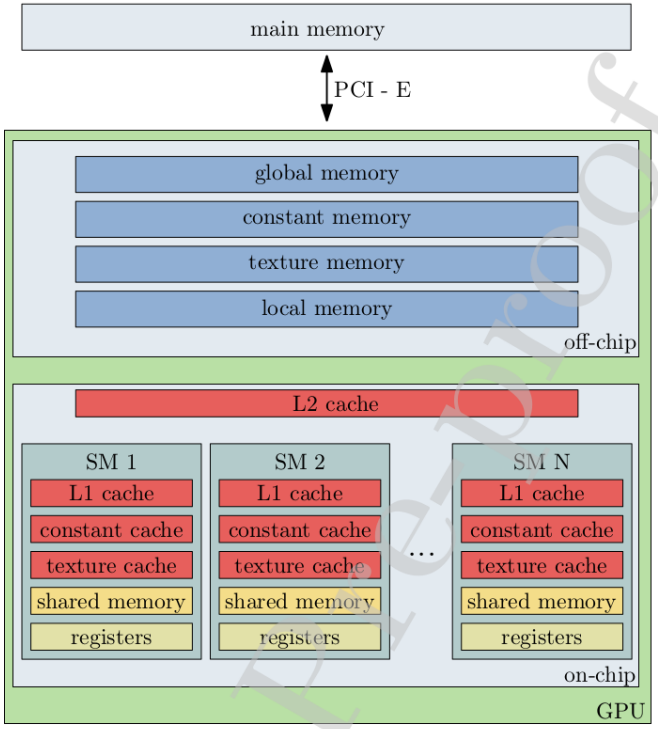


Figure 2: Memory hierarchy for CUDA-enabled GPUs (taken from [18]).

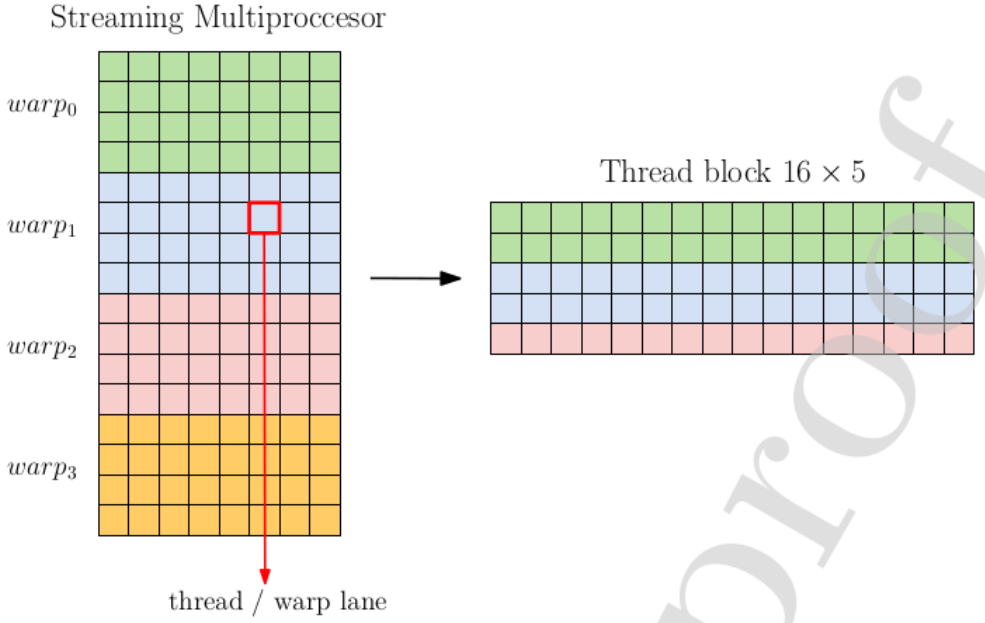


Figure 3: A warp-to-thread block mapping example.

Algorithms & Techniques
CPU

- Three best CPU algorithms, as in Mann et. al.*
  - AllPairs (ALL)
    - First to implement prefix and length filtering
  - PPJoin (PPJ)
    - Extends ALL with positional filtering
  - GroupJoin (GRP)
    - Extends PPJ. Each group of sets with identical prefix are grouped together and handled as a single set
      - Faster indexing as it discards candidate pairs in batches

* Actually seven algorithms

  - Including MPJoin and MPJoin-PEL from our Prof Leonardo Ribeiro

Algorithms & Techniques CPU

Discussion

“There are three key observations provided in [1]. First, all the evaluated algorithms have small performance differences except those which involve sophisticated filtering. Second, efficient verification yields significant performance speedups and renders complex filters inefficient. Finally, candidate generation is indicated as the main bottleneck especially for the techniques that employ the prefix filter.”

[1] : W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, Proceedings of the VLDB Endowment 9 (9) (2016) 636–647. URL http://www.vldb.org/pvldb/vol9/p636-mann.pdf

Algorithms & Techniques CPU-GPU

- CPU-GPU co-processing
  - Which component each phase is assigned to ?
    - GPUs are used to accelerate compute-intensive applications by processing data in a predefined memory space
    - Due to the fact that for a probe set, an arbitrary number of candidate pairs may be generated, the filtering phase remains a CPU task. Verification → GPU.

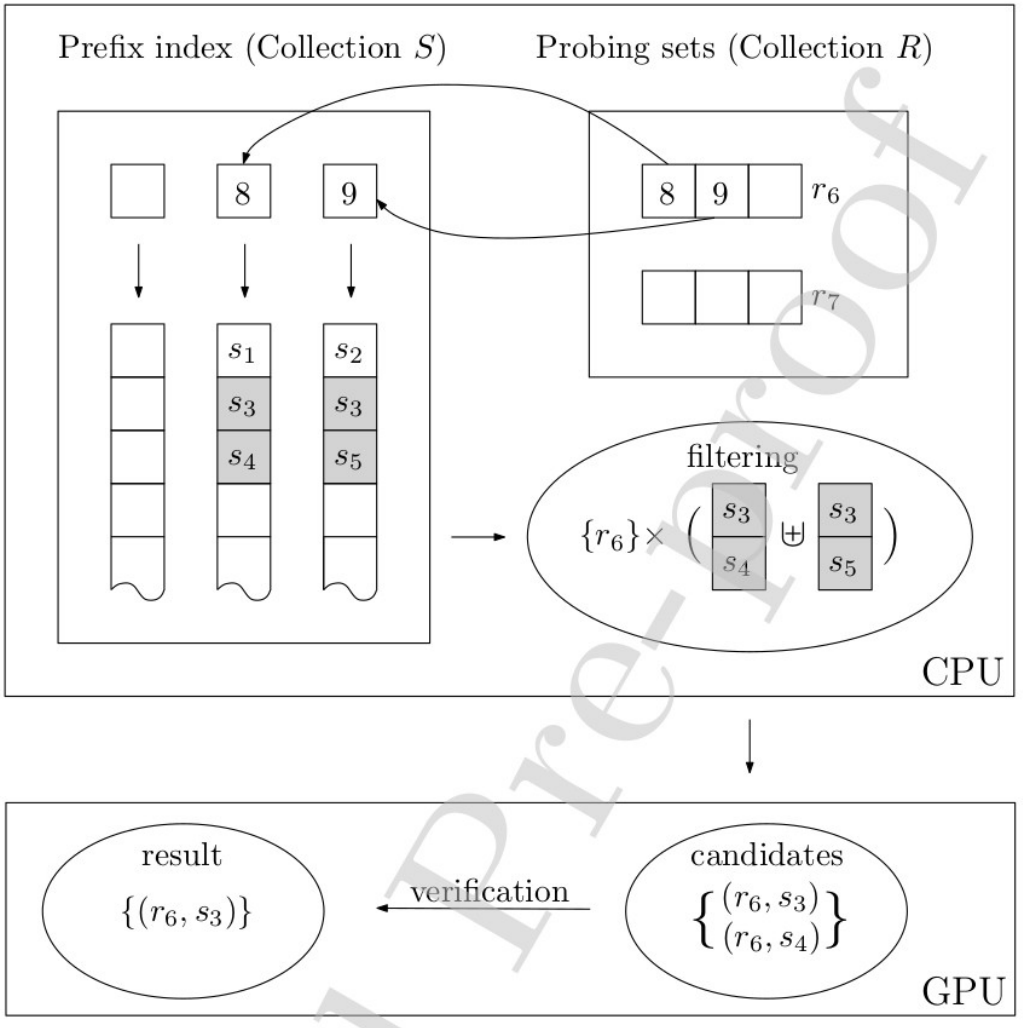


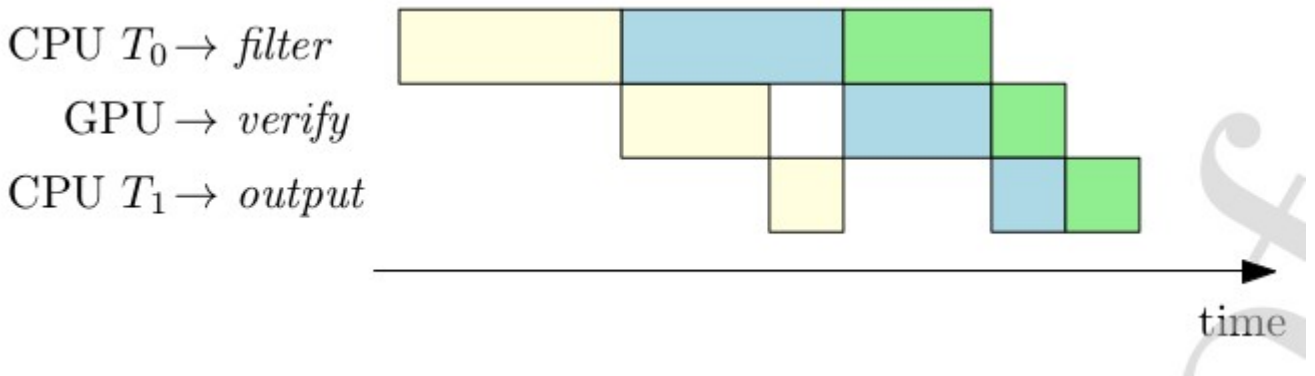
Figure 4: Splitting the workload between CPU (candidate generation) and GPU (candidate verification). The workload representation is adapted from [1].

Algorithms & Techniques CPU-GPU

- The main challenges addressed involve
  - Appropriate data layout on device memory
  - Efficient usage of shared memory
  - Thread workload
    - One thread → multiple candidate paris
    - One thread → one candidate pair
    - One thread → part of a single candidate pair

Algorithms & Techniques CPU-GPU

- Since the limited GPU memory is the most dominant constraining factor, the workload is divided into chunks
  - Execution overlap between filtering and verification



Algorithms & Techniques CPU-GPU

Discussion

> “Even though the average verification runtime is reported as constant for most datasets in [1], the work in [10] proves that employing the GPU for this part improves overall performance. Selecting the appropriate verification technique depending on the average set size, and performing further fine tuning can completely hide the verification time due to the execution overlap, especially where the number of candidates is in billions. However, since the runtime is bounded by the execution time for the filtering phase and due to Amdahl’s law, the achieved speedups are lower than an order of magnitude, e.g., if filtering takes 40% of the total time, the maximum speed-up through parallelizing only the verification phase cannot exceed 2.5X.”

[10] : C. Bellas, A. Gounaris, Exact set similarity joins for large datasets in the gpgpu paradigm, in: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN’19, ACM, New York, NY, USA, 2019

Algorithms & Techniques
GPU standalone

- “Ribeiro et. al. with their successive works in [9,23,24] perform the complete exact set similarity join on the GPU.”

[9] : S. Ribeiro-Junior, R. D. Quirino, L. A. Ribeiro, W. S. Martins, Fast parallel set similarity joins on many-core architectures, Journal of Information and Data Management 8 (3) (2017) 255.

[23] : S. Ribeiro-Júnior, R. D. Quirino, L. A. Ribeiro, W. S. Martins, gssjoin: a gpu-based set similarity join algorithm., in: SBBD, 2016, pp. 64–75.

[24] : R. D. Quirino, S. Ribeiro-Junior, L. A. Ribeiro, W. S. Martins, Efficient filter-based algorithms for exact set similarity join on gpus, in: International Conference on Enterprise Information Systems, Springer, 2017, pp. 74–95.

Algorithms & Techniques
GPU standalone

- GPU-based Set Similarity Join (gSSJoin)
  - Constructs a static inverted index over all tokens
  - For each probe set:
    - Intersection count between input set and every other set
      - Workload evenly distributed among GPU threads
        - Logical vector E
    - Jaccard similarity of every pair is calculated and stored in global memory
    - Pairs with similarity above threshold are transferred to main memory after stream compaction is employed

Algorithms & Techniques
GPU standalone

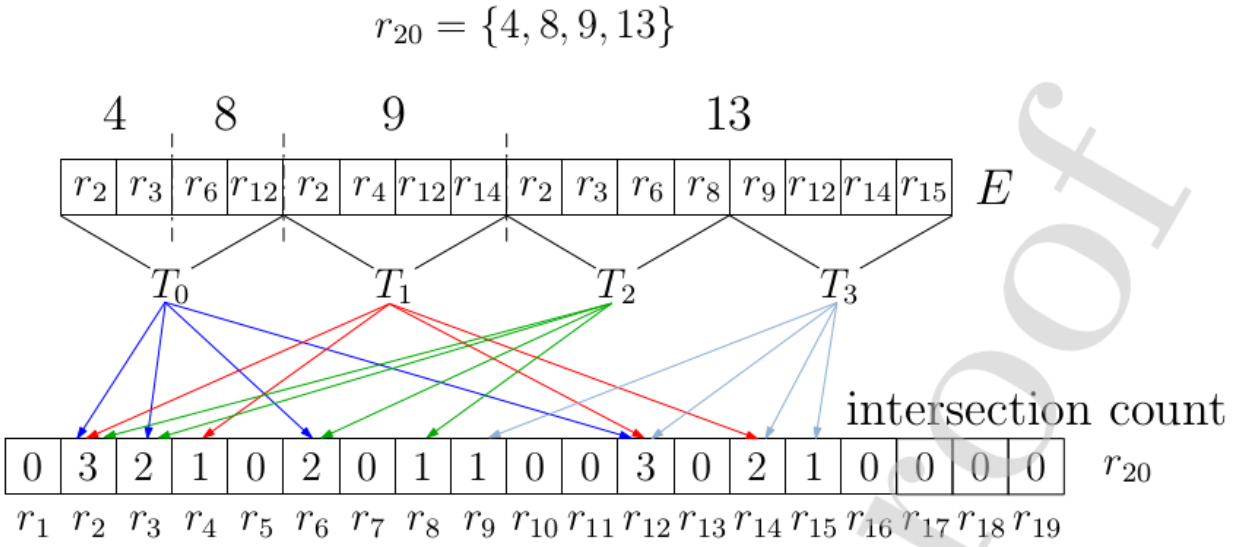


Figure 6: gSSJoin workload allocation example for four GPU threads.

Algorithms & Techniques
GPU standalone

- GPU-based Set Similarity Join (gSSJoin)
  - Efficient brute force approach
  - Remains more robust to variations of threshold values and token frequency distribution
    - Its strongest points
  - Main drawback is the kernel launch overhead, which starts dominating in medium and large datasets (>1M)

Algorithms & Techniques
GPU standalone

- “Filter-based gSSJoin” (fgSSJoin)
  - “Extends” gSSJoin by encapsulating a filtering phase
    - fgssjoin is completely different from gssjoin…
  - In addition, a block partitioning scheme is adopted in order to process collections of arbitrary size, which, by length filtering, enables the skipping of whole block-to-block comparisons.

Algorithms & Techniques
GPU standalone

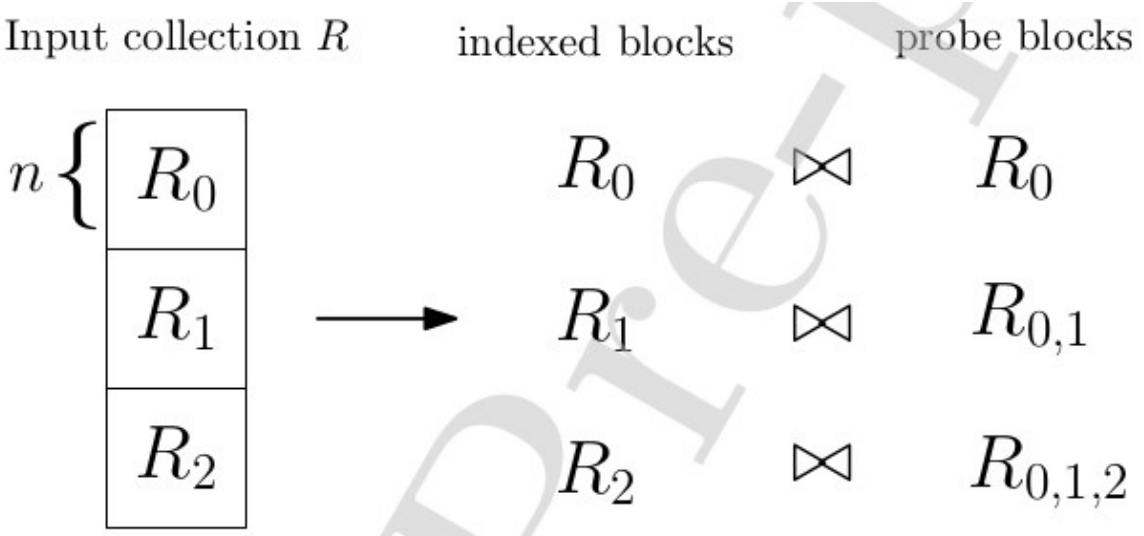


Figure 7: Block partitioning scheme used in fgSSJoin and sf-gSSJoin.

Algorithms & Techniques
GPU standalone

- Every pair that has a non-zero partial intersection count undergoes full verification, whereas the rest is pruned
  - Not true, lenght and positional filtering during verification phase not considered…
- The workload is distributed similarly to gSSJoin
  - Again, not true, gssjoin launches as much kernels as probe sets whereas fgssjoin only launches one kernel (for each of the index, filter, verify phases)…

Algorithms & Techniques
GPU standalone

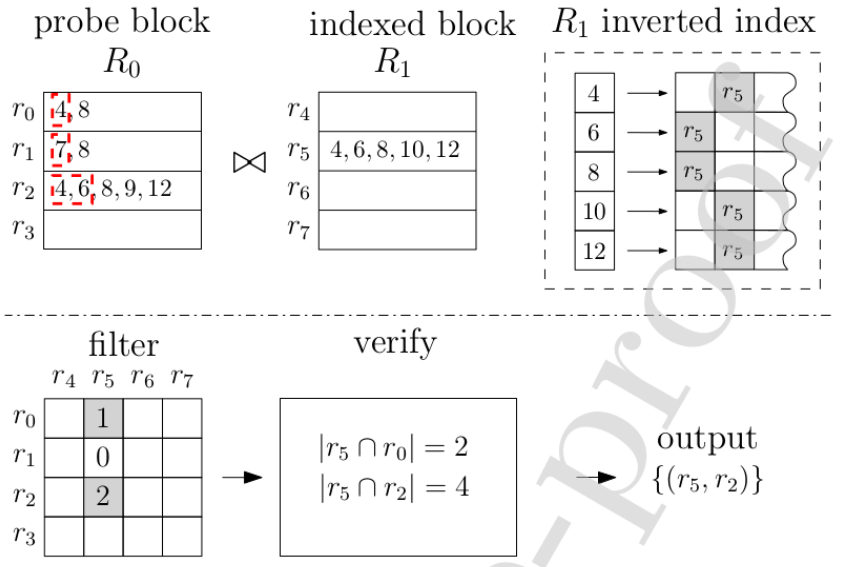


Figure 8: fgSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

Algorithms & Techniques
GPU standalone

- Drawback: $O(n^2)$ space complexity
- Scalability issues arises when processing the quadratic memory space required to store the intermediate intersection counts. More specifically, after each probe block call, this space must be reset to zero to ensure correctness. As a result, the accumulated overhead may become a significant issue, as the dataset size increases.

Algorithms & Techniques
GPU standalone

- Size-filtered gSSJoin (sf-gSSJoin)
  - Adopts the same block partitioning scheme as fgSSJoin
  - In contrast with fgSSJoin, sf-gSSJoin directly calculates the complete intersection counts between probe sets r_0, r_1, r_2 and the indexed set r_5 in the first pass.

Algorithms & Techniques
GPU standalone

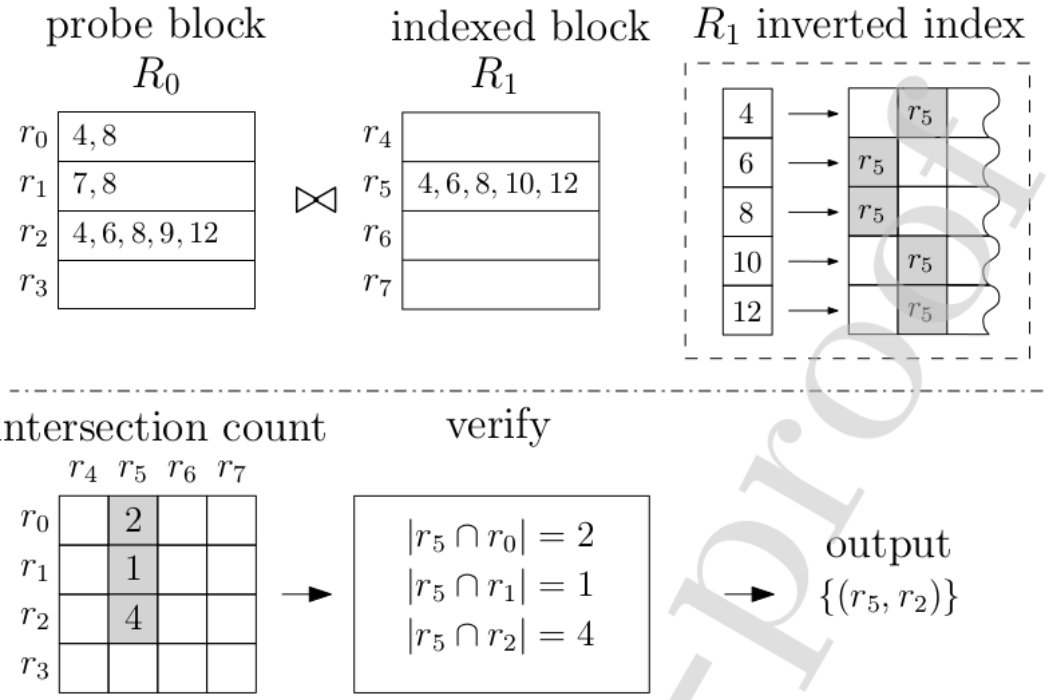


Figure 9: sf-gSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

Algorithms & Techniques
GPU standalone

- Bitmap filter
  - In their work, Mann et al. point out the prefix-filter as the main bottleneck of the filter-verification algorithms and underline that future filtering techniques should invest in faster and simpler methods
  - Driven by this, the authors of [25] propose a new low overhead filtering technique called bitmap filtering

[25] : E. F. Sandes, G. Teodoro, A. C. Melo, Bitmap filter: Speeding up exact set similarity joins with bitwise operations, arXiv preprint arXiv:1711.07295.

Algorithms & Techniques
GPU standalone

- Bitmap filter
  - Essentially, the bitmap filter uses hash functions to create signature bitmaps of size *b* for the input collection sets. From this we can deduce an overlap upper bound for a candidate pair.

$$|r \cap s| \leq \lfloor \frac{|r| + |s| - popcount(b_r \oplus b_s)}{2} \rfloor$$

Algorithms & Techniques
GPU standalone

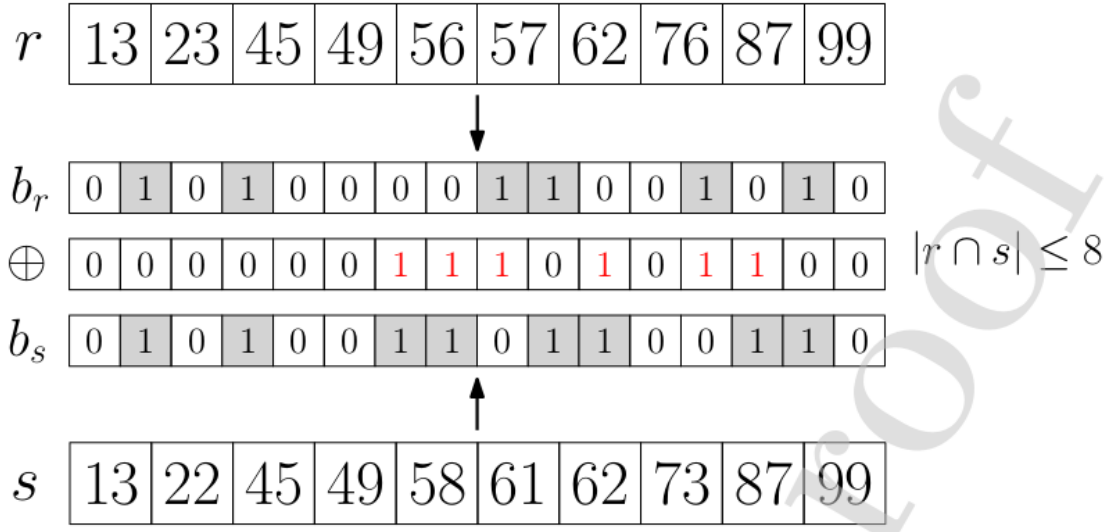


Figure 10: Candidate pair r, s can be safely pruned for $\tau_n = 0.95 \rightarrow \tau = 10$ since its expected overlap upper bound is 8

Algorithms & Techniques
GPU standalone

- Bitmap filter
  - Simple implementation, bitmap filter in GPU, verification in CPU
  - Tranferring of candidate pairs to CPU may be the bottleneck
  - Authors implementations use fgssjoin block partitioning scheme, contrary to original Bitmap implementation, to mitigate scalability issues, keeping the whole join process in GPU

Algorithms & Techniques GPU standalone

- Feature comparison

	index	filters	verification
CPU	See Algorithm 1		
CPU-GPU	in CPU	as in CPU	multiple workload alternatives
gSSJoin	in GPU	-	using atomic operations
fgSSJoin	in GPU	prefix, length	using atomic operations
sf-gSSJoin	in GPU	length	using atomic operations
bitmap	-	bitmap, length	multiple workload alternatives

Table 3: Feature comparison between the evaluated techniques.

Datasets

- **AOL**:
  - Query log data from the AOL search engine. Each set represents a query string and its tokens are search terms.
- **BMS-POL**:
  - Purchase data from an e-shop. Each set represents a purchase and its tokens are product categories in that purchase.
- **DBLP**:
  - Article data from DBLP bibliography. Each set represents a publication and its tokens are character 2-grams of the respective concatenated title and author strings.
- **ENRON**:
  - Real e-mail data. Each set represents an e-mail and its tokens are words from either the subject or the body field.

Datasets

- **KOSARAK**:
  - Click-stream data from a Hungarian on-line news portal. Each set represents a user behavior and its tokens are links clicked by that user.
- **LIVEJOURNAL**:
  - Social media data from LiveJournal. Each set represents a user and its tokens are interests of that user.
- **ORKUT**:
  - Social media data from ORKUT network. Each set represents a user and its tokens are group memberships of that user.
- **TWITTER**:
  - Social data from Twitter. Each set represents a user tweet and its tokens are character 2-grams of the respective tweet text.

Dataset	Cardinality	Avg set size	# diff tokens
AOL	$1.0 \cdot 10^7$	3	$3.9 \cdot 10^6$
BMS-POS	$5.1 \cdot 10^5$	6.5	1657
DBLP (100K)	$1.0 \cdot 10^5$	88	7205
DBLP (200K)	$2.0 \cdot 10^5$	88	8817
DBLP (300K)	$3.0 \cdot 10^5$	88	$1.0 \cdot 10^4$
DBLP (1M)	$1.0 \cdot 10^6$	88	$1.5 \cdot 10^4$
DBLP (Complete)	$6.1 \cdot 10^6$	88	$2.7 \cdot 10^4$
ENRON	$2.5 \cdot 10^5$	135	$1.1 \cdot 10^6$
KOSARAK	$1.0 \cdot 10^6$	8	$4.1 \cdot 10^4$
LIVEJOURNAL	$3.1 \cdot 10^6$	36.5	$7.5 \cdot 10^6$
ORKUT	$2.7 \cdot 10^6$	120	$8.7 \cdot 10^6$
TWITTER	$1.6 \cdot 10^6$	75	$3.7 \cdot 10^4$

Table 4: Datasets characteristics.

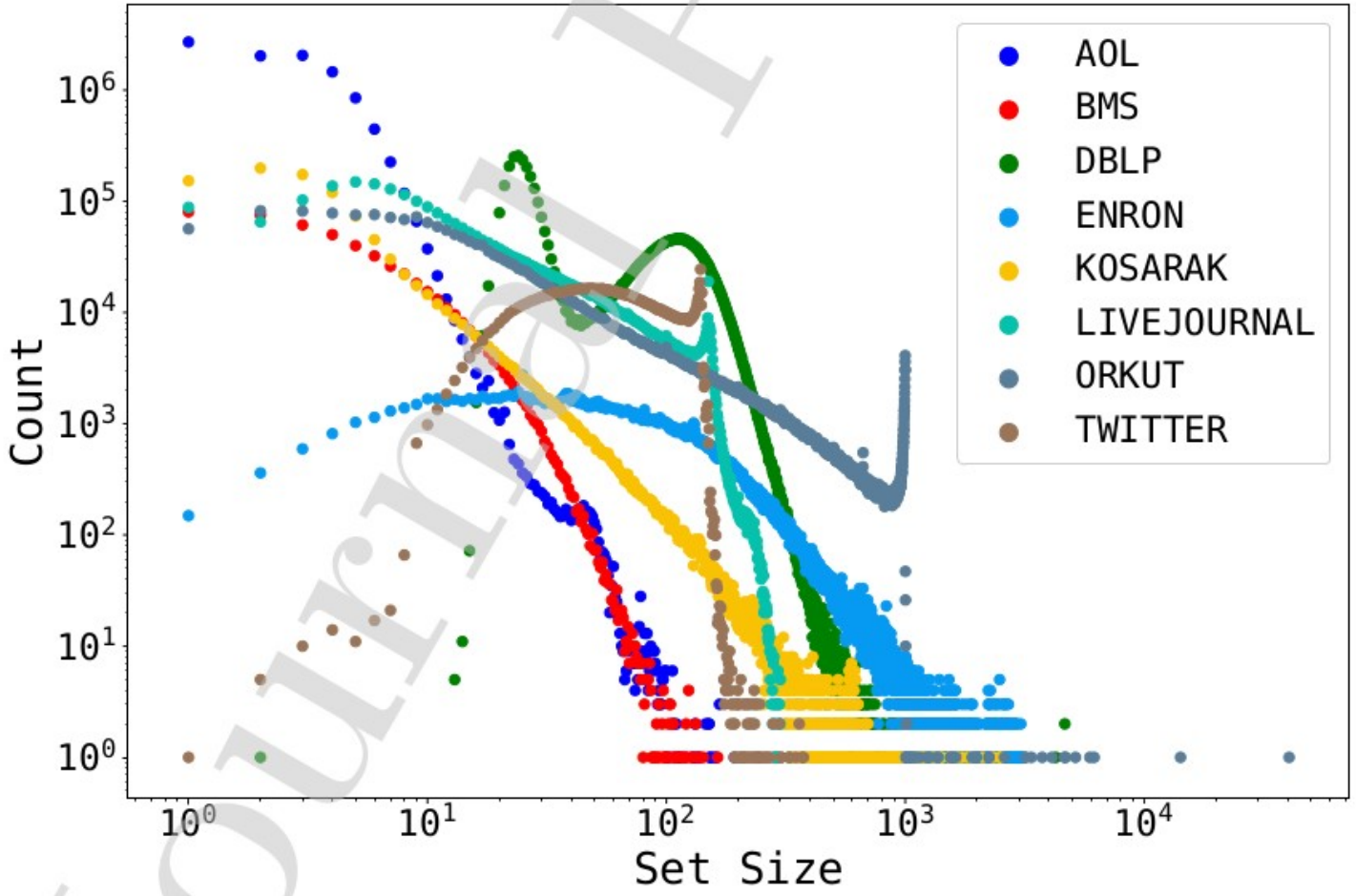


Figure 11: Datasets set size distribution

Experiments

- The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3GHz, 32 GB RAM at 2400MHz and an NVIDIA Titan XP on CUDA 9.1, with 3840 CUDA cores, 12 GB of global memory and a 384-bit memory bus width.
- We focus on self-joins using the Jaccard similarity and experiment on main-memory solutions on a single machine.
- For simplicity, we perform an aggregation on top of the set similarity join, to measure the count of the results, in section 4.5 we provide empirical evidence why this does not affect the results presented.

Experiments

- **Main experiment**
  - We compare the state-of-the-art CPU standalone implementation of Mann [1] against its GPU-accelerated version [10], noted as CPU-GPU and the GPU standalone solutions described in [9, 23, 24]. In Figure 12, we present the best join times measured for all. Each time reported for the CPU is the overall best among the three algorithms (i.e., ALL, PPJ, GRP) and therefore the best we can achieve in our setup.
  - Respectively, for the CPU-GPU co-processing solution, each time reported is the overall best among of the three CPU algorithms and the best GPU verification techniques described in [10].
  - Finally, for gSSJoin and its variations, we report the sum of time for all the GPU operations required to perform the set similarity join.

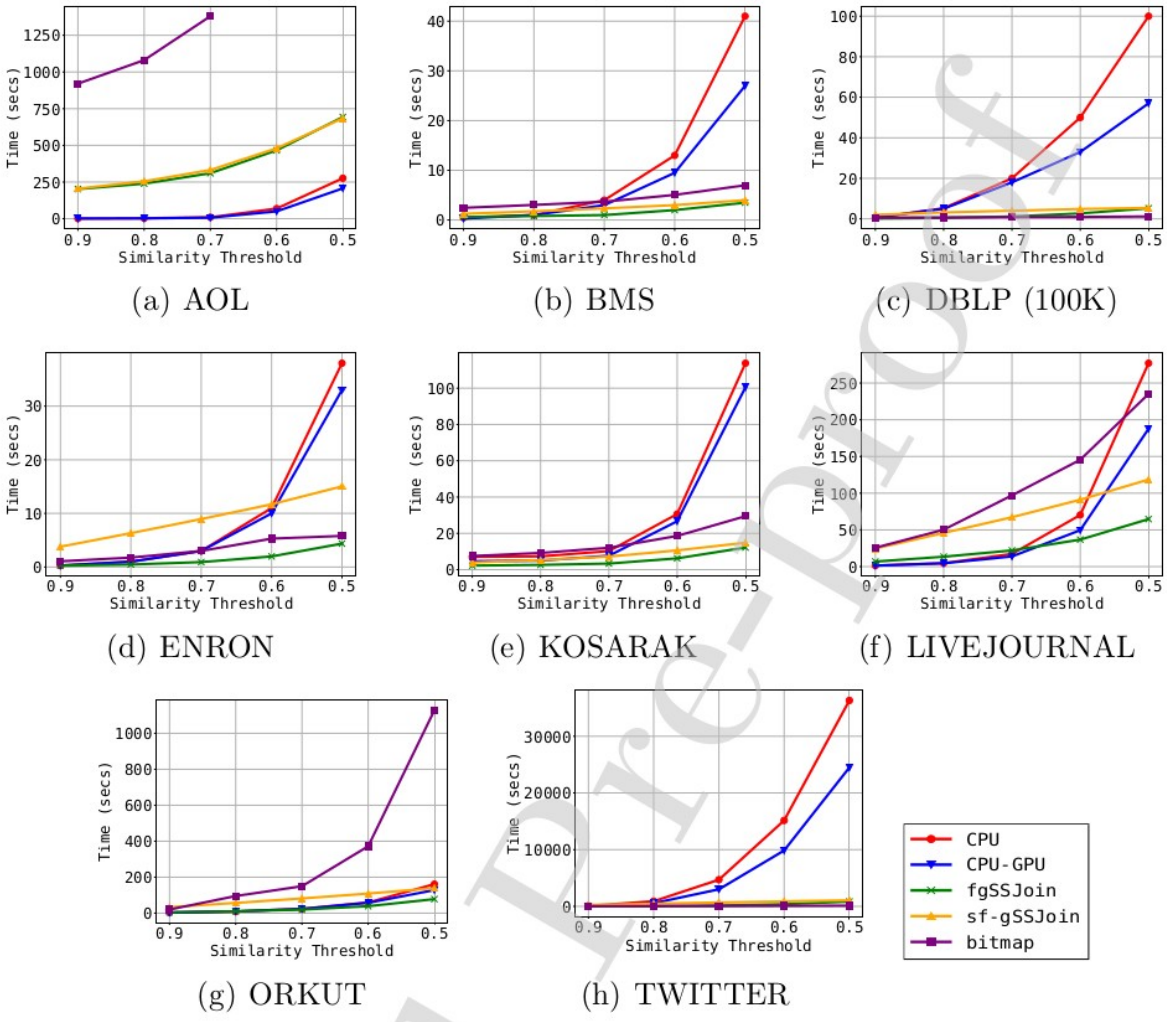


Figure 12: Comparison between the best times for different thresholds.

Experiments

- As shown in Figure 12, for the majority of datasets and high thresholds, i.e. the threshold range is in [0.8, 0.9], invoking the GPU does not yield any performance speedup. Given the fact that in high thresholds, filtering is quite effective, hence the number of candidates is quite small, employing the GPU seems redundant, specially for small datasets.
- On the other hand, as the threshold value decreases ([0.5,0.7]), GPU standalone solutions are quite effective in general. This is due to the fast intersection count conducted in parallel which accelerates the verification phase.
- With the GPU standalone solutions, we observe that are rather inefficient for the biggest dataset (AOL); in contrast, they perform better for the TWITTER dataset. This calls for a deeper analysis. (Another way of saying: “we didn’t understand it”)
- Also, we have omitted the
- We omitted gSSJoin solution, due to its inability to scale.

Experiments

- **Quadratic space overhead of fgSSJoin and sf-gSSJoin**
  - Considering that an intersection count is a 4-byte integer, the required memory space to store all counts for n = 15000 is 900MB. In our experiments, we use the cudaMemset to clear the intersection count space. **This entails a 2 ms overhead per probe call.**
  - For datasets with a high proportion of similar set sizes, such as AOL, length filtering on block level is ineffective. This results in a higher number of GPU calls, increasing the clear operation overhead.

Experiments

τ_n	# probes	accumulated $O(n^2)$ space	time (secs)
0.5	138841	125 TB	261
0.6	94747	85 TB	178
0.7	64928	58 TB	122
0.8	50431	45 TB	95
0.9	42628	38 TB	80

Table 5: Quadratic space overhead for AOL

Experiments

- **Quadratic space overhead of fgSSJoin and sf-gSSJoin**
  - The bottom line is that in large datasets, especially when, due to the dataset size and set size, the length filter cannot prune many pairs, the overhead associated with the quadratic space complexity is not outweighed by any benefits compared to CPU solutions for thresholds above 0.6, and, overall fgSSJoin and sf-gSSJoin are not the optimal GPU-enabled techniques.
  - Concrete numbers are presented in Figure 13 and Table 5. The magnitude of the impact of quadratic space on runtime is dictated by two factors: input collection size and the number of pruned blocks.

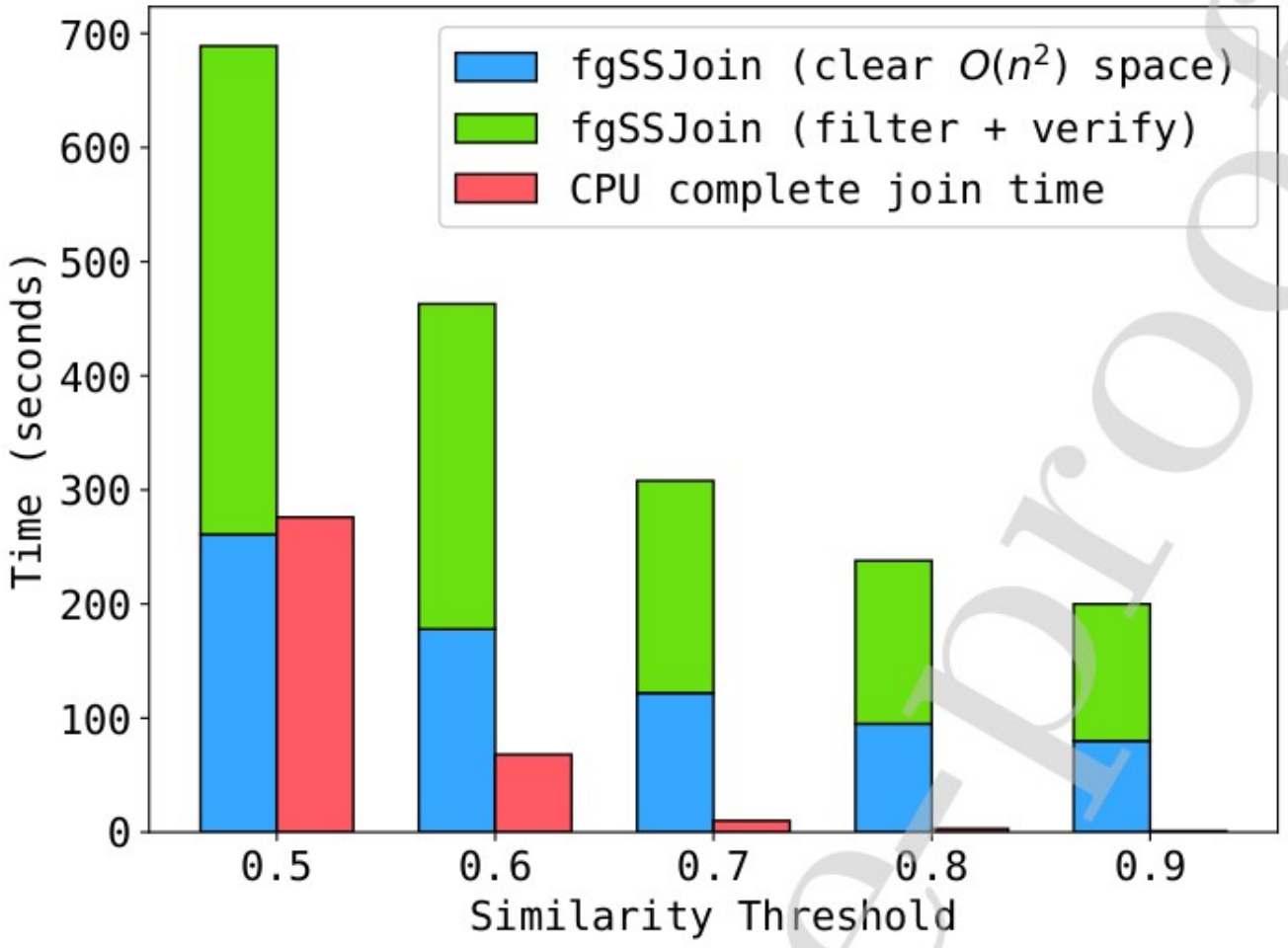
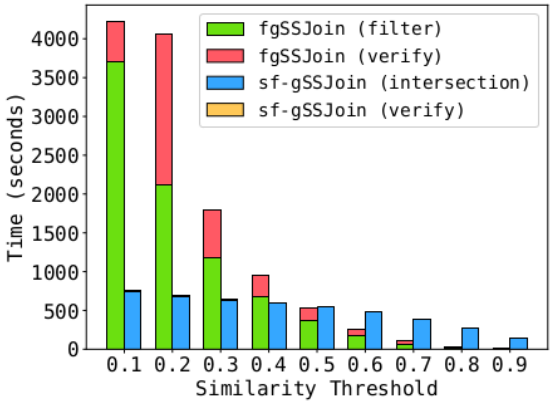
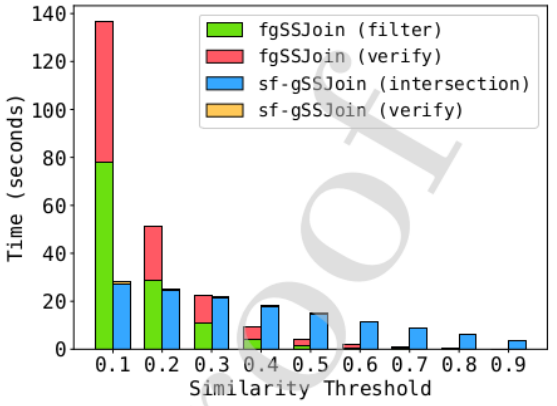


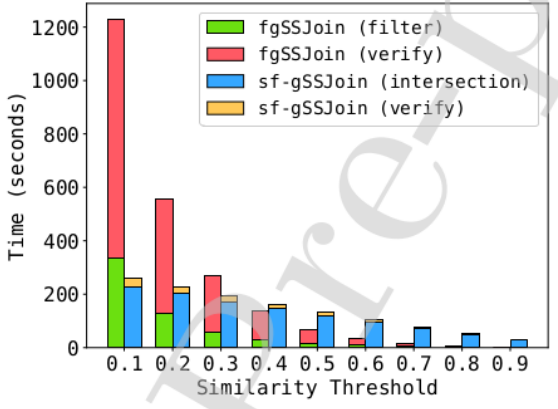
Figure 13: Quadratic space overhead



(a) DBLP (1M)



(b) ENRON



(c) ORKUT

Figure 14: Comparison between the best times of *fgSSJoin* vs *sf-gSSJoin* for different thresholds.

Experiments

- **Bitmap performance**
  - Bitmap filtering can be seen as an alternative to a prefix-based approach. However, the technique is highly dependent on the dataset characteristics, since sets may produce similar bitmaps even if they do not share similar tokens; in general, the probability of such undesired collisions is increased whenever the number of set tokens is increased and the bitmap size is reduced.

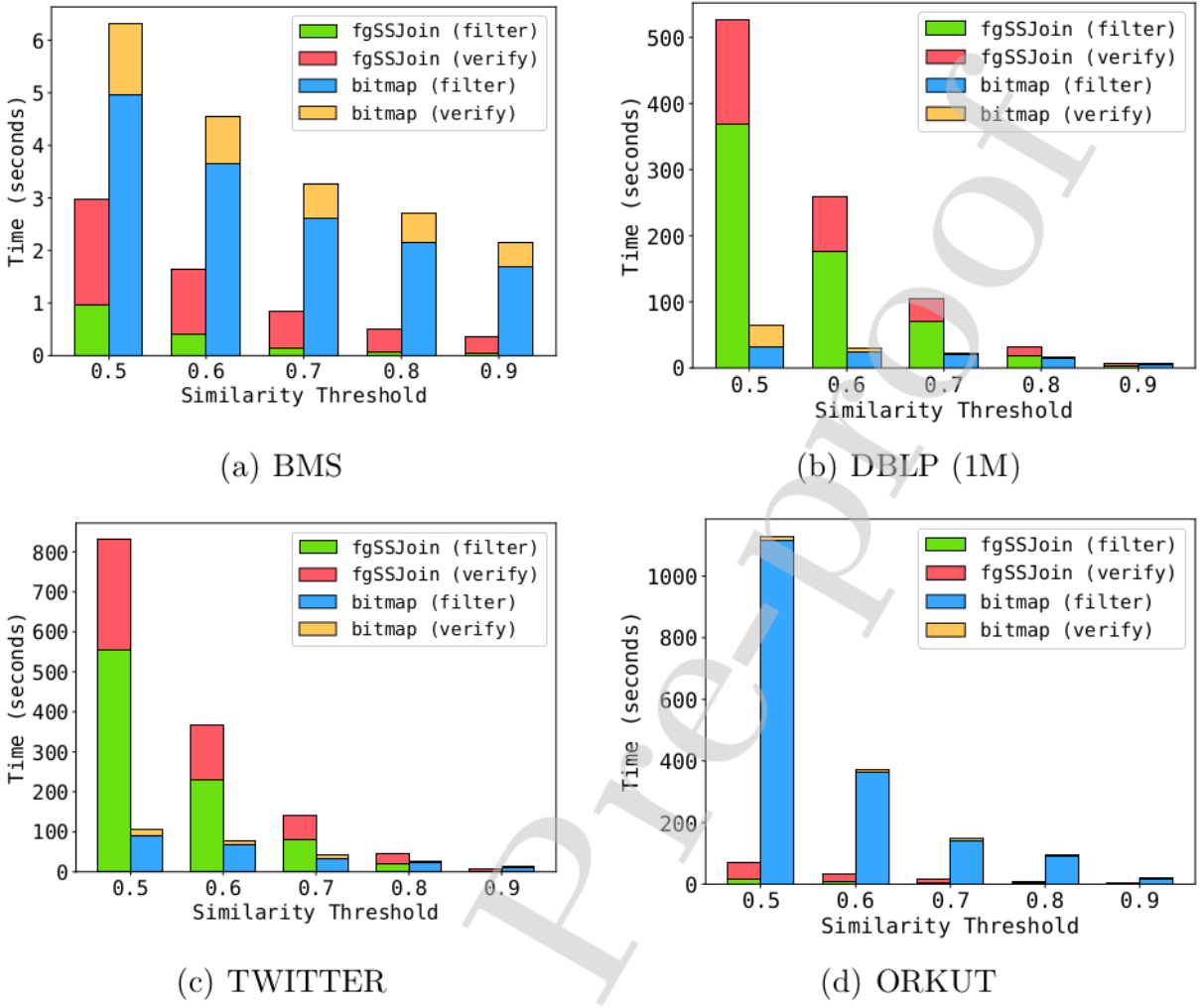
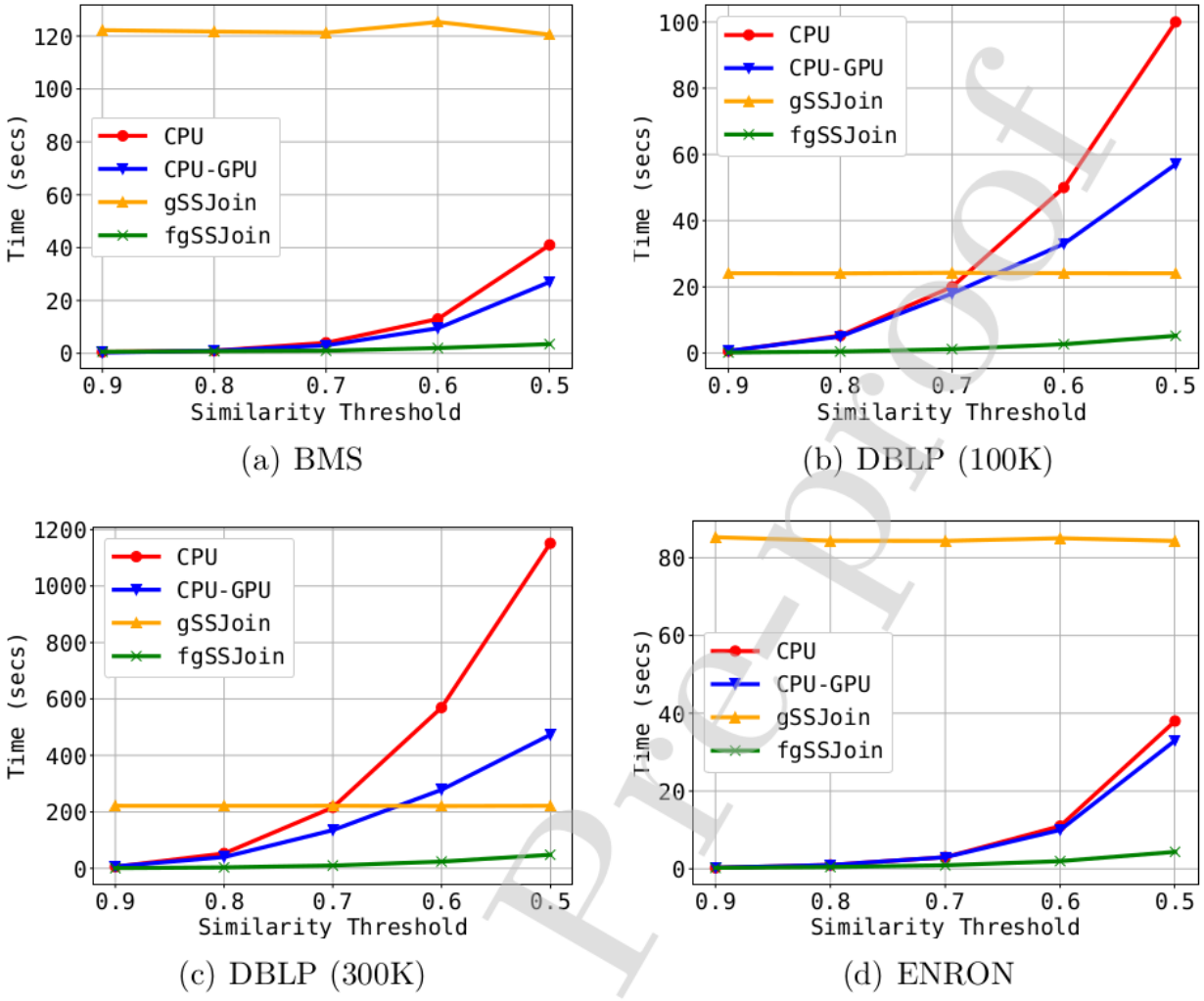


Figure 15: Comparison between the best times *fgSSJoin* vs *bitmap* for different thresholds.

Experiments

- **gSSJoin launch overhead**
  - As stated in the technique presentation, gSSJoin launches a sequence of separate GPU kernel calls per probe set in order to conduct the set similarity join operation. Given the fact that launching a large number of small kernels is considered a bad practice, executing gSSJoin results into an accumulated launch and execution overhead.



(a) BMS (b) DBLP (100K)

(c) DBLP (300K) (d) ENRON

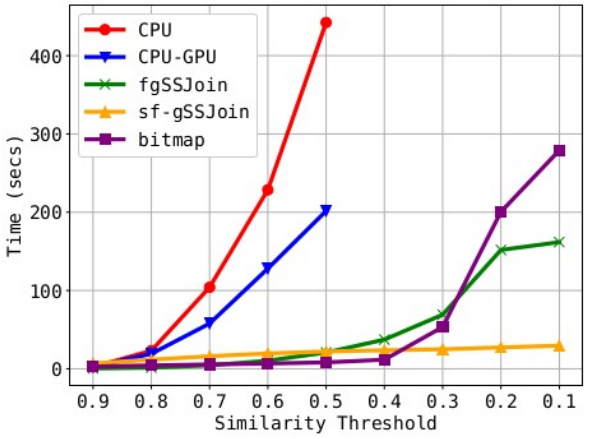
Figure 16: gSSJoin runtimes for different thresholds compared to other techniques

Additional experiments

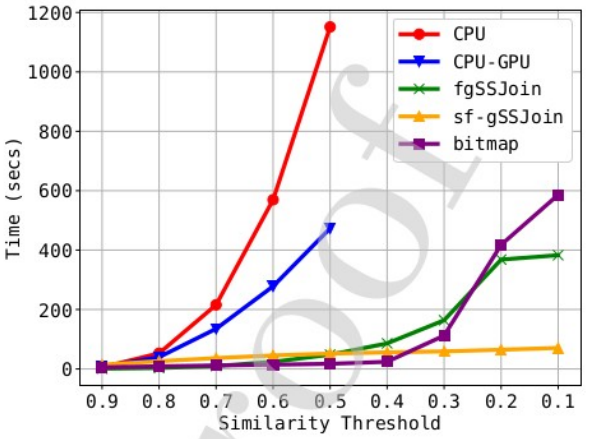
- **Larger and synthetic data**
  - Experiments with larger portions of **DBLP**
  - We can see that the speed-ups of the GPU standalone solutions are much more evident as the collection size increases... This is further verified from partial results regarding the full DBLP dataset... The benefits from employing GPUs are tangible even for larger thresholds.

τ_n	CPU	CPU-GPU	fgSSJoin	bitmap	sf-gSSJoin
0.8	-	32387	1259	607	10353
0.9	7244	2814	230	303	5227

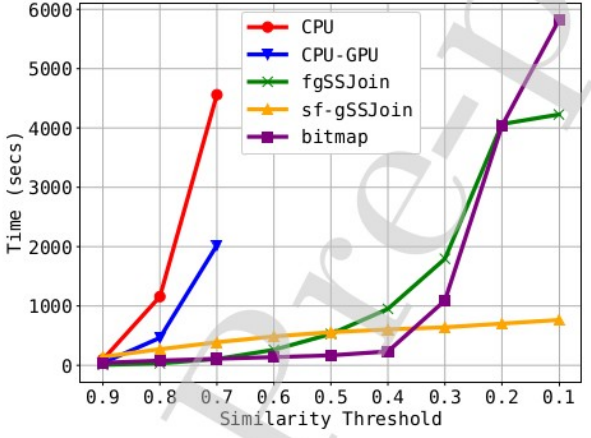
Table 7: Runtimes for the complete DBLP consisting of 6.1M sets (in secs)



(a) DBLP (200K)



(b) DBLP (300K)



(c) DBLP (1M)

Figure 17: Comparison between the best times for larger portions of the DBLP dataset.

Additional experiments

τ_n	CPU	CPU-GPU	fgSSJoin	bitmap	sf-gSSJoin
0.8	-	32387	1259	607	10353
0.9	7244	2814	230	303	5227

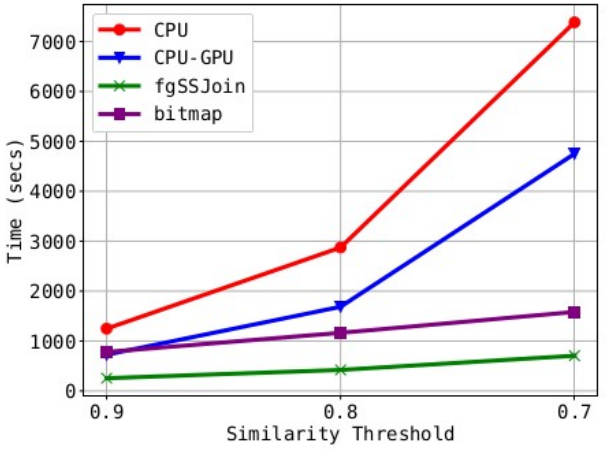
Table 7: Runtimes for the complete DBLP consisting of 6.1M sets (in secs)

Additional experiments

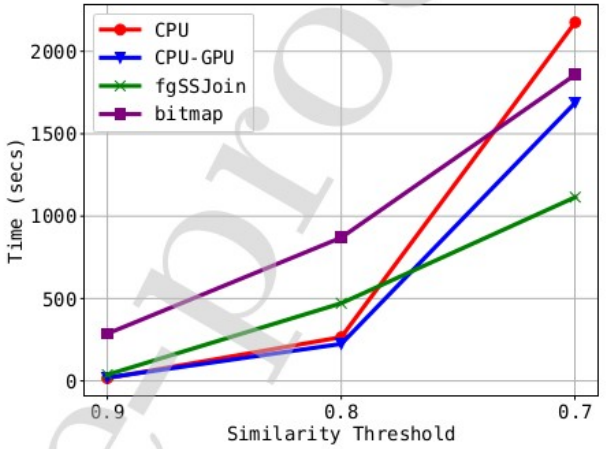
- **Larger and synthetic data**
  - Experiments with synthetically enlarged versions of **BMS**, **ENRON** and **LIVEJOURNAL**

Dataset	Cardinality	Avg set size	# diff tokens
BMS (x25)	$1.3 \cdot 10^7$	6.5	1681
ENRON (x25)	$6.1 \cdot 10^6$	135	$1.1 \cdot 10^6$
LIVEJOURNAL (x5)	$1.5 \cdot 10^7$	36.5	$7.5 \cdot 10^6$

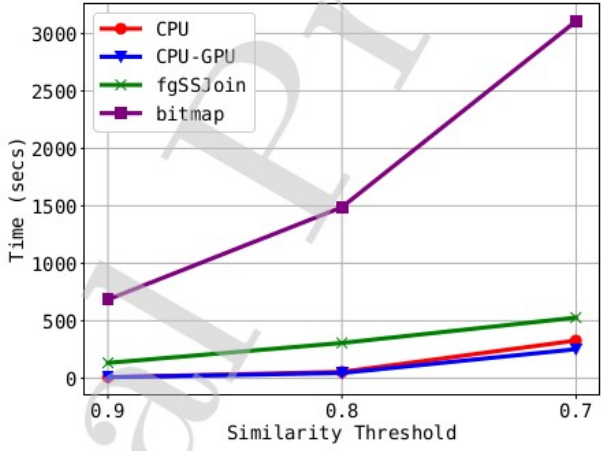
Table 6: Larger datasets' characteristics. Within parentheses is the increase factor.



(a) BMS (x25)



(b) ENRON (x25)

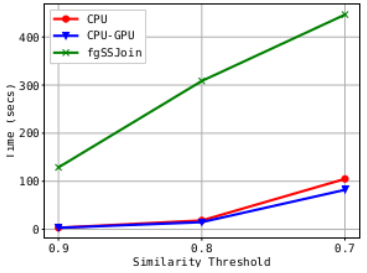


(c) LIVEJOURNAL (x5)

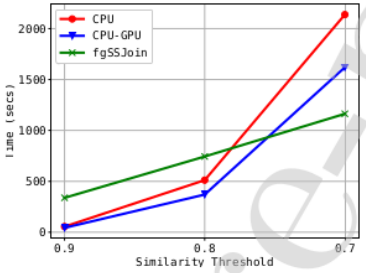
Figure 18: Comparison between the best times for the increased datasets.

Dataset size	5M, 10M, 20M
Number of different tokens	50K, 500K
Average set size	5, 25

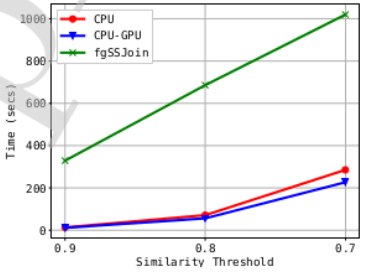
Table 8: Synthetic datasets' characteristics



(a) 10M - 50K - 5

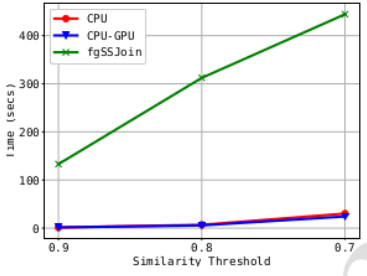


(b) 10M - 50K - 25

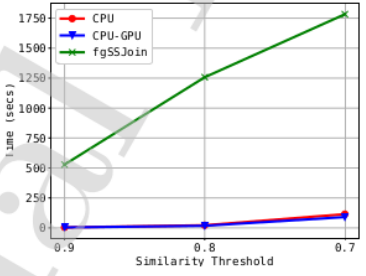


(c) 10M - 500K - 25

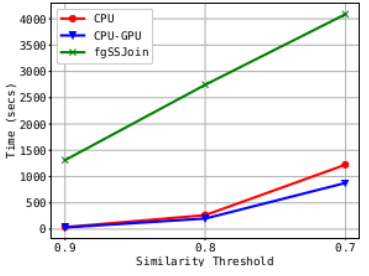
Figure 19: Comparison between the best times for synthetic datasets with fixed dataset size.



(a) 10M - 500K - 5



(b) 20M - 500K - 5



(c) 20M - 500K - 25

Figure 20: Comparison between the best times for synthetic datasets with fixed number of different tokens.

Summarizing

- **CPU**
  - Strong points: handling high and very high thresholds where (prefix) filtering is effective, e.g., no small average set size combined with high token cardinality
  - Dominant cases: very high thresholds unless small dataset and small average set size, where moving the dataset to the GPU and perform extremely quick analysis there is more beneficial
  - Weak points: handling threshold values lower than 0.8

Summarizing

- **CPU-GPU**
  - Strong points: handling large datasets (but cannot scale with decreasing threshold values).
  - Dominant cases: medium-high thresholds

    $(0.5 < t < 0.8)$ and large datasets.
  - Weak points: thresholds; and (ii) medium thresholds and no large datasets, because in these cases, the filtering phase (not parallelized by CPU-GPU) dominates and/or is significant

Summarizing

- **fgSSJoin**
  - Strong points: handling medium and high thresholds

    (0.5 < t < 0.8) but not large datasets, where the initial index-based prefix filtering is ineffective
  - Dominant cases: Same as the cases in strong points
  - Weak points: handling larget datasets due to the quadratic complexity in the block size and the associated overhead

Summarizing

- **sf-gSSJoin**
  - Strong points: handling not very big datasets combined with low thresholds
  - Dominant cases: threshold below 0.5, where sophisticated filtering, e.g. prefix ones, is not effective
  - Weak points: handling the cases, where prefix-based filtering can manage to prune a significant portion of candidate pairs, e.g., queries with high thresholds, especially when combined with high average set size

Summarizing

- **bitmap**
  - Strong points: handling cases where bitmap signatures are effective, i.e., high set size and not high number of different tokens
  - Dominant cases: medium thresholds and dataset size combined with low number of different tokens and high average set size
  - Weak points: scalability in the dataset size

Summarizing

τ_n	Dataset size - Average set size				
	small - small	small - large	medium - large	large - small	large - large
Very high(0.9)	fgSSJoin / CPU	CPU	fgSSJoin / CPU	CPU	CPU / CPU-GPU
High(0.8)	fgSSJoin	fgSSJoin	fgSSJoin / CPU-GPU	CPU-GPU	CPU-GPU
Medium(0.5-0.7)	fgSSJoin	fgSSJoin	fgSSJoin / bitmap	CPU-GPU	fgSSJoin / CPU-GPU
Low(<0.5)	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	N/A

Result size	Dataset size - Average set size				
	small - small	small - large	medium - large	large - small	large - large
$< 10^4$	N/A	fgSSJoin / CPU	fgSSJoin / CPU	CPU	CPU / CPU-GPU
$[10^4 - 10^6]$	N/A	fgSSJoin	fgSSJoin / CPU-GPU	CPU-GPU	CPU-GPU
$[10^7 - 10^8]$	fgSSJoin / CPU	fgSSJoin	fgSSJoin / bitmap	CPU-GPU	CPU-GPU
10^9	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	fgSSJoin / CPU-GPU
$> 10^9$	sf-gSSJoin	sf-gSSJoin	sf-gSSJoin	N/A	N/A

Figure 22: Summary of the best techniques per scenario examined