Information Systems 89 (2020) 101485

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

An empirical evaluation of exact set similarity join techniques using GPUs

Christos Bellas *, Anastasios Gounaris

Department of Informatics, Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 17 April 2019
Received in revised form 6 December 2019
Accepted 10 December 2019
Available online 13 December 2019
Recommended by F. Naumann

Keywords:
Set-similarity join
GPU computing
CUDA

ABSTRACT

Exact set similarity join is a notoriously expensive operation, for which several solutions have been proposed. Recently, there have been studies that present a comparative analysis using MapReduce or a non-parallel setting. Our contribution is that we complement these works through conducting a thorough evaluation of the state-of-the-art GPU-enabled techniques. These techniques are highly diverse in their key features and our experiments manage to reveal the key strengths of each one. As we explain, 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. All our work is repeatable and extensible.

1. Introduction

Given two collections of sets and a threshold, set similarity join is the operation of computing all pairs, the overlap of which exceeds the given threshold. Similarity joins are used in a range of applications, such as plagiarism detection, web crawling, clustering and data mining and have been the subject of extensive research recently, e.g., [1–5].

In very large datasets, finding similar sets is not trivial. Due to the inherent quadratic complexity, a set similarity join between even medium sized datasets can take hours to complete on a single machine. For example, in the same setting used in our experiments to be presented later, a similarity join over the complete DBLP dataset using a Jaccard threshold of 0.85 takes approximately 8.5 h when employing only a modern CPU. In addition, challenges like high dimensionality, sparsity, unknown data distribution and expensive evaluation arise. To tackle scalability challenges, two main and complementary approaches have been followed. Firstly, to devise sophisticated techniques, which safely prune pairs that cannot meet the threshold as early as possible, typically through simple computations related to the prefix and the suffix of the ordered sets, e.g. [1,4]. Secondly, to benefit from massive parallelism offered by parallel paradigms such as MapReduce [5–8] and GPGPU (General-Purpose computing on Graphics Processing Units) [9,10]. Orthogonally, there exist several proposals that trade accuracy for faster times, such as techniques for approximate set similarity or for nearest neighbor search, e.g., [11–13] (the detailed discussion of related work is deferred to Section 5). Our work deals with exact set similarities joins exclusively.

Comparative evaluations on the state-of-the-art techniques for set similarity joins that focus on either a MapReduce or a non-parallel single machine setting have recently appeared [1,14]. The goal of our work is to fill the gap and thoroughly evaluate the state-of-the-art exact set similarity algorithms and techniques in a single machine setup that can benefit from massive parallelism through the usage of a graphics card. Hence, we distinguish our work from [14], where distributed set similarity join algorithms are evaluated. On the other hand, we evaluate techniques, which may use the massive parallelism contrary to the rationale in [1].

Modern GPUs offer a high-parallel environment at low cost. As a result, GPGPU has been introduced to accelerate a large variety of applications [15]. In general, GPGPU takes advantage of the different and complementary characteristics offered by CPUs and GPUs to improve performance. It has been employed in domains like deep learning, bioinformatics, numerical analytics and many others. However, implementing existing algorithms and techniques on a GPU requires in-depth knowledge of the hardware specifications and is often counter-intuitive. In addition, not all tasks are suitable for GPU-side processing. A traditional CPU surpasses at complex branching in application logic, while a GPU is superior at mass parallel execution of simple tasks and floating point operations [16].

In this work, we perform a comprehensive experimental evaluation between GPU-accelerated set similarity joins and CPU standalone implementations using the framework provided in [1]. Moreover, we examine two alternatives (i) transferring the whole workload onto the GPU, or (ii) splitting the workload between

* Corresponding author.
E-mail addresses: chribell@csd.auth.gr (C. Bellas), gounaria@csd.auth.gr (A. Gounaris).

https://doi.org/10.1016/j.is.2019.101485
0306-4379/

the CPU and the GPU and assigning the most suitable tasks to each part. Our findings demonstrate that there does not exist a clear winner among the evaluated techniques; in other words, each alternative has its own sweet spot depending on the data and query characteristics. In summary, the contributions of our work are as follows:

- To the best of our knowledge, this is the first comprehensive presentation and comparative evaluation of GPU accelerated set similarity joins. In our study, we include the GPU-oriented state-of-the-art techniques.
- We conduct extensive performance analysis using eight real world datasets. We compare our findings against the state-of-art CPU and GPGPU implementations. We identify the conditions under which each solution becomes the dominant one so that to derive a set of guidelines as to when to use each technique.
- We provide a repository with all techniques, so that third-part researchers can repeat and extend our work.[1]

Paper outline. Next, we give an overview of set similarity joins and provide the basic details about the CUDA programming model. We present the-state-of-the-art techniques in Section 3. Our experimental analysis is presented and discussed in Section 4. We provide an overview of the related work in Section 5. We conclude our study and discuss open issues in Section 6.

2. Background

We introduce the filter-verification framework used by state-of-the-art main memory set similarity join algorithms in line with the comparison work conducted by Mann et al. [1]. We also provide a comprehensive overview of the CUDA programming model, which is proprietary to NVIDIA [17] yet widespread in practice, and explain its main concepts.

2.1. Set similarity joins

The state-of-the-art main memory set similarity algorithms conform to a common filter-verification framework, as explained in [1]. The common idea behind all these algorithms is (i) to avoid comparing all possible set pairs by applying filtering techniques on preprocessed data to prune as much candidate pairs as possible; and (ii) then to proceed to the actual verification of the remaining candidates. We summarize the key points of the work of [1] that are relevant to our research below.

2.1.1. 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. As a result of such a preprocessing phase, all the tokens of a set are unique. The input data tokens are represented by integers and are sorted by their frequency in increasing order, so that infrequent tokens appear first in a set. The sets of a collection are sorted first by their size and then lexicographically within each block of sets of equal size.

2.1.2. Set similarity functions

To measure the similarity between sets, the normalized similarity functions Jaccard, Dice and Cosine are typically used. The given normalized threshold τ_n is translated to an equivalent overlap τ, which defines the minimum number of tokens that need to be shared between two sets to satisfy the threshold (see Table 1). For example, if the Jaccard similarity threshold of two 10-token sets is set to 0.8, this is translated to an overlap threshold of 9 tokens that need to be shared.

[1] Source code is publicly available from https://github.com/chribell/gpgpu_ssjoin_eval.

Table 1
Similarity functions.
Source: Adapted from [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

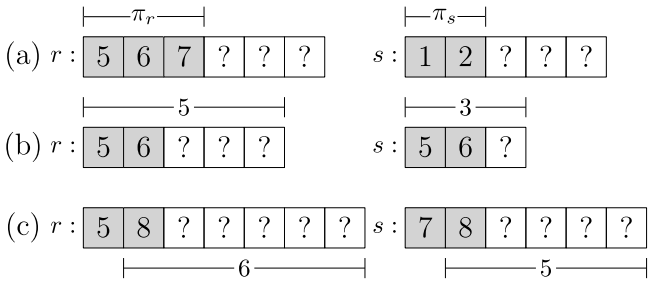


Fig. 1. Filters used for candidate pruning: (a) prefix, (b) length, (c) positional.

2.1.3. Filters

The most widely used filter, called *prefix-filter*, exploits the given threshold and similarity function by examining only two subsets called *prefixes*, one from each sorted set in the candidate pair, and discards the pair if there is no overlap between the prefixes. A π-prefix is formed by the π first tokens of the set, i.e. for set r, $\pi_r = |r| - \tau + 1$ and for set s, $\pi_s = |s| - \tau + 1$. In Fig. 1(a), for $\tau = 4$, there is no overlap between the respective set prefixes, thus, even if there is an overlap on the remaining tokens, any overlap threshold set to 4 or higher cannot be reached, and in such cases, the candidate pair (r, s) can be safely pruned.

Another filter, known as *length filter*, takes advantage of the normalized similarity functions dependency on set size. Hence, a candidate pair can be pruned if the set size inequality $\tau_n \cdot |r| \leq |s| \leq |r|/\tau_n$ is not satisfied. In Fig. 1(b), if $\tau_n = 0.8$, the shown candidate pair (r, s) can be pruned despite the prefix equality because a 6-token r set requires a s set of size $4 \leq |s| \leq 6$.

The last filter used in the examined algorithms is the *positional filter*. Given the first match position, it evaluates if a candidate pair can reach the similarity threshold. As an example, in Fig. 1(c), if the threshold implies that at least 6 tokens should be shared, the pair is pruned since the remaining tokens from set s are not enough to reach the similarity threshold.

2.1.4. Algorithm outline

The set similarity join operation is achieved by executing an index nested loop join consisting of three steps (see Algorithm 1 and the corresponding main notation in Table 2). First, through an index lookup, a preliminary candidate set (pre-candidates) is generated (line 5). In the second step, pre-candidates are deduplicated and filtered (line 6). The pairs that pass all filters form the final candidates. These two steps compose the *filtering* phase. In the third and final step, also noted as *verification* in the literature, the similarity score for each of the remaining candidate pair is computed and if it exceeds the threshold, the pair is added to the output (line 13).

A special but very common case is where the set similarity join is a self-join using only a single collection of sets. In that case, a token set is first probed against the current index contents and then added to the index itself. This allows for incremental index

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
```

Table 2
Notation.

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

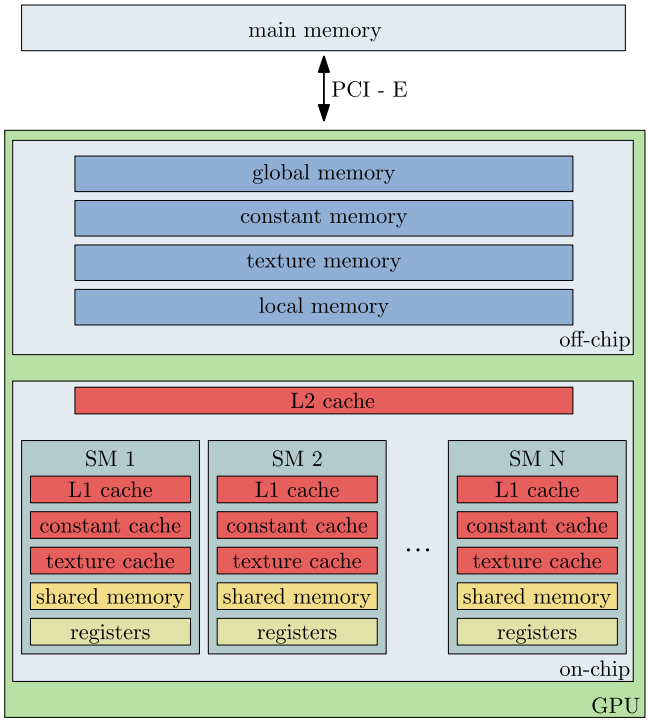


Fig. 2. Memory hierarchy for CUDA-enabled GPUs.
Source: Taken from [18].

building that is interleaved with verification. Also, the fact that a set that probes the index is always no shorter than the current indexed sets, since the sets are also ordered by their size, leads to more candidate pairs to be pruned earlier.

2.2. CUDA overview

In CUDA terminology, the CPU and the main memory are referred to as *host*, while the GPU and its own memory are referred to as *device*. In this work, we use the terms CPU and host (resp. GPU and device) interchangeably.

2.2.1. Architecture

CUDA-enabled GPUs have many cores called *Streaming Processors (SPs)*, which are divided into groups called *Streaming Multiprocessors (SMs)*. Each SM includes also other units such as ALUs, instruction units, memory caches for load/store operations, and follows the *Single Instruction Multiple Data (SIMD)* parallel processing paradigm.

2.2.2. Thread organization

Threads are organized in logical blocks called *thread blocks*. A thread block is scheduled and executed in its entirety on a SM in groups of 32 threads called *warps*. Threads within a warp are called *lanes* and share the same instruction counter, thus they are executed simultaneously in a SIMD manner. An example of how warps map to a thread block to be executed is shown in Fig. 3; thread block sizes are defined as kernel launch parameters. There are two cases of thread divergence, which degrade performance, namely *inter-warp*, when concurrent warps run unevenly and *intra-warp*, when warp lanes take different execution paths. The latter is also simply referred to as *warp divergence*.

2.2.3. Memory hierarchy

There are several memory types on CUDA-enabled GPUs. They are divided into on-chip and off-chip ones. Off-chip memories include the *global*, *constant*, *texture* and *local* memory. The global memory is the largest (in the order of GBs) but slowest memory. Data transferred from the host to device resides in global memory and it is visible to all threads. The constant memory is read-only and much smaller (in the order of KBs). It is used for short access times on immutable data throughout the execution. The texture memory is essentially a read-only global memory and is preferred when 2-dimensional spatial locality occurs in memory access patterns. The local memory is part of the global memory and is used when the registers needed for a thread are fully occupied or cannot hold the required data. This is called *register spilling*. On-chip memories include the *caches*, the *shared* memory and the *registers*. For data reuse, there are caches per SM and the L2 cache is shared across all SMs. The shared memory is the second fastest memory type. Each SM has its own shared memory. Data stored in shared memory can be accessed by all threads within the same block, thus threads of the same block are allowed to inter-communicate via shared-memory. The registers are the fastest memory type and contain the instructions of a single thread and the local variables during the lifetime of that thread. An overview of the memory hierarchy is illustrated in Fig. 2.

2.2.4. Kernel grid

Every function to be processed in parallel by the GPU is called a *kernel*. Each kernel is executed by multiple thread blocks, which form the kernel *grid*. The grid can be regarded as an array of blocks with up to two dimensions. Each block is, in turn, an array of threads with up to three dimensions. CUDA can schedule blocks to run concurrently on a SM depending on the shared memory and registers used per block. Increasing either of these factors can lead to limited concurrent block execution, which results

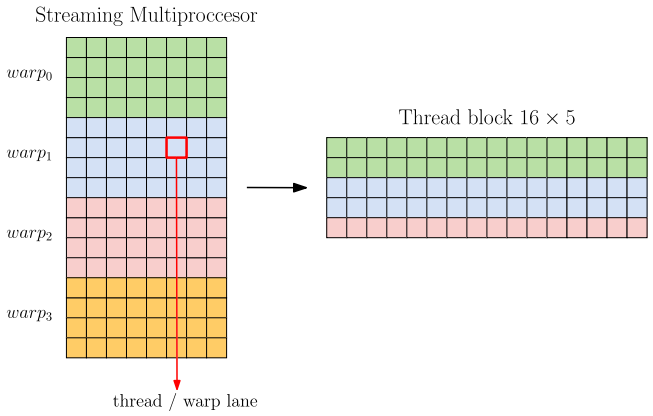


Fig. 3. A warp-to-thread block mapping example.

in low occupancy. *Occupancy* is defined as the ratio of active warps on a SM to the maximum allowed active warps per SM. Maximizing occupancy is a good heuristic approach but it does not always guarantee performance gain. On the contrary, maintaining high warp *execution efficiency*, i.e. the average percentage of active threads in each executed warp, is a more robust approach for data-management tasks, as shown in executing generic theta-joins on GPUs [18].

3. Algorithms & techniques

In this section, we review and discuss the algorithms and techniques that are evaluated in this work. First, we summarize the findings of Mann [1] regarding the best three CPU algorithms, which we consider as our *baseline*. Second, we present three techniques, which employ the GPU to accelerate set similarity joins in a different manner.

3.1. CPU

In [1], seven main memory algorithms[2] are compared using real world datasets. The key characteristics of the best three algorithms are summarized below.

AllPairs (ALL). It is the first and most naive main memory algorithm to exploit the given threshold. During the inverted list lookup, it applies the prefix and length filters to prune candidate pairs [2].

PPJoin (PPJ). It extends ALL by applying the positional filter on pre-candidates [19]; therefore its verification phase is less loaded at the expense of a higher overhead during filtering.

GroupJoin (GRP). It is an extension to PPJ. Sets with identical prefix are *grouped* together. Each group is handled as a single set. Thus GRP has faster indexing, as it discards candidate pairs in batches. During the verification phase, the candidate pairs are expanded [3].

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.

[2] AllPairs [2], PPJoin and PPJoin+ [19], MPJoin [20], MPJoin-PEL [21], AdaptJoin [22] and GroupJoin [3].

3.2. GPGPU

The techniques that use the GPU fall into two categories: (i) those that split the workload between the CPU and the GPU, and (ii) those that move the whole workload to the GPU.

3.2.1. CPU–GPU co-processing

Since the algorithms solving the set similarity problem efficiently conform to the filter-verification framework, splitting the workload between CPU and GPU involves specifying which component each phase is assigned to. In principle, GPUs are used to accelerate compute-intensive applications by processing data in a predefined memory space. In addition, based on the way that the CPU invokes the GPU, it is considered an anti-pattern if a GPU operates autonomously, e.g., self-exiting based on some condition, which may hurt the overall performance.

In [10], a co-process scheme between CPU and GPU is introduced in order to efficiently compute set similarity join. 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. Thus, the CPU remains responsible for index building and initial pruning of candidate pairs. On the other hand, the verification phase is stated as more suitable for parallelization, as it involves a merge-like loop, where the overlap of candidate pairs is computed and therefore is delegated to the GPU. As a result, the multithreaded framework depicted in Fig. 4 is proposed. The main challenges addressed involve the appropriate data layout on the device memory, efficient usage of the fast shared memory and thread workload; regarding the latter, three main alternatives have been explored: a thread can evaluate multiple candidate pairs, or only one pair or part of a single candidate pair.

ALL, PPJ and GRP, reported as the most efficient algorithms in [1], are used for the filtering phase. Since the limited GPU memory is the most dominant constraining factor, the workload is divided into chunks and the GPU is invoked several times. Hence, the CPU iteratively transfers as many candidates as the GPU memory can handle. This results in a non-blocking filtering phase, which naturally lends itself to an execution overlap between filtering and verification as shown in Fig. 5; the third phase is for the construction of the final output pairs on the CPU. The two CPU phases run in different threads. As reported in [10], the runtime is improved in several cases, especially for large datasets.

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.

3.2.2. GPU standalone

Ribeiro et al. with their successive works in [9,23,24] perform the complete exact set similarity join on the GPU. We provide an overview of the three main techniques proposed below along with a brief discussion.

GPU-based Set Similarity Join (gSSJoin) [23]. *gSSJoin* first constructs a static inverted index over all tokens; then, for each probe set, it performs a set similarity operation consisting of three steps: (i) First, an intersection count between the input set and every other set is calculated. By using the inverted index,

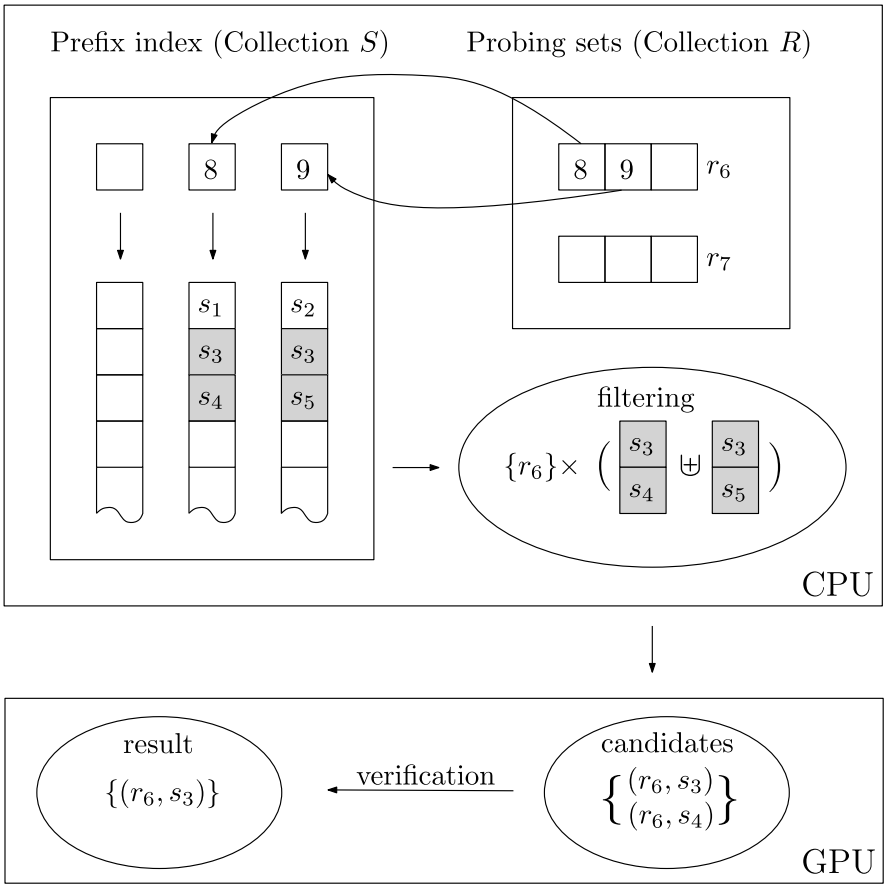


Fig. 4. Splitting the workload between CPU (candidate generation) and GPU (candidate verification). *Source:* The workload representation is adapted from [1].

the workload is evenly distributed among the GPU cores. Fig. 6 depicts how *gSSJoin* evenly distributes the workload among four GPU threads. In the example in the figure, first, the four inverted lists corresponding to the tokens of probe set r_{20} with ids 4,8,9, and 13 are concatenated in a logical vector E. Next, given the number of GPU threads, $|T| = 4$, each thread is assigned with $|E|/|T| = 16/4 = 4$ elements. Each element contributes to the computation of the intersection count of a specific candidate pair containing r_{20}. Each thread processes independently its corresponding partition; this implies that a count may be incremented by multiple threads and thus the atomic add operation is used to ensure correctness. (ii) Then, the Jaccard similarity of every pair is calculated and stored in global memory. (iii) Finally, the pairs that have similarity higher or equal to the threshold value are selected and transferred to the main memory. To reduce the transfer size, a stream compaction technique is employed.

Discussion. *gSSJoin* can be considered as an efficient brute force approach, since, for every pair, the corresponding similarity score is calculated. By completely avoiding any kind of filtering, it remains more robust to variations of threshold values and token frequency distribution; this is its strongest point. By not using any kind of filtering, proven to be quite effective especially in large threshold values, *gSSJoin* conducts numerous unnecessary intersection counts and may add extra overhead to the overall runtime in contrast to filter-based approaches. However, the main drawback of *gSSJoin* is the inherent launch overhead, which starts dominating in medium and large datasets (>1M): as the dataset size increases, the launch overhead increases as well, since each probe set corresponds to several independent kernel calls.[3]

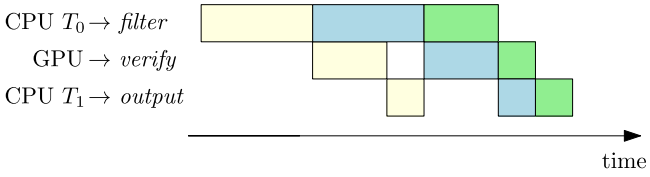


Fig. 5. Execution overlap between the CPU threads ($T_0 \rightarrow$ filter, $T_1 \rightarrow$ output) and GPU (verify), each color corresponds to a different data chunk of candidates.

Filter-based gSSJoin (fgSSJoin) [24]. *fgSSJoin* extends *gSSJoin* by encapsulating a filtering phase. In addition, a block partitioning scheme is adopted in order to process collections of arbitrary size. In summary, the input collection is divided into blocks of size n, with n being an input parameter chosen accordingly for the result to fit in global memory. Through an iterative process, a block is indexed and probed against itself and all its predecessors. Fig. 7 illustrates the block partitioning and probing. First, block R_0 is indexed and probed against itself. After that, block R_1 is indexed and probed against itself and its predecessor R_0. Finally, the same procedure applies to block R_2. The sets are ordered by their size;

[3] In our setting, a launch overhead is approximately 0.01 ms, which is several times higher than the execution overhead of a single kernel. Given the fact that for each probe set, gSSJoin launches many kernels to find its similar pairs, as the dataset size increases, the aggregated overhead becomes the main performance bottleneck.

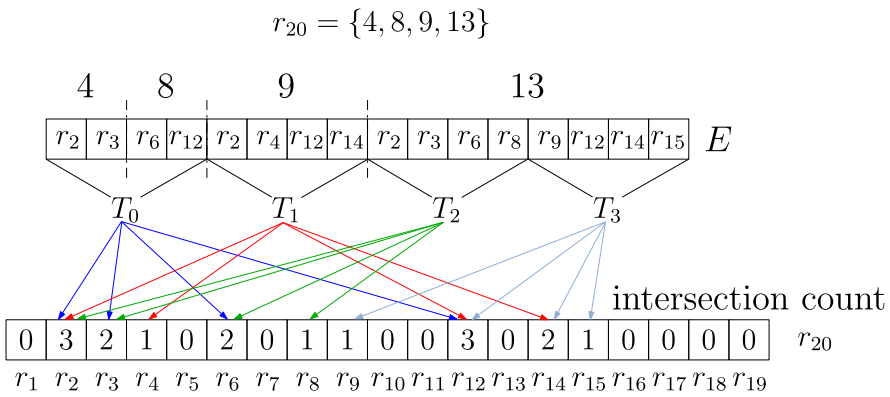


Fig. 6. gSSJoin workload allocation example for four GPU threads.

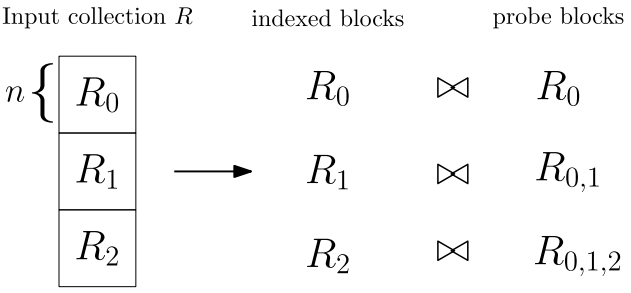


Fig. 7. Block partitioning scheme used in fgSSJoin and sf-gSSJoin.

this allow a length filter to be applied and whole block-to-block comparisons to be skipped.

During the filtering phase, partial intersection counts between indexed and probed sets prefixes are calculated and stored in global memory, given that on-chip memories are too small to hold the results. This incurs an $O(n^2)$ space complexity to cover the worst case scenario, where all set pairs are candidates. Subsequently, every pair that has a non-zero partial intersection count undergoes full verification, whereas the rest is pruned. The workload is distributed similarly to *gSSJoin*. Fig. 8 depicts the filter-verification steps of *fgSSJoin*. Initially, by using the prefix filter, *fgSSJoin* computes the partial intersection counts between the probe sets r_0, r_1, r_2 and the indexed set r_5. Before moving to the verification phase, the pair (r_5, r_1) can be safely pruned since its partial intersection count is zero. Finally, the non-zero partial intersection count pairs undergo full verification. As a result, only the pair (r_5, r_2) has the required overlap and thus it is added in the output.

Discussion. By integrating a filtering stage, *fgSSJoin* reduces the candidate search space, especially in large threshold values. Its most significant contribution is the block partitioning scheme, which overcomes the launch overhead limitation of *gSSJoin*. In addition, thanks to a length filter, whole blocks can be pruned. A scalability issue 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.

Size-filtered gSSJoin (sf-gSSJoin) [9]. *sf-gSSJoin* adopts the same block partitioning scheme as *fgSSJoin*. It performs the set similarity join into two phases per probe. First, it calculates the complete intersection counts between probed and indexed sets without any prefix filtering. Then, for each non-zero intersection count, it calculates the corresponding pair's Jaccard similarity. Finally, the output stored in linear global memory is transferred back to main memory. Fig. 9 gives an overview of *sf-gSSJoin*'s probing. In contrast with *fgSSJoin*, *sf-gSSJoin* directly calculates the

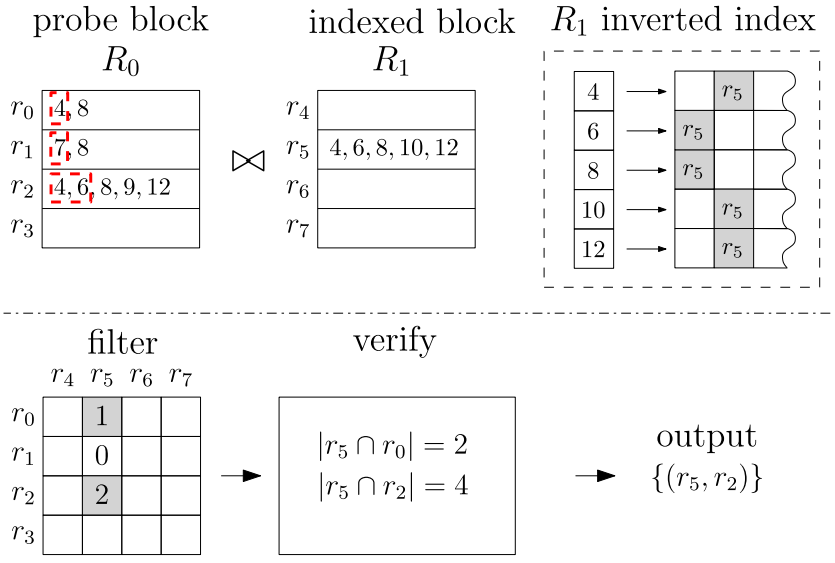


Fig. 8. fgSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

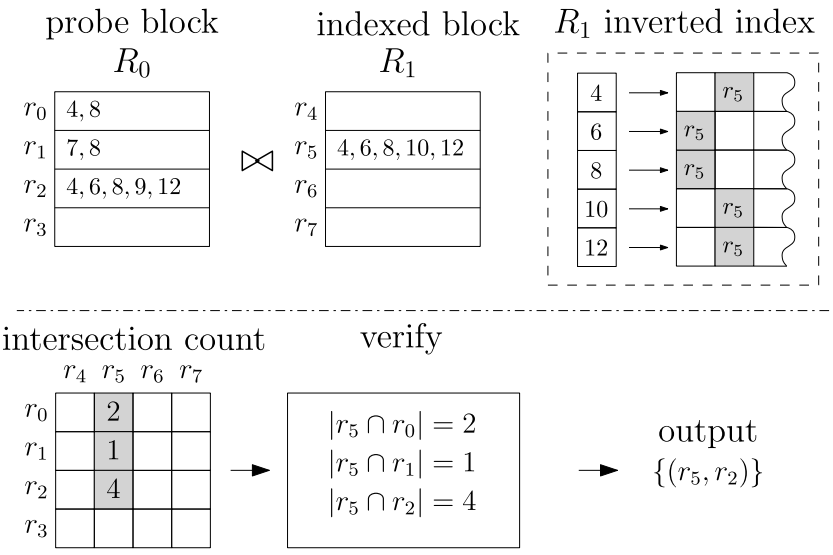


Fig. 9. sf-gSSJoin probe block example (block size $n = 4$, threshold $\tau_n = 0.8$).

complete intersection counts between probe sets r_0, r_1, r_2 and the indexed set r_5 in its first pass. This results to more global memory accesses, compared to *fgSSJoin* when prefix filtering is effective.

Discussion. As stated before, as the threshold value decreases, filter effectiveness degrades leading to an increased number of candidates and subsequently larger execution times. *sf-gssJoin* combines the robustness of *gSSJoin* against varying threshold values and data token frequencies with the scalability of the block partitioning scheme used in *fgSSJoin*; thus, as reported in [9], it outperforms its two predecessors when the filtering does not make much difference, e.g., for threshold values lower than 0.5. Contrary to *fgSSJoin*, no prefix filter is employed; only the length one is used.

3.2.3. Bitmap filter

In their work, Mann et al. [1] 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* filter.

Essentially, the bitmap filter uses hash functions to create signature bitmaps of size b for the input collection sets. Thus, the initial dataset tokens are mapped in a fixed bitmap space. Without compromising the exactness of set similarity join, bitmap filter can deduce an overlap upper bound for a candidate pair Eq. (3.1). If the upper bound is less than the minimum required overlap, the candidate can be safely pruned. Fig. 10 illustrates the application of bitmap filter on an example candidate pair.

$$|r \cap s| \leq \lfloor \frac{|r| + |s| - popcount(b_r \oplus b_s)}{2} \rfloor \tag{3.1}$$

Discussion. The simplicity and speed of bitmap filter relies on the fast bitwise operations. In order to deduce an overlap upper bound, a population count operation of the signatures' hamming distance is required, which in turn can be directly converted to the corresponding hardware instruction. Such bitwise operations can easily benefit from the massive parallel environment of the GPU. To support their reasoning, the authors of [25] describe a simple GPU implementation, in which the GPU generates candidate pairs using the bitmap filter and the CPU verifies them. As a result, significant speedup is achieved over the sequential algorithms depending on the dataset characteristics. However, their solution lacks scalability, since it involves transferring complete candidate pairs via the PCI-E bus, which becomes the main bottleneck. We investigate the application of bitmap filter utilizing the *fgSSJoin* block partitioning scheme; contrary to the initial proposal in [25], to mitigate scalability issues, we keep the whole join process on the GPU.

Table 3
Feature comparison between the evaluated techniques.

	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

3.2.4. Summary view

In Table 3, we provide a unified table, where all the GPU-enabled alternatives are shown with regards to their differences from the standard CPU algorithms. During verification, we distinguish between the techniques where multiple thread workload alternatives are employed and those that simply resort to atomic add operations in a straight-forward manner.

As already explained, our techniques are main-memory ones. Therefore, the dataset must fit into CPU main memory, while the GPU's main memory must be large enough to hold the inverted index in the GPU standalone techniques.

4. Evaluation

The goals of our experimental evaluation are threefold: (i) to show the extent of the achieved speedups between CPU standalone and GPU-accelerated implementations while improving on the CPU should not be taken for granted; (ii) to identify the conditions under which each solution becomes the dominant in practice, and (iii) to provide explanations about the observed behavior.

For ease of presentation, we split the experiments into two parts: the main part, which is adequate to reveal the main pros and cons of each technique, and the additional part, where we emphasize on scalability and on using synthetic data to cover a larger range of data characteristics.

The remarks of both sets of experiments are either the same or complementary and are summarized at the end.

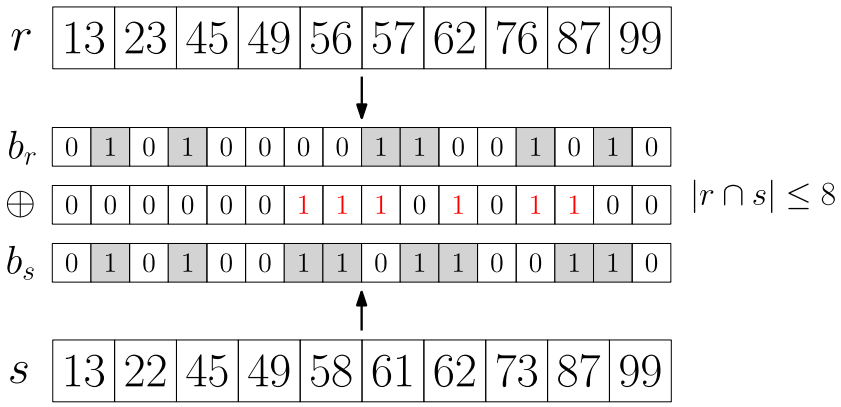


Fig. 10. Candidate pair r, s can be safely pruned for $\tau_n = 0.95 \rightarrow \tau = 10$ since its expected overlap upper bound is 8.

4.1. Setting of the main experiments

The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3 GHz, 32 GB RAM at 2400 MHz and an NVIDIA Titan XP on CUDA 9.1. This GPU has 3840 CUDA cores, 12 GB of global memory and a 384-bit memory bus width.

We report the overall runtime, noted as *join* time, which is the composition of filtering (if any) and the verification conducted. When we drill-down, we explicitly refer to the former as *filtering* time and to the latter as *verification* time. We do not include any data preprocessing time spent for tokenization and de-duplication, which are performed exactly as in [1]. Note that data transfer times are also included in the *join* time for the GPU-enabled solutions. We conduct experiments for all datasets presented below using by default five thresholds in the range [0.5, 0.9]. We focus on self-joins using the Jaccard similarity and experiment on main-memory solutions on a single machine. Thus, for the CPU-based solutions, datasets must fit in main-memory. For the GPU-enabled approaches, the complete datasets are also copied to the GPU memory, possibly in chunks. 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. The reported time for each experiment is an average over 5 independent runs but is important to note that no significant deviation was observed. We measure the total *join* time with the *std::chrono* and *time.h* libraries. For the GPU operations we measure time by using the CUDA event API.

We experiment with eight real world datasets; seven of them were also employed in [1] and we also employed the TWITTER dataset found in [26]. Table 4 shows an overview of each dataset characteristics. Some datasets follow a Zipf-like distribution of set sizes, as shown in Fig. 11, but in general, the distribution types differ. A summary of each dataset (adapted from [1]) is as follows:

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.

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.

Table 4
Datasets characteristics.

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	$3.7 \cdot 10^4$

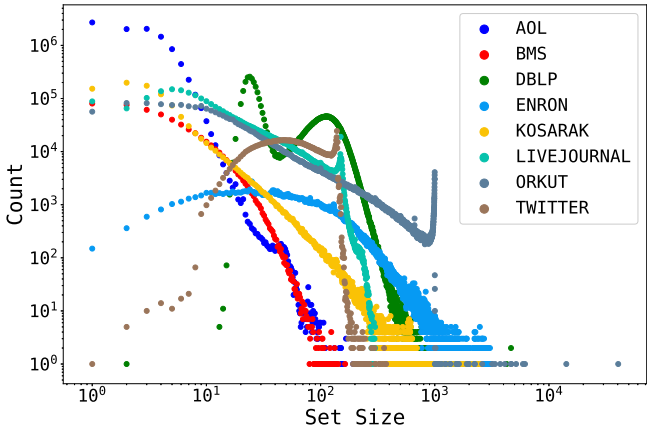


Fig. 11. Datasets set size distribution.

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.

In the remainder of the discussion, datasets of 10^5 will be referred to as small, the ones in the order of 10^6 medium, and those in 10^7 as large. As can be observed from Table 4, the datasets differ significantly in the average set size and number of distinct tokens. When the average set size is less than 10, it will be referred to as small; otherwise, as large. Also, the number of different tokens will be characterized as small when in the order of 10^4, and large when in the order of 10^6. These

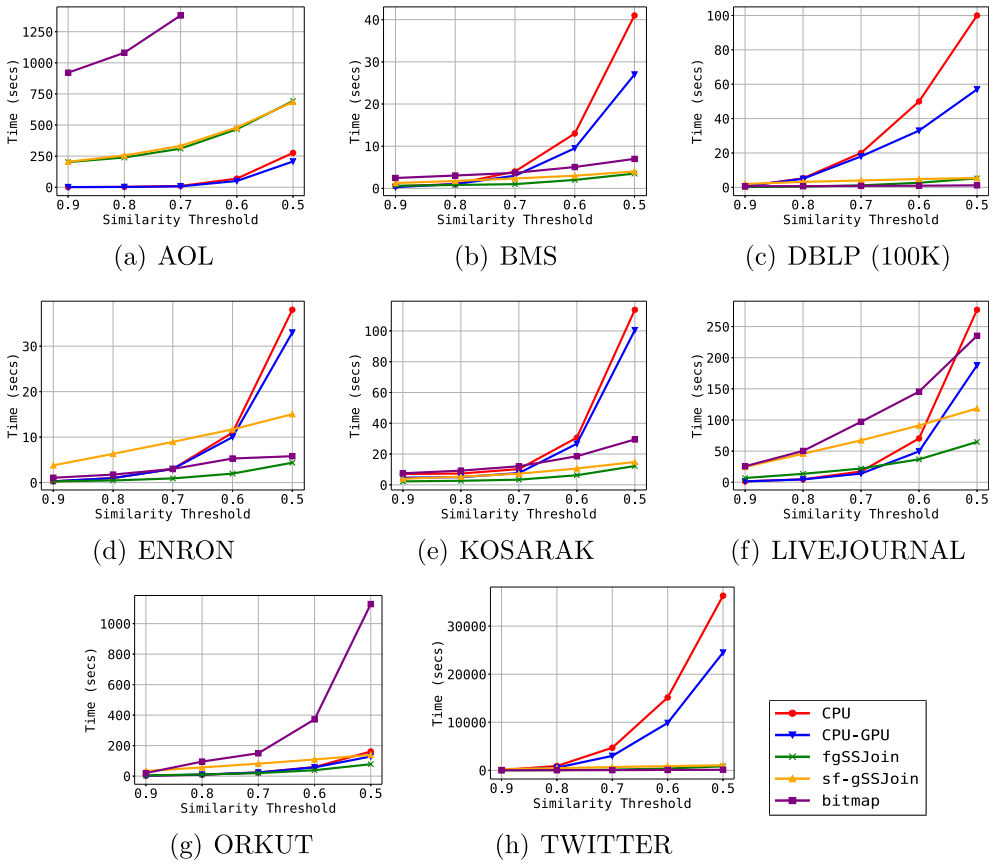


Fig. 12. Comparison between the best times for different thresholds.

properties are essential to derive the sweet spots of each technique, because, along with the threshold values, they affect the amount of candidate pairs that need to be verified. For example, for the same dataset size and threshold values, fewer pairs are filtered when the number of distinct tokens is small. Similarly, any filtering becomes less effective when the average set size is small, especially when combined with low number of distinct tokens.

4.1.1. Launch configuration

In GPGPU computing, selecting the best launch parameters, i.e. number of blocks and threads per block is required to fully exploit the massive parallelism. Although recent compiler optimizations tend to offload the developer from the burden to meticulously pick these parameters, depending on the context and the GPU code, it may be inevitable for these not to be *hand picked*.

For the *CPU–GPU* co-processing framework [10], with fine tuning[4] and by selecting the appropriate workload allocation between threads, depending on each dataset characteristics, we manage to fully overlap filtering and verification phases, and the final phase, where the CPU constructs the result pairs. For *gSSJoin* and its variations, we used a fixed number of thread blocks (equal to the number of SMs) and threads per block (max supported value) as proposed by their authors. Specifically, we launch a 1-dimensional grid consisting of 15 thread blocks and 1024 threads per thread block. In addition, we choose to partition the input collection to blocks of size $n = 15\,000$.

[4] Fine-tuning for this approach is only required for the small datasets where, depending on certain key dataset characteristics, the GPU verification may add an extra overhead. As the dataset size increases, the overall join time is bounded by the CPU filtering time. Hence, fine-tuning the GPU in such cases would not yield any performance speedup.

4.2. Main experiments

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 Fig. 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.

As shown in Fig. 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, especially 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, but as explained later, there are several other factors that impact on performance.

Nevertheless, there is no clear pattern. Continuing 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. The detailed observations with the key strengths and weaknesses of each technique are deferred to Section 4.4.3 and after we explain the technique behavior in more detail. Also, we have omitted the *gSSJoin* solution, due to its inability to scale, as explained below.

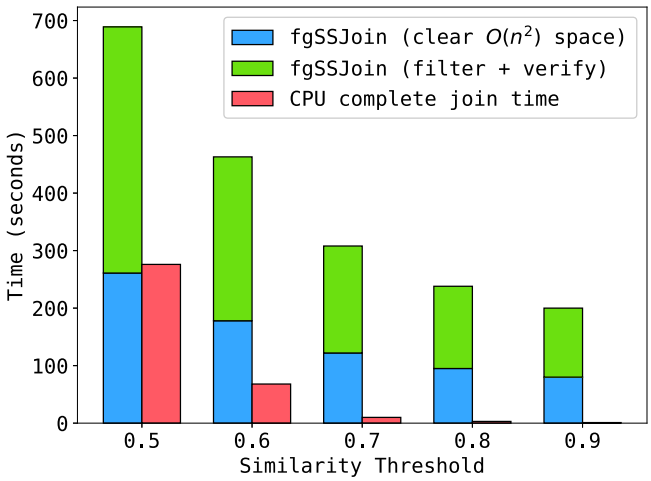


Fig. 13. Quadratic space overhead.

4.3. Performance analysis

Motivated by the fact that neither technique is the most dominant one (as also shown in Fig. 12), first we shed light on the impact of the fact that the GPU standalone solutions require quadratic space in the global memory. We also discuss the effect of dropping the prefix filter in *sf-gSSJoin*. Further, we evaluate the use of bitmap filter, instead of prefix filter used in *fgSSJoin*, and provide useful insights. In addition, we explain why we omitted *gSSJoin* from the main experiments. Another aspect of our analysis, involves the evaluation of the scalability of all solutions as we run them over artificially increased datasets. Finally, we experiment on synthetic datasets to further highlight the impact of specific dataset characteristics. The two latter aspects, due to their significance, are examined separately.

4.3.1. Quadratic space overhead of fgSSJoin and sf-gSSJoin

For datasets, the output of which does not fit in the GPU memory, *fgSSJoin* and *sf-gSSJoin* use a block division scheme to process input data gradually. To conduct a set similarity join between two blocks, an $O(n^2)$ memory space is required to store the intersection counts. After each block probe, this memory space must be cleared to ensure correctness for the next probe calls.

Considering that an intersection count is a 4-byte integer, the required memory space to store all counts for $n = 15\,000$ is 900 MB. In our experiments, we use the `cudaMemset` function 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 which in turn increases the clear operation overhead of the quadratic memory space. Concrete numbers are presented in Fig. 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.

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 the CPU solution for thresholds above 0.6, and, overall *fgSSJoin* and *sf-gSSJoin* are not the optimal GPU-enabled techniques.

Table 5
Quadratic space overhead for AOL.

τ_n	# probes	Accumulated $O(n^2)$ space	Time (s)
0.5	138 841	125 TB	261
0.6	94 747	85 TB	178
0.7	64 928	58 TB	122
0.8	50 431	45 TB	95
0.9	42 628	38 TB	80

4.3.2. Filtering vs complete intersection (fgSSJoin vs sf-gSSJoin)

Assume that someone would like to run a similarity query that is rather close to a cartesian product by setting the threshold to low values, such as $t < 0.5$. In this case, adopting and relying on effective filtering is not beneficial. This renders *sf-gSSJoin*, which is closer to brute-force, the best performing technique. In general, *sf-gSSJoin* exhibits robustness with regards to low threshold values. While it may perform worse than *fgSSJoin*, it scales better when we gradually lower the threshold. Fig. 14 shows concrete examples for the DBLP(1M), ENRON and ORKUT datasets, and complements the relevant results from Fig. 17.

4.3.3. Bitmap performance

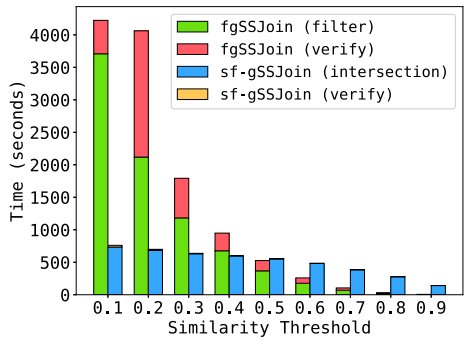
Bitmap filtering can be seen as an alternative to a prefix-based approach. However, the technique is highly dependent on the dataset characteristics, since sets may produce similar bitmaps even if they do not share similar tokens; in general, the probability of such undesired collisions is increased whenever the number of set tokens is increased and the bitmap size is reduced. Trying to increase the bitmap size along with the number of set tokens does not scale either because it leads to more global memory accesses. Nonetheless, the *bitmap*-based solution has its own sweet spot: high average set size combined with low number of different tokens, as in the TWITTER and DBLP datasets. Fig. 12 shows the best timings of the bitmap solution for our main experiments (we have experimented with several signature sizes).

For the AOL dataset, bitmap has the worst performance because it inherits the quadratic space overhead of *fgSSJoin* and its filtering is ineffective. For the BMS, ENRON and KOSARAK datasets and high threshold values, bitmap performs worse than every other technique. However, as the threshold value decreases, the performance difference between the best performing techniques and bitmap is smaller. The best bitmap performance is observed for the DBLP(100k) and TWITTTER datasets. Furthermore, in Fig. 17 to be discussed later in detail, we show that bitmap remains efficient for larger portions of the DBLP dataset. On the other hand, for the LIVEJOURNAL and ORKUT datasets, bitmap has the worst performance for the majority of threshold values.

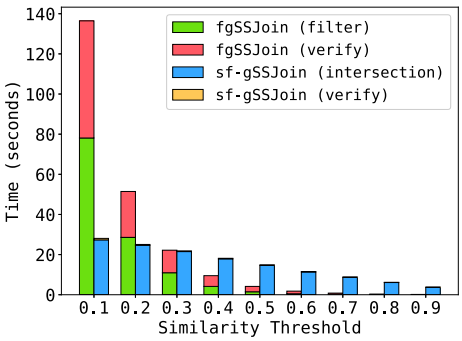
Finally, as shown in Fig. 15, we show the impact of the average set size and the number of different tokens on the efficiency of bitmap filter. The BMS, DBLP (1M) and the TWITTER datasets have small number of different tokens. However, BMS, contrary to DBLP and TWITTER has small average set size, which leads to a big increase in the filtering time. For ORKUT, even though it has a large average size, due to its large number of different tokens, it renders bitmap filtering ineffective (the same behavior was also observed for the LIVEJOURNAL dataset).

4.3.4. GSSJoin launch overhead

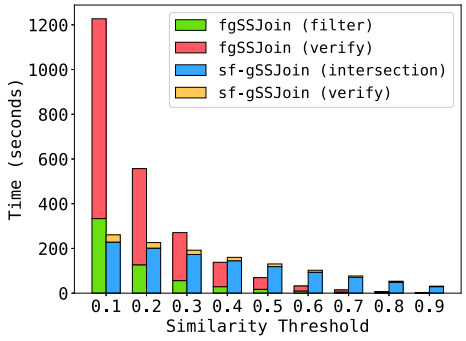
As stated in the technique presentation, *gSSJoin* launches a sequence of separate GPU kernel calls per probe set in order to conduct the set similarity join operation. Given the fact that launching a large number of small kernels is considered a bad practice, executing *gSSJoin* results into an accumulated launch and execution overhead as shown in Fig. 16. Hence, the overall



(a) DBLP (1M)

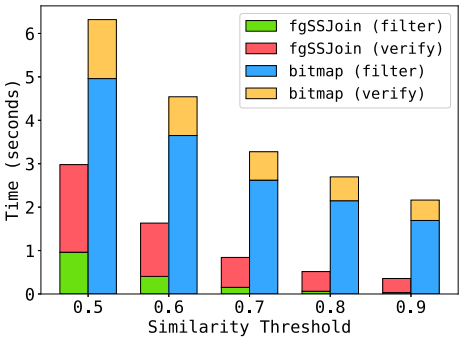


(b) ENRON

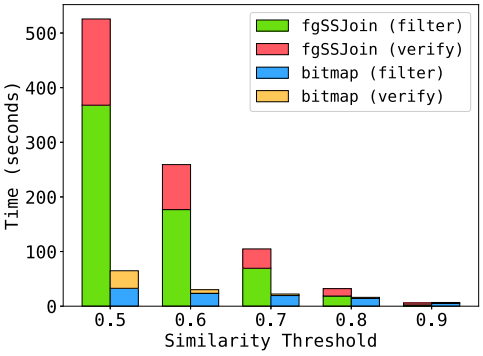


(c) ORKUT

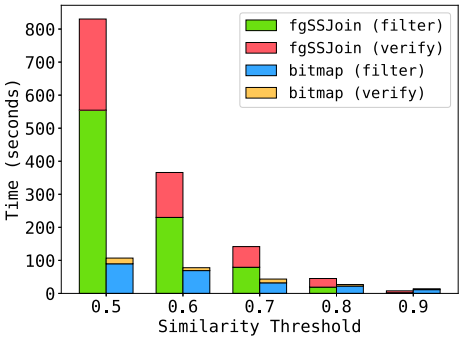
Fig. 14. Comparison between the best times of *fgSSJoin* vs *sf-gSSJoin* for different thresholds.



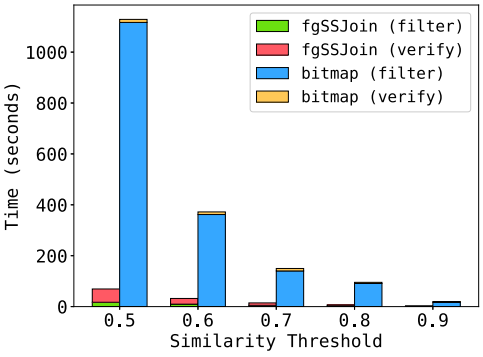
(a) BMS



(b) DBLP (1M)



(c) TWITTER



(d) ORKUT

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

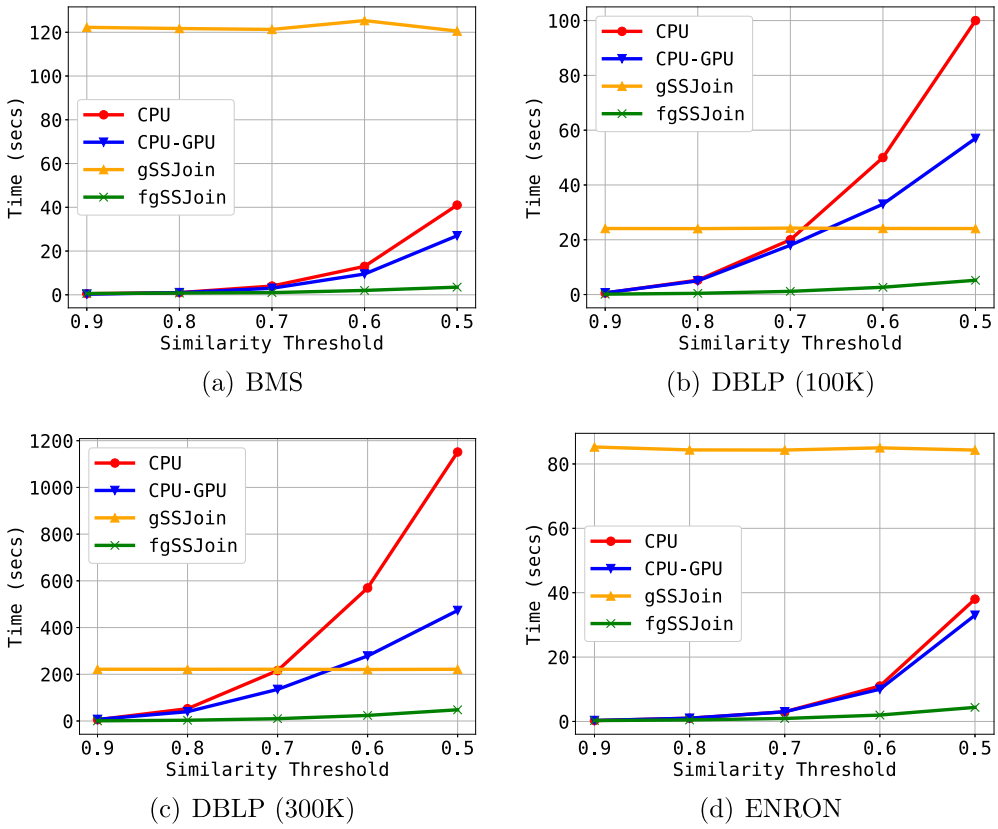


Fig. 16. gSSJoin runtimes for different thresholds compared to other techniques.

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

Dataset	Cardinality	Avg set size	# diff tokens
BMS (×25)	$1.3 \cdot 10^7$	6.5	1681
ENRON (×25)	$6.1 \cdot 10^6$	135	$1.1 \cdot 10^6$
LIVEJOURNAL (×5)	$1.5 \cdot 10^7$	36.5	$7.5 \cdot 10^6$

runtimes are dominated by the overhead required to invoke the GPU and it is threshold value independent. In some cases, such as the DBLP datasets (Figs. 16(b) and 16(c)), *gSSJoin* performs better than the CPU at lower thresholds since the GPU invocation overhead is lower than the filtering phase conducted on the CPU. However, as the input collection size increases the GPU overhead increases dramatically, rendering *gSSJoin* inefficient for real world applications and is totally dominated by *sf-gSSJoin*.

4.4. Additional experiments

Before presenting the main outcomes, we run additional experiments using larger and synthetic datasets.

4.4.1. Scalability

We examine the scalability of the evaluated solutions (i) on larger portions of the DBLP dataset, which follow the same token distribution frequency and, (ii) on artificially increased versions of the BMS, ENRON and LIVEJOURNAL datasets. We populate the increased datasets similarly to [7]. All the corresponding dataset characteristics are located in Table 6.

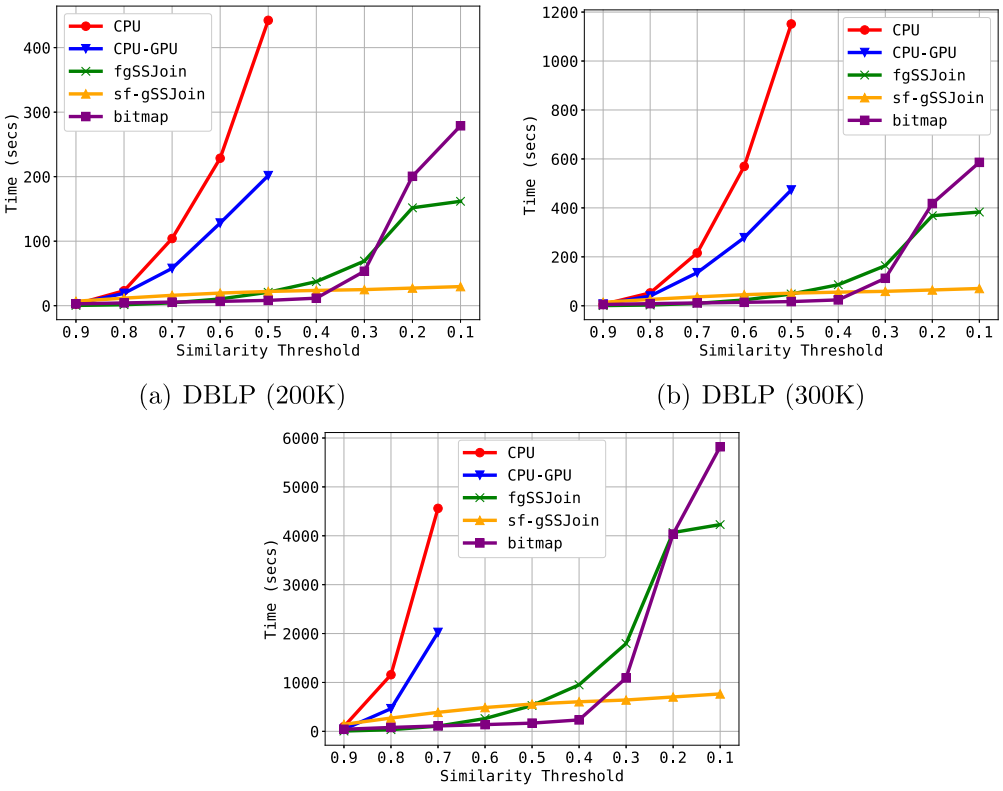
From Fig. 17, we can see that the speed-ups of the GPU standalone solutions are much more evident as the collection size increases compared to the CPU standalone for larger portions of the DBLP dataset. This is further verified from partial results regarding the full DBLP dataset, as shown in Table 7. The benefits from employing GPUs are tangible even for larger thresholds, given that, for the complete dataset, the candidate pairs are tens of billions and CPU takes up to two hours approximately for DBLP dataset with $\tau_n = 0.9$.

However, the correct interpretation needs to take into account the behavior for the AOL dataset, where the GPU standalone solutions are suboptimal. The key observation is that in Fig. 17,[5] the GPU standalone solutions perform very well not due to their capability to scale for large datasets, but due to the length filter they encapsulate. This type of filter is particularly effective for the DBLP dataset but not for the AOL. The important lesson learned is that dataset characteristics should be always taken into account in all their main dimensions (dataset size, average set size and number of different tokens).

Consequently, we proceed by experimenting on artificially increased datasets using the best performing techniques. We also narrow the threshold range to $\tau_n \in [0.7 - 0.9]$ due to the large overall join times. Depending on the index size, it may not be feasible to increase datasets arbitrarily. For example, we could populate a x10 version of the ORKUT dataset following the method used in [7]. However, this leads to a 13GB index size, which cannot fit in the GPU memory. We selectively choose two small datasets, i.e. BMS and ENRON, one medium dataset, i.e. LIVEJOURNAL, and increase each by an appropriate factor in order to comply with the main-memory restriction.

As it can be seen in Fig. 18(a), increasing the BMS dataset does not yield any performance difference between the techniques compared to the original dataset. This is mainly due to the combination of the small number of different tokens and the

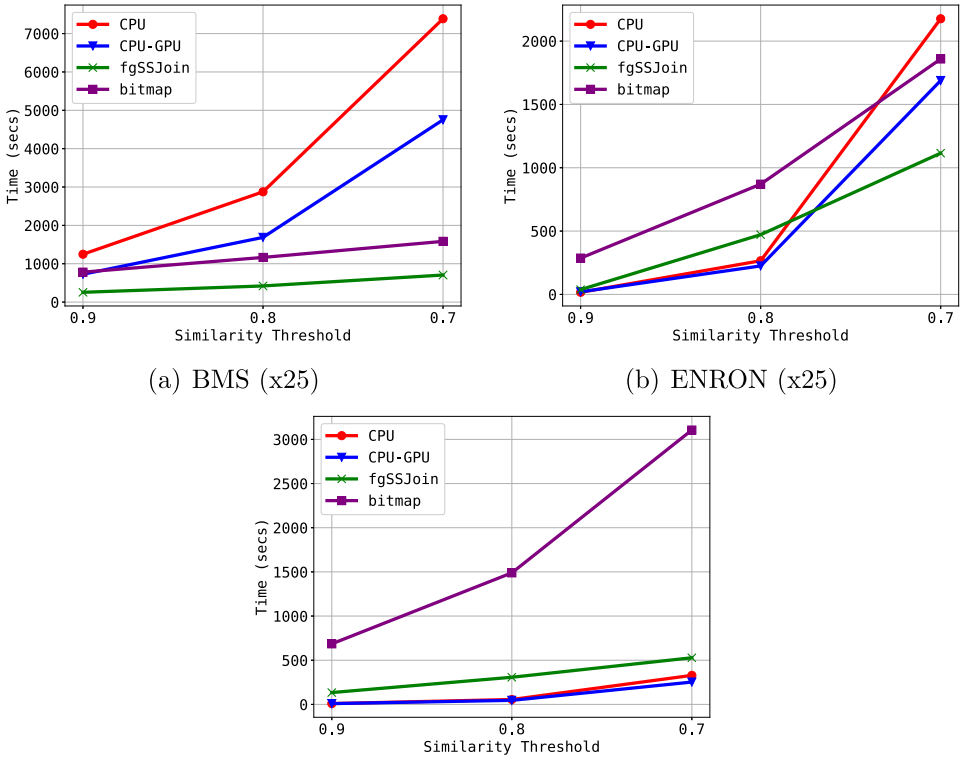
[5] When $\tau_n = 0$, the similarity join is essentially a cartesian product, where the only practical solution is *sf-gSSJoin* with running time as for $\tau_n = 0.1$.



(a) DBLP (200K) (b) DBLP (300K)

(c) DBLP (1M)

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



(a) BMS (x25) (b) ENRON (x25)

(c) LIVEJOURNAL (x5)

Fig. 18. Comparison between the best times for the increased datasets.

Table 7
Runtimes for the complete DBLP consisting of 6.1M sets (in s).

τ_n	CPU	CPU–GPU	fgSSJoin	Bitmap	sf-gSSJoin
0.8	–	32 387	1259	607	10 353
0.9	7244	2814	230	303	5227

Table 8
Synthetic datasets' characteristics.

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

small average set size, which results in a high number of frequent tokens. As a result, this benefits *fgSSJoin* to conduct filtering and intersection count faster.

In contrast, the *CPU* and *CPU–GPU* techniques perform better than *fgSSJoin* on the increased ENRON dataset for $\tau_n = 0.8$ as shown in Fig. 18(b). Because of the large number of different tokens coupled with the large average set size, there is a high number of infrequent tokens, which favors prefix filtering. While the prefix filter remains effective, CPU-based filtering techniques also remain competitive. For the same reasons, the performance difference between the CPU-based filtering techniques and *fgSSJoin* becomes more evident in the increased LIVEJOURNAL dataset (Fig. 18(c)). However, as the threshold value is decreased for the ENRON dataset, prefix filtering becomes ineffective, leading *fgSSJoin* to perform better (Fig. 18(b), $\tau_n = 0.7$).

4.4.2. Evaluation on synthetic datasets

We highlight three key dataset characteristics: (i) dataset size, (ii) average set size, and (iii) the number of different tokens. In order to evaluate the impact of each of these factors in a more controlled manner, we create twelve synthetic datasets using the combination of characteristics listed in Table 8. Furthermore, the synthetic datasets follow a zipf-like token frequency distribution. We use the original scripts provided by the authors of [1]. To distinguish each dataset, we use the notation *Dataset Size - Number of different tokens - Average set size*.

Initially, we keep the dataset size fixed to 10M and vary the values of the other two characteristics as shown in Fig. 19. In the same manner, we keep the number of different tokens fixed to 500k (Fig. 20) and the average set size fixed to 25 (Fig. 21). In all of the synthetic datasets, we observe two behavioral patterns. As such, we can classify the runtimes into two categories, (i) those where the CPU-based filtering outperforms *fgSSJoin* for every examined threshold value and, (ii) those in which *fgSSJoin* begins to perform better for $\tau_n = 0.7$.

We see that CPU-based solutions are constantly more efficient than *fgSSJoin* apart from the cases where the number of distinct tokens is not large and the average set size is large. More specifically, in cases where the number of different tokens is 500k, there is a high number of infrequent tokens, which benefits CPU-based filtering. For lower number of distinct tokens, prefix filtering gradually becomes ineffective, and thus *fgSSJoin* performs better, as previously stated in Section 4.4.1.

In conclusion, the combination of the three main dataset characteristics, namely dataset size, number of distinct tokens and average set size, directly impacts on the prefix filter efficiency, and consequently on which technique is the best-performing one. A small number of different tokens alongside a large average set size will result also in more frequent tokens, which does not allow prefix filtering to prune a lot of candidate pairs. On the contrary, for larger datasets with a high number of different tokens, it is more probable to have infrequent tokens.

4.4.3. Summary of remarks

We summarize the strong and weak points of each technique. In the discussion, threshold values of 0.9 are considered as very high, 0.8 as high, 0.5–0.7 as medium and less than 0.5 as low.

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.
Dominating 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.

CPU–GPU:
Strong points: handling large datasets (but cannot scale with decreasing threshold values).
Dominating cases: medium–high thresholds ($0.5 < t < 0.8$) and large datasets.
Weak points: handling (i) small 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.

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.
Dominating cases: Same as the cases in the strong points.
Weak points: handling large datasets due to the quadratic complexity in the block size and the associated overhead.

sf-gSSJoin:
Strong points: handling not very big datasets combined with low thresholds.
Dominating 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.

bitmap:
Strong points: handling cases where bitmap signatures are effective, i.e., high set size and not high number of different tokens.
Dominating 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.

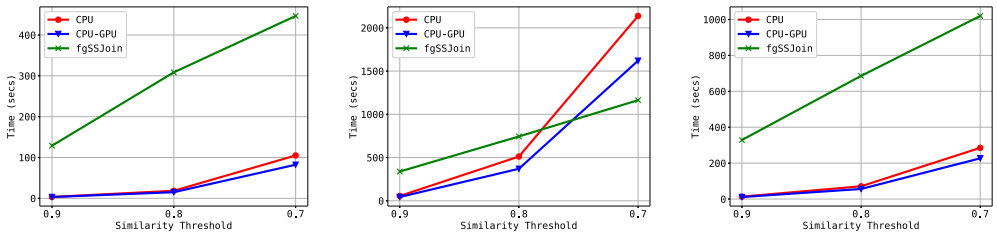
In Fig. 22, we provide an overview of the cases, where each technique is the dominant one, based on our experiments. When there are two techniques for the same combination of dataset size and average set size, the distinct token cardinality makes the difference. The top part of the figure splits the rows according to the τ_n values. The bottom part repeats the table with rows split according to the result size. As can be seen, the contents are highly correlated, thus the discussion above based on threshold values can be rephrased to be based on result sizes.

4.5. Output consideration

In all the experiments above, the set similarity results contained only the count for the GPU-enabled solutions. However, when considering outputting the complete results, there are negligible changes.

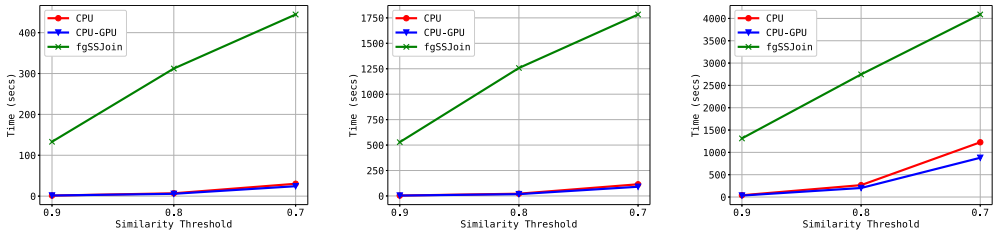
More specifically, *CPU–GPU* manages to completely hide the final output construction due to the thread overlapping shown in Fig. 5.

The rest of the techniques share the same approach to constructing the final result pairs in the GPU and then counting them. However, the main bottleneck is to transfer the results back to the CPU. In Fig. 23, we show the overhead for the DBLP dataset. For



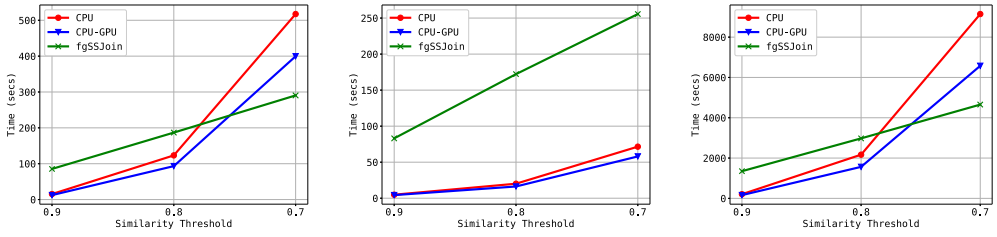
(a) 10M - 50K - 5 (b) 10M - 50K - 25 (c) 10M - 500K - 25

Fig. 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

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



(a) 5M - 50K - 25 (b) 5M - 500K - 25 (c) 20M - 50K - 25

Fig. 21. Comparison between the best times for synthetic datasets with fixed average set size.

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

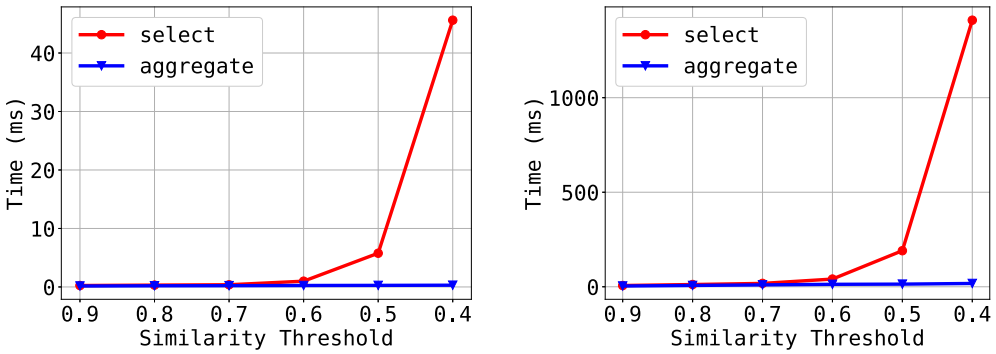
Fig. 22. Summary of the best techniques per scenario examined.

instance, in a challenging case where the dataset is 1M, $\tau_n = 0.4$ and the number of output pairs is $1.3 \cdot 10^9$, *fgSSJoin* transfers back to the main memory a tuple consisting of the pair and its similarity degree for each result item. Thus, each tuple has a size of 12 bytes, which amounts to roughly 15.6 GB for the complete output. This can be transferred in less than 1.5 s through the slow PCI-E CPU–GPU connection, which is negligible compared to the join times reported in Fig. 17.

5. Additional related work

Although extensive research has been carried out on set similarity join for parallel paradigms, such as MapReduce [5,7,8], there are few additional studies investigating set similarity join on the GPGPU paradigm, which, however, focus on approximate solutions while we deal with exact ones exclusively.

An early proposal has appeared in [11], according to which Lieberman et al. cast the similarity join operation as a GPU sort-and-search problem. First, they create a set of space-filling curves using bitonic sort on one of the input relations; then, they process each record from the other relation in parallel by executing searches in the space filling curves, using the Minkowski metric for similarity. In [12], the authors employ the parallel-friendly MinHash algorithm to estimate the Jaccard similarity of two sets. Their solution is space-efficient since they only store set signatures instead of whole sets to perform the similarity join.



(a) DBLP (100K) (b) DBLP (1M)

Fig. 23. Transfer times via PCI-E for the select (pairs) and aggregate (counts) queries for the *fgSSJoin*,*sf-gSSJoin* and *bitmap* techniques.

However, due to the MinHash nature (i.e. data partitioning in *bins*), fine-tuning is required to achieve balance between accuracy (to avoid false positives) and execution time. The main limitations of the above techniques is that, apart from being approximate, they are inherently limited to Jaccard similarity only.

Similarity joins are also discussed in [27], where two nested loop join (NLJ) algorithms are presented: a naive NLJ and a faster index-supported NLJ. The index is created on the CPU side during the preprocessing phase. Both algorithms use the Euclidean distance for similarity and thus they are not suitable for set similarity joins. Nevertheless, the solutions described in [10] also perform sophisticated CPU-side indexing before the GPU-side processing.

Another problem, which is close to set similarity join and has been studied on the GPGPU paradigm, is similarity (nearest neighbor) search. Examples include [13,28–31] but none of them can be applied to our problem. More specifically, the proposal in [28,29] uses a GPU-based parallel Locality Sensitive Hashing (LSH) algorithm to perform approximate k-nearest neighbor (kNN) search. In [30], a hybrid CPU–GPU framework, which uses LSH combined with other techniques, such as reservoir sampling is presented, where, the CPU constructs hashtables and the GPU process them and performs a count-based top-k selection. In [31], the authors propose a two-level tree and a re-ranking method for fast approximate nearest neighbor search. In [13], Johnson et al. present a framework to compute a k-NN graph. All these proposals can be deemed as devising key elements for building additional approximate set similarity join techniques.

6. Final discussion

This work summarized and thoroughly evaluated the existing state-of-the-art GPU-based exact set similarity joins. We observe that the main techniques have been proposed in the last few years and have different characteristics, which supports the hypothesis that GPU-enabled similarity joins is still a technology in evolution. More importantly, there is no clear winner, which leaves the question as to whether a globally dominant solution exists open. In Section 4.4.3, we summarize the key strengths and sweet spots of each technique. Below, we summarize additional generic observations, lessons learned and technical challenges encountered:

1. This work aims to extend comprehensive evaluations, such as these in [1], where a single physical machine is employed, through allowing computations to be also performed on a single GPU. We assume that the initial datasets are up to several GBs in size, so that they can fit into the RAM. In such a setting, judiciously employing the GPGPU paradigm can lead to speed-ups up to two orders of magnitude, e.g., 339X for the Twitter dataset when $\tau = 0.5$ (see Fig. 12). By judiciously, we mean deciding the technique to employ according to the summary table in Fig. 22, which may suggest not to use GPU at all. However, this is the exception; despite working with datasets fitting into the main memory of a modern machine, GPU outperforms CPU-only solutions, especially when $\tau_n < 0.9$.
2. A main contribution of this work is to reveal and describe the strengths and weaknesses of each technique. A prerequisite is to identify the main factors that impact on the relative performance. The dataset size, token frequency distributions, number of different tokens, average set sizes and τ_n parameter (related to the result size) have been identified as such factors, with the dataset size, average set size and τ_n parameter being the most important ones. At a high-level, these parameters directly impact the effectiveness of the filter types employed by the different techniques.
3. From a technical perspective, the most challenging issues are the design choices, which concern the kernel grid and balanced splitting of the workload across GPU threads. Filtering and verification using the intersection count technique, share common data, and as such, they are tightly coupled. Therefore, they cannot be regarded as individual operations and must be implemented taking into account their interactions. Based on how data is accessed and the workload of each thread, certain GPU features, such as shared memory, that otherwise could speed up the join process, may not be possible to exploit to their full extent. For example, *fgSSJoin* splits the workload among threads based on the static inverted index, which leaves global memory atomic operations as the only alternative to process filtering and intersection count. As a result, the fast on-chip shared memory is not utilized.
4. Elaborating on the technical issues, brute-force-like solutions, such as *gSSJoin* cannot scale because of the high launch overhead, which is associated with the input dataset size. More sophisticated solutions, such as *fgSSJoin* and *bitmap* adopt a block partitioning scheme that enables them to process thousands of sets per GPU invocation. However, their current limitations, which are generic in the GPGPU programming, concern (i) memory consistency and (ii) kernel invocation.

    - Regarding memory consistency, for *fgSSJoin* to ensure correctness, prefix filtering is conducted via atomic operations on global memory. This may lead to a performance degradation in cases where multiple threads

must update a specific global memory address. For *bitmap*, as the signature size increases, the global memory access footprint also increases and eventually dominates the filtering phase.

- Regarding kernel invocation, for the GPU-standalone techniques, a GPU invocation consists mainly of two kernels, a filtering and a verification kernel. Essentially, the filtering kernel produces candidate pairs for the verification kernel to consume and verify in order to output the final result. Due to the GPGPU programming nature, a kernel cannot consume any portion of data produced by a prior kernel until the latter runs to completion. This has an impact on resource utilization and consequently on performance since a thread block that has finished the filtering phase waits for every block of the grid to finish. Recent advancements on GPGPU programming such as multi-block cooperative groups which enable collective operations, such as verification, to work across all threads in a group, may contribute to alleviate this issue, but this remains an open issue.

As mentioned above, the long-term goal is the development of a potentially hybrid and adaptive technique that is dominant across all cases. To this end, several remaining issues need to be deeper investigated. We identify the following main directions for direct extensions to our work with arbitrary order of significance:

- We consider our work as a key step towards understanding in depth the strengths and weaknesses of each technique. Nevertheless, there is always room for even more thorough comparative analysis. In our experiments, the largest dataset was up in the order of 10^7 sets given that we focused on techniques employing a single GPU. Even many years ago, such datasets can be deemed as medium, rather than large ones [32]. Scaling in an efficient manner to billions of records is a challenging open issue, given also that the initial results on MapReduce-based techniques have not shown good scalability [14]. Perhaps using multiple GPUs is a promising approach that remains to be investigated.
- Following up on the point above, the full space of significant parameters has not been explored. We have identified that the average set size and the number of distinct tokens play a significant role, since they are directly related to the amount of work. However, a wider range of combinations of dataset size, token frequency distributions, number of different tokens and average set sizes needs to be explored.
- In addition to exploring the impact of multiple GPUs, the efficient usage of heterogeneous devices is an open issue, which entails more systematic work on workload splitting and task assignment. Some early work in this regard has already been performed in [9].
- To tackle scalability problems in large datasets, approximate techniques need to be thoroughly evaluated as well. In Section 5, we have listed several of them, but no comparative evaluation study has been conducted yet.
- There is still room for additional techniques. For example, no GPGPU techniques to accelerate filtering in parallel through the usage of an inverted index exist to date. Such techniques may be coupled efficiently with the work in [10], provided that atomic operations on the global memory are avoided and shared memory is exploited to the largest possible extent. Similarly, exact set similarity joins may be able to benefit from recent advances in GPU indices, such as [29].

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.is.2019.101485.

Acknowledgment

The authors gratefully acknowledge the support of NVIDIA, United States Corporation through the donation of the GPU used through the GPU Grant Program.

References

[1] W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, Proc. VLDB Endow. 9 (9) (2016) 636–647, URL http://www.vldb.org/pvldb/vol9/p636-mann.pdf.
[2] R.J. Bayardo, Y. Ma, R. Srikant, Scaling up all pairs similarity search, in: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007, 2007, pp. 131–140.
[3] P. Bouros, S. Ge, N. Mamoulis, Spatio-textual similarity joins, PVLDB 6 (1) (2012) 1–12.
[4] Y. Jiang, G. Li, J. Feng, W. Li, String similarity joins: An experimental evaluation, PVLDB 7 (8) (2014) 625–636.
[5] A.D. Sarma, Y. He, S. Chaudhuri, Clusterjoin: A similarity joins framework using map-reduce, PVLDB 7 (12) (2014) 1059–1070.
[6] R. Baraglia, G.D.F. Morales, C. Lucchese, Document similarity self-join with mapreduce, in: ICDM, 2010, pp. 731–736.
[7] R. Vernica, M.J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: SIGMOD Conference, 2010, pp. 495–506.
[8] A. Metwally, C. Faloutsos, V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors, PVLDB 5 (8) (2012) 704–715.
[9] S. Ribeiro-Junior, R.D. Quirino, L.A. Ribeiro, W.S. Martins, Fast parallel set similarity joins on many-core architectures, J. Inf. Data Manag. 8 (3) (2017) 255.
[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, in: DaMoN'19, ACM, New York, NY, USA, 2019, 5:1–5:10.
[11] M.D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, 2008, pp. 1111–1120.
[12] M.S. Cruz, Y. Kozawa, T. Amagasa, H. Kitagawa, Gpu acceleration of set similarity joins, in: International Conference on Database and Expert Systems Applications, Springer, 2015, pp. 384–398.
[13] J. Johnson, M. Douze, H. Jégou, Billion-scale similarity search with gpus, 2017,
[14] F. Fier, N. Augsten, P. Bouros, U. Leser, J.-C. Freytag, Set similarity joins on mapreduce: an experimental survey, Proc. VLDB Endow. 11 (10) (2018) 1110–1122.
[15] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, D. Glasco, Gpus and the future of parallel computing, IEEE Micro 31 (5) (2011) 7–17.
[16] S. Mittal, J.S. Vetter, A survey of cpu-gpu heterogeneous computing techniques, ACM Comput. Surv. 47 (4) (2015) 69.
[17] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors - A Hands-on Approach, second ed., Morgan Kaufmann, 2013.
[18] C. Bellas, A. Gounaris, GPU processing of theta-joins, Concurr. Comput.: Pract. Exper. 29 (18) (2017).
[19] C. Xiao, W. Wang, X. Lin, J.X. Yu, G. Wang, Efficient similarity joins for near-duplicate detection, ACM Trans. Database Syst. 36 (3) (2011) 15:1–15:41.
[20] L.A. Ribeiro, T. Härder, Prefix filtering to improve set similarity joins, Inf. Syst. 36 (1) (2011) 62–78.
[21] W. Mann, N. Augsten, PEL: Position-enhanced length filter for set similarity joins, in: Proceedings of the 26th GI-Workshop Grundlagen Von Datenbanken, 2014, pp. 89–94.
[22] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering?: an adaptive framework for similarity join and search, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2012, pp. 85–96.
[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.
[25] E.F. Sandes, G. Teodoro, A.C. Melo, Bitmap filter: Speeding up exact set similarity joins with bitwise operations, 2017, arXiv preprint arXiv:1711.07295.
[26] A. Go, R. Bhayani, L. Huang, Twitter sentiment classification using distant supervision, CS224N Project Report, Stanford, 1(12), 2009.
[27] C. Böhm, R. Noll, C. Plant, A. Zherdin, Index-supported similarity join on graphics processors, in: BTW, Vol. 144, 2009, pp. 57–66.
[28] J. Pan, D. Manocha, Fast gpu-based locality sensitive hashing for k-nearest neighbor computation, in: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2011, pp. 211–220.
[29] J. Zhou, Q. Guo, H.V. Jagadish, L. Krcál, S. Liu, W. Luan, A.K.H. Tung, Y. Yang, Y. Zheng, A generic inverted index framework for similarity search on the gpu, in: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, 2018, pp. 893–904.
[30] Y. Wang, A. Shrivastava, J. Ryu, Flash: Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search, 2017, arXiv preprint arXiv:1709.01190.
[31] P. Wieschollek, O. Wang, A. Sorkine-Hornung, H.P.A. Lensch, Efficient large-scale approximate nearest neighbor search on the gpu, in: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[32] G. Graefe, New algorithms for join and grouping operations, Comput. Sci. - Res. Dev. 27 (2012) 3–27.