HySet: A Hybrid Framework for Exact Set Similarity Join Using a GPU

Christos Bellas[a], Anastasios Gounaris[a]

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

Abstract

Set similarity join is a fundamental operation used in a wide range of applications such as data mining, data cleaning and entity resolution. Existing methods proposed for set similarity join conform to a filter-verification framework where potential candidate pairs are generated in the filtering phase and then undergo a verification phase to output the final result. Several different kinds of filtering techniques have been proposed and techniques also differentiate in the manner they couple filtering with verification. However, it has been shown that no globally dominant technique exists. Depending on the dataset and query characteristics, each technique has its own strong and weak points. Based on these findings, the main contribution of this work is the development of a hybrid framework for the set similarity join operation for a single GPU-equipped machine setting. Our framework encapsulates a partitioning mechanism to utilize appropriately both the CPU and the GPU. We present all technical details and we show performance speedups up to 3.25x after thorough evaluation.

Keywords: set-similarity join, GPU computing, CUDA

1. Introduction

Exact set similarity join is the operation of finding all similar pairs between two collections of sets. Two sets are considered similar only if their similarity degree is equal to, or exceeds a user defined threshold. Because of its generality, set similarity join is used in a wide range of application domains including data mining [1], data cleaning [2] and entity resolution [3].

Existing solutions proposed for the set similarity join problem conform to a filter-verification framework. First, in the filtering phase, potential candidate pairs are generated in order to be fully evaluated next, in the verification phase. The pairs that successfully pass the verification phase are considered similar. Different filtering techniques have been developed to accelerate the filtering phase [2, 4, 5, 6, 7, 8]. However, it is now well established from the work of Mann et al. [9], that most of the computational cost is spent on the filtering phase (for relatively high thresholds) and also that sophisticated filtering hurts overall performance.

The majority of the proposed techniques in literature are implemented in a non-parallel setting. Nonetheless, there have been previous studies for set similarity join in a parallel setting, namely, using MapReduce and General-Purpose GPU (GPGPU) paradigms. For the MapReduce paradigm, Fier et al. [10] show in an experimental survey that distributed solutions cannot scale efficiently and are sensitive to certain data characteristics such as set size, set element frequency, and also to query characteristics such as threshold value.

On the other hand, there are more promising results for the GPGPU paradigm as shown in [11]. Even so, employing the GPU may not be beneficial in some occasions. More specifically, according to the experimental evidence in [11], no globally dominant technique exists, and as also reported in [10], it is extremely challenging to achieve high speedups for the set similarity problem. Depending on the dataset and query characteristics, each technique has its own strong and weak points.

This work makes the following three contributions:

1. Motivated by the fact that in a single GPU-endowed machine setting, no globally dominant technique exists, we develop a hybrid framework for the exact set similarity join operation which utilizes concurrently both the CPU and the GPU. Thus, we achieve an execution overlap between both ends.
2. We propose a two-level partitioning scheme in order to process the join operation conveniently on both the CPU and the GPU. In addition, we present two main approaches to handle the workload splitting, (i) by using a concurrent queue, or (ii) by dichotomizing the input appropriately.
3. We conduct extensive experiments on real world and synthetic datasets and provide a thorough experimental analysis. We show that our hybrid framework can achieve speedup of up to 3.25x over the best single-threaded CPU-standalone and GPU-enabled techniques. We also show the benefits over multi-threaded CPU solutions, where speedups are higher.

The rest of the paper is organized as follows: In Section 2, we provide some preliminaries on the set similarity

join problem alongside with the state-of-the-art solutions. We present our framework in Section 3. We evaluate our framework in Section 4 and discuss our findings in Section 5. We give an overview of the related work in Section 6. Finally, we conclude our study and discuss possible future work in Section 7.

2. Background

In this section, we give a formal definition for the set similarity join problem and introduce the state-of-the-art solutions to tackle the problem. In addition, we give an overview of how incorporating a GPU may lead to significant speedup.

2.1. Problem Definition

Given collections R, S and a normalized threshold value $\tau_n \in [0, 1]$, set similarity join is the operation of computing:

$$R \bowtie S = \{(r, s) \in R \times S | sim(r, s) \geq \tau_n\}$$

where $sim(\cdot, \cdot)$ corresponds to a function used to calculate the similarity degree between each (r, s) set pair. Sets consist of elements from a finite universe $E = \{e_1, e_2, \ldots, e_m\}$ with m the number of distinct elements.

Similarity Functions. To measure the similarity between sets, normalized similarity functions such as Jaccard, Cosine and Dice are typically used. The given normalized threshold τ_n is translated to an equivalent overlap τ and thus, a pair (r, s) is considered similar only if $|r \cap s| \geq \tau$. In addition, τ_n can be further used to denote the set size range for all possible candidates of a set r. Table 1 shows the τ_n translation and size bounds for the Jaccard similarity function.

Self Join. Most commonly, the set similarity join is investigated as a self-join using only a single collection of sets ($R = S$). However, non self-joins can be transformed to self-joins as discussed in [5].

Data Layout. Set elements are sorted by their frequency in increasing order, so that infrequent elements 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. An example collection R consisting of ten sets is illustrated in Figure 1. In this figure, the sets are sorted by their size in ascending order. For instance, consider the set r_8, the first element of which is the most infrequent one in the complete dataset, i.e. e_1 and in contrast, the most frequent element, i.e. e_{16}, is at the end. This data layout is preferred in order to enable effective filtering, as we explain next.

2.2. Existing Solutions

Existing solutions for the set similarity join problem conform to a filter-verification framework (Algorithm 1) with the majority focusing on how the filtering cost could be decreased. As a result, several filters have been proposed. We examine the most established and effective filters.

r_1	$\{e_{13}, e_{15}\}$
r_2	$\{e_4, e_7, e_{13}, e_{14}, e_{16}\}$
r_3	$\{e_6, e_9, e_{12}, e_{15}, e_{16}\}$
r_4	$\{e_9, e_{11}, e_{14}, e_{15}, e_{16}\}$
r_5	$\{e_{10}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_6	$\{e_8, e_{10}, e_{11}, e_{12}, e_{14}, e_{15}, e_{16}\}$
r_7	$\{e_5, e_8, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{16}\}$
r_8	$\{e_1, e_2, e_7, e_8, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_9	$\{e_3, e_5, e_9, e_{10}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}\}$
r_{10}	$\{e_6, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{14}, e_{15}, e_{16}\}$

Figure 1: An example collection of records R.

Algorithm 1 Filter-Verification Framework

Input: A collection of sets R, a threshold τ_n
Output: All similar *pairs* with $sim(r_i, r_j) \geq \tau_n$

```
1: I ⟵ index(R)
2: for each set r_i ∈ R do
3:     C ⟵ {}
4:     for each element e ∈ pre_{π_i}(r_i) do
5:         if (r_i, r_j) ∉ C and |r_j| − pos_e(r_j) + 1 ≥ τ then
6:             C ⟵ C ∪ {(r_i, r_j)}
7:     for each pair (r_i, r_j) ∈ C do
8:         if |r_i ∩ r_j| ≥ τ then
9:             output(r_i, r_j)
```

2.2.1. Filters

There are two basic filters that rely on the existence of an index-like data structure, namely, the prefix and the partition filter. Both have the highest filtering potential among all filters found in literature. Based on which filter they use, existing algorithms can be categorized into (i) prefix-based, and (ii) partition-based.

Apart from basic filters, there are other simpler filters that exploit the given threshold value and set size. Nevertheless, more sophisticated filters exist, such as the one presented in [12]. We give a concise description for each of the examined filters below. We also omit the partition filter since our framework's basic filter is prefix-based.

Prefix filter. The first applied filter, called prefix-filter, examines 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. More specifically, a prefix of a set r_i, denoted as $pre_{\pi_i}(r_i)$ is formed by the $\pi_i = |r_i| - \tau + l$ first tokens of the set, where l is the required overlap. Thus (r_i, r_j) is considered a candidate pair if

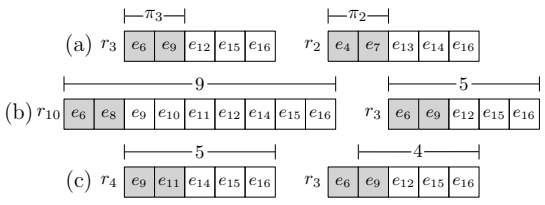

Figure 2: Example of the most established filters: (a) prefix, (b) length, (c) positional.

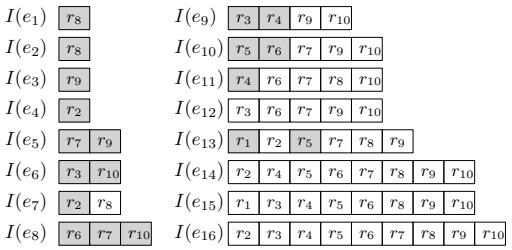

Figure 3: The complete inverted index for collection R. The materialized index is highlighted in gray cells ($\tau_n = 0.8$, $l = 1$).

$$|pre_{\pi_i}(r_i) \cap pre_{\pi_j}(r_j)| \geq l$$

In Figure 2(a), for $\tau_n = 0.8 \rightarrow \tau = 5$ and $l = 1$, 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 5 or higher cannot be reached, and in such cases, the candidate pair can be safely pruned. Since sets in R are processed in increasing size order, no candidate set r_j is longer than the current probing set r_i. This enables the use of shorter prefixes of size $\pi_i = |r_i| - \lceil lb_{r_i} \rceil + l$ for indexing which consequently results in a smaller inverted index. An example of such inverted index can be seen in Figure 3.

Length filter. Another filter, known as length filter, takes advantage of the normalized similarity functions dependency on set size. Hence, a set r_j is considered a candidate pair of r_i if

$$lb_{r_i} \leq |r_j| \leq ub_{r_i}$$

In Figure 2(b), if $\tau_n = 0.8$, the shown candidate pair (r_{10}, r_3) is pruned by length filter despite the prefix overlap because set r_{10} requires a candidate set r_j of size $8 \leq |r_j| \leq 11$.

Positional filter. Given the first match position for an element e, denoted as pos_e, positional filter evaluates if a candidate pair can ultimately reach the required overlap. Thus (r_i, r_j) is a candidate pair if

$$|r_j| - pos_e(r_j) + 1 \geq \tau$$

As an example, in Figure 2(c), the first match position is $pos_{e_9}(r_3) = 2$, and thus for $\tau_n = 0.8 \rightarrow \tau = 5$, the pair (r_4, r_3) is pruned since the remaining tokens from set r_3 are not enough to reach the required overlap.

Bitmap filter. The authors of [12] 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 collection elements 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. If the upper bound is less than the minimum required overlap the candidate can be safely pruned. More formally, (r_i, r_j) is considered a candidate pair if

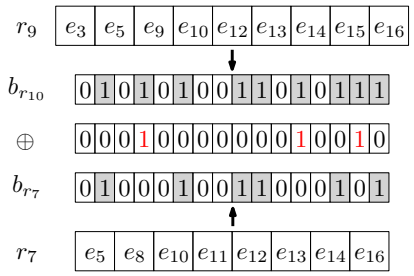

Figure 4: Candidate pair (r_9, r_7) can be safely pruned for $\tau_n = 0.8 \rightarrow \tau = 8$ since its expected overlap upper bound is 7 (Adapted from [12]).

$$\lfloor \frac{|r_i| + |r_j| - popcount(b_{r_i} \oplus b_{r_j})}{2} \rfloor \geq \tau$$

with b_{r_i},b_{r_j} the bitmap signatures of r_i,r_j respectively, and *popcount* the operation of counting the hamming distance of the bitmap signatures. Figure 4 illustrates the application of bitmap filter on the (r_9, r_7) candidate pair.

2.2.2. CPU Algorithms

The main difference between existing CPU algorithms for set similarity join is observed in the filtering phase and specifically on what filters each employs. We summarize the best three among seven state-of-the-art main memory algorithms as reported in [9]. The first algorithm to involve the prefix and length filters, noted as *AllPairs*, was proposed in [4]. The authors of [5] introduce *PPJoin*, extending AllPairs by applying the positional filter during candidate generation. Last, in [6] introduce *GroupJoin* as an extension to PPJoin. In GroupJoin sets with identical prefix are grouped together and each group is handled as a single set. This results in faster filtering, as candidate pairs are discarded in batches. During the verification phase, the candidate pairs are expanded.

The key observation provided in [9] is that all the evaluated algorithms have small performance differences except those which involve sophisticated filtering and that future techniques should investigate on lower overhead filters. To this end, the authors of [12], developed and incorporated bitmap filter on the best performing CPU algorithms. As a result, they manage to achieve speedups of up to 4.50×.

2.2.3. Leveraging Set Relations

The majority of the techniques proposed for set similarity join examine each set independently. This results in an accumulated computational cost and possible work overlap. Motivated by this, Wang et al. [13] introduce two skipping techniques, on top of prefix-based methods, which leverage relations among sets and achieve a computational cost decrease through shared computations.

Moreover, in the first skipping technique, noted as *index-level skipping*, the inverted index is rearranged so that set elements indexed in the same inverted list are partitioned into different blocks. Each block consists of sets of the

same size and its entries are sorted in non-decreasing order of the set element position on the corresponding sets. Thus, for a probe set, whenever an entry fails the position filter all the remaining unprobed entries in the same block are skipped. This also applies for the next sets to be evaluated. As a result, by exploiting index relations among sets, there is a significant reduction on the amount of redundant index probes.

The second skipping technique, noted as *answer-level skipping*, takes advantage of the possibility that two similar sets may also have similar answer-sets. An answer-set $A(r_i)$ is defined as the collection of similar sets for set r_i. For each similar set record r_j where $(r_i, r_j) \in A(r_i)$, using a cost estimation, it is determined if $A(r_j)$ will be computed from scratch or derived from $A(r_i)$. The latter leads to a complete skip of the r_j evaluation.

2.2.4. Using the GPU

Set similarity join is mostly investigated in a non-parallel environment. As shown in [11], prominent speedups for set similarity join in a parallel environment emerge from incorporating the GPGPU paradigm. The techniques that employ the GPU fall into two categories: (i) those that move the whole workload on the GPU, and (ii) those that split the workload between the CPU and the GPU. We give a concise overview of each category below.

Standalone GPU. To process collections of arbitrary size, state-of-the-art standalone GPU techniques adopt a block partitioning scheme. Thus, the input collection is divided into blocks that fit in GPU memory. Workload is evenly distributed among GPU cores using the inverted index in case of prefix filter or by bitmap signatures in case of bitmap filter. In general, the GPU outperforms single core implementations and can lead up to two orders of magnitude speedup. However, this is not always the case. In particular, to conduct the join, the GPU must allocate and process an $O(n^2)$ memory space on each invocation. This results in an accumulated overhead, which may dominate runtime[1].

Cooperative CPU-GPU. In certain cases, where the GPU standalone techniques do not perform better than single core implementations, a cooperative solution that splits workload among CPU and GPU is preferred. More specifically, filtering is conducted on the CPU, while verification is delegated to the GPU. As a result, there is an execution overlap which leads up to 2.5x speedup over the single threaded CPU implementations [14], which is a previous work of ours. In the current work, we also combine CPU and GPU but in a different manner; where the technique in [14] can be combined with CPU-only solutions.

Table 1: Normalized threshold τ_n translation and size bounds for the $Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|}$ similarity function (Adapted from [9]).

Equivalent overlap (τ)	$\frac{\tau_n}{1+t_n}(\|r\| + \|s\|)$
Size lower bound (lb_r)	$\tau_n \|r\|$
Size upper bound (ub_r)	$\frac{\|r\|}{\tau_n}$

Table 2: Notation summary.

Notation	Description
$R = \{r_1, \ldots, r_{\|R\|}\}$	A collection of records
$E = \{e_1, \ldots, e_{\|E\|}\}$	Element universe set
$R_i = R_1^i \cup \cdots \cup R_{\|R_i\|}^i$	A partition of blocks
R_j^i	The j-th block of the i-th partition
τ_n	Normalized threshold
n	Block size
p	Number of partitions

3. Framework

In this section, we present our hybrid framework. With the exception of the cooperative CPU-GPU technique, every other technique works as standalone on either the CPU or the GPU. Consequently, this results to inactivity of the other end. The main goal of our framework is to minimize this inactivity by splitting the join workload in smaller chunks and utilize both ends efficiently.

First, we introduce our partitioning scheme which leverages two existing GPU-based techniques [15, 14] and combines them with CPU-based solutions. Next, we propose two workload allocation strategies to utilize both the CPU and GPU. Finally, we give a concise overview of our framework's pipeline and provide further details on the implementation.

3.1. Partitioning Scheme

CPU-standalone techniques process the complete join in a loop-style fashion where each iteration outputs similar sets for a specific set. On the other hand, GPU-standalone techniques process the complete join in batches due to the GPU's limited memory space and inefficiency in dynamic memory allocation. We conform to a two-level partitioning scheme that supports both.

Our two-level partitioning scheme is composed of two type of segments: (i) block, and (ii) partition. On the first level, input collections are divided into blocks of size n so that the required $O(n^2)$ memory space fits in the GPU. On the second higher level, input collections are divided into p partitions. Essentially, a partition R_i is a segment consisting of a sequence of blocks $(R_0^i, R_1^i \ldots, R_{|R_i|}^i)$. Figure 5 illustrates our partitioning scheme. We summarize frequently used notations in Table 2.

[1]We refer the reader to [11] for more details on the quadratic space overhead.

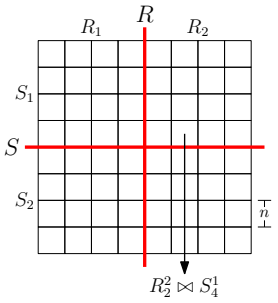

Figure 5: Partitioning scheme, each both input collections are divided into partitions ($p = 2$) that are composed of blocks of size n.

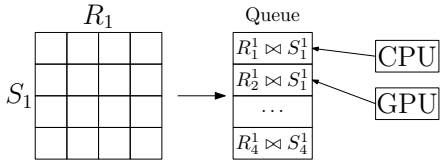

Figure 6: Concurrent queue workload allocation strategy.

3.2. Workload Allocation Strategies

In order to utilize both ends concurrently and efficiently, workload allocation is of paramount importance. Based on the partitioning scheme, the GPU supports only joins between blocks, whereas the CPU supports also joins between partitions. In respect to this, we develop two workload allocation strategies. We describe each strategy below.

Concurrent Queue. The most evident approach to keep both the CPU and GPU utilized is by splitting the complete join workload into smaller joins and have each end, as soon as it becomes available, process a portion of them. Hence, we split the join matrix into pairs of blocks which are used to populate a concurrent lock-free queue. Thus, both ends can run concurrently and independently until the queue is empty. An overview of our concurrent queue strategy is depicted in Figure 6.

Dichotomy. Another approach is by dichotomizing the complete join workload. To achieve this, we split the probe collection into two partitions ($p = 2$), i.e. two separate joins between partitions, and assign each to either the CPU or the GPU. For the GPU, the partition join is decomposed to joins between blocks. Figure 7 illustrates the dichotomy work allocation strategy.

Table 3: An overview of the applicable filters.

		CPU	GPU	CPU-GPU
main	prefix	✓	✓	✓
	bitmap		✓	
auxiliary	length	✓	✓	✓
	positional	✓	✓	✓

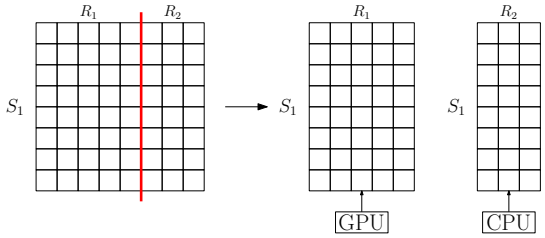

Figure 7: Dichotomy workload allocation strategy.

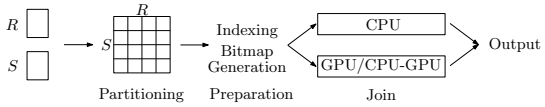

Figure 8: Framework pipeline.

3.3. Overview and Technique Selection

Since there is no dominant solution for the set similarity join problem as shown in [11], our framework must encapsulate the best performing techniques. We enumerate three categories of available techniques, (i) CPU-standalone and (ii) GPU-standalone, where the complete join is conducted exclusively on either side, and (iii) cooperative CPU-GPU where filtering is conducted on the CPU and verification on the GPU.

We categorize the applicable filters per category of techniques into (i) *main*, i.e. those which require an external structure (either an index or bitmap signatures), and (ii) *auxiliary*, i.e. those which exploit only set size to determine if the required overlap can be met. Table 3 summarizes the filters used in our framework. Bitmap can also be used as a main filter for the CPU-enabled techniques alongside prefix filter. However, the speedup gains in this case for the best performing CPU techniques are relatively small, up to 1.35x on average [12]. In addition, length filter can also be used at block level which enables pruning of whole blocks.

Figure 8 depicts our framework's pipeline. Initially we partition input collections and proceed to the preparation step where, based on our work allocation strategy, we build the inverted index, and if necessary the bitmap signatures. Then, we launch two separate threads, (i) a CPU-only thread that executes only CPU-standalone techniques, and (ii) a GPU-enabled thread that may run GPU-standalone or cooperative CPU-GPU techniques. Both these threads run in parallel until the whole process is finished.

In order to determine the most effective filters and techniques per execution, we rely on the findings of the work presented in [11]. There, the authors distinguish three key dataset characteristics, (i) dataset cardinality, (ii) average set size, and (iii) the number of different elements, and discuss on how they affect the performance. Dataset cardinalities in the order of 10^5 are referred to as small, the ones in the order of 10^6 medium, and those in 10^7 as large. When the average set size is less than 10, it is referred to as small; otherwise, as large. Finally, the number of dif-

ferent elements is characterized as small up to the order of 10^4, and otherwise, as large.

Table 4 presents the summary of the best performing techniques as reported in [11]. We consider threshold values of 0.9 as very high, 0.8 as high and $0.5-0.7$ as medium. We do not consider threshold values lower than 0.5 where filtering is not effective and only the alternative of brute-forcing the output is the best performing approach. As it can be seen in Table 4, the GPU-standalone prefix technique handles better non large datasets on medium and high threshold values. As the dataset cardinality increases, cooperative CPU-GPU techniques tend to perform better on medium and high thresholds. Respectively, the CPU-standalone prefix technique remains competent regardless of the dataset cardinality on very high thresholds. Last, the GPU-standalone bitmap technique is quite efficient on medium thresholds for medium dataset cardinalities combined with low number of different elements and high average set size.

By using the findings of [11] as a baseline, we transit to a hybrid solution more conveniently. More specifically, since we have a good overview on each technique's strong and weak points, we can select the best performing ones per scenario and adapt them to our work allocation strategies. For example, it is evident that for large datasets and high to very high thresholds, the GPU-standalone techniques do not perform well. Therefore, in these cases, we prefer to run concurrently CPU-standalone techniques alongside cooperative CPU-GPU techniques. We provide further details in Section 4.

Algorithm 2 Hyset Queue

Input: R, n, τ_n, Bitmap size bmp
Output: All similar $pairs$ with $sim(r_i, r_j) \geq \tau_n$

1: $blocks \longleftarrow divide(R, n)$
2: $I \longleftarrow \{\}$
3: **for each** block $b_i \in blocks$ **do**
4: $\quad I \longleftarrow I \cup \{index(b_i, \tau_n), offsets(b_i)\}$
5: $bitmaps \longleftarrow \{\}$
6: **if** $bmp > 0$ **then**
7: $\quad bitmaps \longleftarrow constructBitmaps(R, bmp)$
8: $queue \longleftarrow populate(blocks)$
9: **while** $queue \neq \emptyset$ **do**
10: $\quad cpuJoin(queue.pop(), I, \tau_n)$ } Runs in parallel
11: $\quad gpuJoin(queue.pop(), I, bitmaps, \tau_n)$ }

3.4. Implementation details

We give an algorithmic overview of our two work allocation strategies in Algorithms 2 and 3. In order to incorporate the state-of-the-art techniques in a single framework, there are some implementation peculiarities. The first one concerns how index data might be shared between the CPU and GPU, while the second one concerns the effect of the selected work allocation strategy on the indexing step.

Algorithm 3 Hyset Dichotomy

Input: R, n, τ_n, Bitmap size bmp, Dichotomy point x
Output: All similar $pairs$ with $sim(r_i, r_j) \geq \tau_n$

1: $R_0 \longleftarrow R[0 - x)$
2: $R_1 \longleftarrow R[x - |R|)$
3: $I_0 \longleftarrow \{index(R_0, \tau_n), offsets(R_0)\}$
4: $bitmaps \longleftarrow \{\}$
5: **if** $bmp > 0$ **then**
6: $\quad bitmaps \longleftarrow constructBitmaps(R, bmp)$
7: $cpuJoin(R_0, I_0, \tau_n)$ } Runs in parallel
8: $gpuJoin(R_1, R, bitmaps, n, \tau_n)$ }

Since prefix filter is supported in every category of techniques, we design a uniform inverted index structure in order to be used by both the CPU and GPU conveniently. Thus, the inverted index consists of two linear memory arrays, (i) *index*, and (ii) *offsets*. The former holds inverted lists in a sequence, while the latter is used to delimit each inverted list boundaries.

We delegate indexing to the GPU since it can be regarded as a composition of two primitives, sort and prefix sum, in which the GPU excels the CPU. However, depending on the selected work allocation strategy, there are different memory requirements. For the concurrent queue, since joins are conducted at a block level, the required memory space for the offsets array is increased by a factor of $\frac{|R|}{n}$. On the other hand, that is not the case for the partition join, since the inverted index is constructed at a partition level and the required memory for the offsets array is increased only by a factor of p.

In addition, we note that for the concurrent queue, the complete inverted index is constructed on the GPU and copied back in main memory where it resides throughout the complete process (Algorithm 2, lines 3-4). Whenever the GPU is to conduct a block join, we transfer the corresponding block's inverted index to the GPU memory. In case of dichotomy, we build the inverted index to be used by the CPU beforehand (Algorithm 3, line 3) and construct every inverted index required for a block join on the GPU (Algorithm 3, line 8) on the fly .

For the bitmap filter, since it is only available for the GPU part, we generate and store all bitmap signatures in the GPU memory (Algorithm 2, lines 6-7, Algorithm 2, lines 5-6). Thus, for both work allocation strategies, the CPU relies on the inverted index whereas the GPU is index-independent.[2]

4. Evaluation

In this section, we experimentally evaluate our hybrid framework with the state-of-the-art techniques found in

[2]Source code is publicly available from https://github.com/chribell/hyset.

Table 4: Summary of the best techniques as reported in [11].

τ_n	Dataset cardinality - Average set size							
	small-small		small-large	medium-large		large-small	large-large	
Very high (0.9)	GPU-prefix	CPU-prefix	CPU-prefix	GPU-prefix	CPU-prefix	CPU-prefix	CPU-prefix	CPU-GPU
High (0.8)	GPU-prefix		GPU-prefix	GPU-prefix	CPU-GPU	CPU-GPU	CPU-GPU	
Medium (0.5-0.7)	GPU-prefix		GPU-prefix	GPU-prefix	GPU-bitmap	CPU-GPU	GPU-prefix	CPU-GPU

Table 5: Dataset characteristics.

Dataset	Cardinality	Average set size	# different elements
original			
AOL	$1.0 \cdot 10^7$	3	$3.9 \cdot 10^6$
BMS	$5.1 \cdot 10^5$	6.5	1657
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	$1.5 \cdot 10^4$
ENRON	$2.5 \cdot 10^5$	135	$1.1 \cdot 10^6$
KOSARAK	$1.0 \cdot 10^6$	8	$4.1 \cdot 10^4$
LVJ	$3.1 \cdot 10^6$	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	$3.7 \cdot 10^4$
increased			
BMS-25	$1.3 \cdot 10^7$	6.5	1681
ENRON-25	$6.1 \cdot 10^6$	135	$1.1 \cdot 10^6$
LVJ-5	$1.5 \cdot 10^7$	36.5	$7.5 \cdot 10^6$

Table 6: Synthetic datasets' characteristics.

Cardinality	5M, 10M, 20M
# different elements	50K, 500K
Average set size	5, 25

literature and conduct in-depth analysis of the results. We begin by providing some preliminaries on our experimental environment and dataset characteristics. We proceed with our main experiments and evaluate our work allocation strategies. Finally, we report end-to-end runtimes and conclude with some overall observations.

4.1. Experimental Environment

In order to incorporate the CPU-standalone techniques and the cooperative CPU-GPU techniques to our framework, we modify the original implementations, provided by [9] and [11] respectively. For the GPU-standalone techniques we use the implementation of [15] for the prefix filter and of [11] for the bitmap filter.

The experiments were conducted on a machine with an Intel i7 5820k clocked at 3.3 GHz, 32 GB RAM at 2400 MHz and a NVIDIA Titan XP on CUDA 10.1. This GPU has 3840 CUDA cores, 12 GB of global memory and a 384-bit memory bus width. We compile our code with g++/nvcc with the -O3 flag.

We focus on self-joins using the Jaccard similarity and encapsulate in our framework, main-memory techniques on a single machine as in previous works [9, 11]. Thus, datasets alongside any other required data, must fit in the RAM main-memory of the host machine. We perform an aggregation on top of the set similarity join to measure the count of the results but after producing the full result set.

The reported runtime for each experiment, noted as join time, is the average over 3 independent runs. Since there is no significant deviation, no higher number of repeated runs is needed. We do not include any data preprocessing and initial I/O read time unless explicitly stated. Note that GPU data transfer times (i.e., transferring the initial datasets and/or the indices from RAM to the GPU and the results from the GPU to RAM) are also included in the join time. We measure the total join time with the std::chrono library. Correspondingly, we use the CUDA event API to measure the GPU operations.

We conduct experiments on three types of datasets: (i) original, choosing the real-world datasets that have been previously used in the literature, (ii) increased, where we inflate specific original datasets by an appropriate factor in order to test the performance in even greater datasets, and (iii) synthetic, where we create artificial datasets using different combinations of dataset characteristics. Table 5 shows an overview of the characterictis of each original and increased dataset. Accordingly, for the synthetic datasets the combination of characteristics are listed in Table 6. The original datasets can be found in [9] and [11]. We describe the steps followed to reproduce the increased and synthetic datasets further below.

4.2. Main Experiments

We compare our two hybrid work allocation strategies denoted as *queue* and *dichotomy* with the state-of-the-art single threaded CPU-standalone and GPU-enabled techniques for the set similarity join. We present the best join times measured for all the techniques for the original datasets in Table 7 and our speedups in Table 8. Respectively, for the increased datasets, the join times are presented in Table 9 and our speedups in Table 10. Last, for the synthetic datasets, the join times are presented in Table 11 and our speedups in Table 12. We note every unsuccessful test, either due to memory constraints or very long run time, as not available (N/A). Each time reported for the *CPU* is the overall best among the three best performing algorithms (i.e. Allpairs, PPJoin, GroupJoin) as stated in [9]. To account for advances after the publication of [9], we also report times for the skipping algorithm presented in [13], denoted as *SKJ*, which has a single threaded CPU implementation. The source code of SKJ was provided by the authors of [13]. Respectively, for the *CPU-GPU* each time reported is the overall best among the three CPU algorithms and the best GPU verification techniques described in [14]. For the *GPU*, each time reported is the best between the total times for all the GPU operations

required to perform the set similarity join, using either the prefix or bitmap filter in line with the discussion in Section 3.3.

We conduct experiments for all the original datasets in the threshold range $\tau_n \in [0.5 - 0.9]$. Respectively, for all the increased and synthetic datasets, we narrow the threshold range to $\tau_n \in [0.7 - 0.9]$ due to large overall join times. In addition, for our partitioning scheme, we set a global block size $n = 10000$. In general, this block size is smaller than the available GPU global memory available (after storing the data and indices), but we have found that it leads to lower amortized overhead for memory cleaning and more efficient length filtering at the block level. We analyze the experimental results for each dataset category in turn.

Original datasets. We present the main results in Table 7, where, the *dichotomy* technique is restricted so that each type of device is allocated at least 20% of the workload. As can be seen from the table, in 70% of the cases, our hybrid techniques achieve speedup over the state-of-the-art techniques. We observe 1.68x speedup on average, with the largest one achieved at 3.05x for the ORKUT dataset with $\tau_n = 0.9$. Table 8 provides an overview of the speedups per dataset and per threshold for the original datasets.

On the other hand, for the rest 30% of the combinations of datasets and thresholds, none of the hybrid techniques yield any performance speedup, when the dichotomy technique is forced to allocate at least 20% to a second type of device. This happens for three reasons. First, in certain datasets such as BMS, DBLP with its variations and LVJ, for very high thresholds, i.e. $\tau_n = 0.9$, standalone prefix filtering is quite effective and the overall join times are in general very low; in these cases the queue-based hybrid technique is slightly worse than the optimal solution. Second, for datasets such as DBLP-1M and TWITTER with threshold $\tau_n \in [0.5 - 0.8]$, GPU-standalone bitmap filter is the fastest solution due to its high filtering ratio (effectiveness) and efficiency. In contrast, incorporating the CPU by using prefix filter in such cases only adds overhead. Third, there is the case of AOL dataset, where for threshold $\tau_n \in [0.5 - 0.6]$, the skipping algorithm SKJ [13] is the fastest solution.

We defer the discussion about the cases where SKJ dominates when we consider the synthetic datasets. Regarding the first and second reasons above, we need to stress that they are direct consequence of splitting the workload in the dichotomy technique so that a processor takes at least 20% of the workload. If we allowed either CPU or GPU to process up to 100% of the workload and we had a mechanism to derive the optimal splitting point, then the dichotomy would be the best performing solution in all cases, apart from those where SKJ dominates (96% of the cases).

The bottom line of our observations is that the hybrid framework, even without fine-tuning is capable of yielding tangible benefits. If we perform fine tuning and allow dichotomy to split the workload arbitrarily, then the dichotomy would perform better than what reported in Table 7, while queue-based hybrid would become obsolete.

Increased datasets. We artificially increase the BMS, ENRON and and LVJ datasets using the method employed in [16]. When increasing the datasets, the benefits of our hybrid solutions become even more significant. As reported in Table 9, for the 89% of the total experiments, we measure 1.94X speedup on average, with the largest speedup observed being 3.25X for the ENRON-25 dataset with $\tau_n = 0.7$. Please note the threshold in these experiments is 0.7 to 0.9; for lower threshold, the speedups increase on average. As an exception, for the LVJ-5 dataset with $\tau_n = 0.7$ the SKJ algorithm is faster.

The respective speedups for the original datasets is 1.26X, i.e., our hybrid techniques achieve further improvement by 70%. We present the exact speedups in Table 10. When the prefix filter is efficient, both the CPU and GPU can contribute greatly to the set similarity join. Hence, when the dataset cardinality increases, our hybrid techniques can utilize both edges quite effectively and as a result, speedups become more evident. However, we note the inability to run our queue hybrid flavor due to memory constraints in some cases. We further analyze the queue's memory cost in Section 4.3.

Synthetic datasets. We create twelve artificial datasets using the combination of characteristics list in Table 6. In addition, the synthetic follow a zipf-like element frequency distribution. We use the original script provided by the authors of [9]. To distinguish each dataset, we use the notation *Dataset cardinality - Number of different elements - Average set size.*

The aggregate results for the synthetic datasets are presented in Table 11. For the 44% of the total experiments our hybrid techniques achieve 1.48X speedup on average. The largest speedup measured is 2.22x for the 10M-500K-5 with $\tau_n = 0.9$. For the rest 56%, SKJ achieves 2.08X speedup on average over the state-of-the-art and 1.49X over our hybrid techniques. Table 12 shows the speedups of our hybrid techniques for the synthetic datasets. By comparing the hybrid solutions against SKJ, we see a pattern on the runtimes which is independent of the dataset cardinality and, instead, is mainly based on the number of different elements and average set size or a combination of both. When the average set size is small (5), hybrid techniques perform well since prefix filtering on both the CPU and GPU remain competent, especially for threshold $\tau_n \geq 0.8$. However, as the average set size increases, SKJ performs better because of the computational cost sharing it encapsulates. In addition, with the combination of a low number of different elements (50K), sets have higher probability of sharing common elements, which favors SKJ.

However, when SKJ behaves better, we need to keep in mind that SKJ is constructed in a different manner, where indexing is performed out of the filtering phase. Thus, the

Table 7: Join Times for the original datasets (in seconds).

		Threshold τ_n				
		0.5	0.6	0.7	0.8	0.9
AOL	CPU	276.16	69.32	10.41	3.23	1.21
	GPU	693.21	466.36	310.14	239.87	202.56
	CPU-GPU	207.12	50.41	7.31	2.52	1.01
	SKJ	**126.74**	**29.71**	7.22	3.67	2.16
	Queue	142.63	65.42	29.13	20.78	16.92
	Dichotomy	163.12	49.31	**6.00**	**2.47**	**0.74**
DBLP-200K	CPU	442.30	228.59	104.11	23.13	2.33
	GPU	8.21	6.73	4.39	**1.47**	**0.41**
	CPU-GPU	201.60	128.08	57.83	19.14	2.81
	SKJ	615.02	212.01	52.11	8.14	0.86
	Queue	4.64	4.16	2.81	1.68	0.54
	Dichotomy	**3.81**	**3.51**	**2.80**	2.09	1.77
DBLP-1M	CPU	N/A	N/A	4560.34	1157.92	96.76
	GPU	72.97	**40.09**	**29.11**	**16.95**	**6.76**
	CPU-GPU	N/A	N/A	2023.29	463.16	41.17
	SKJ	N/A	N/A	N/A	234.06	16.60
	Queue	**63.60**	44.61	35.53	19.5	6.91
	Dichotomy	102.86	41.63	33.44	17.13	6.93
KOSARAK	CPU	113.57	30.66	10.22	7.33	7.05
	GPU	12.2	6.23	3.35	2.63	2.24
	CPU-GPU	100.5	26.69	7.66	5.01	4.63
	SKJ	52.10	13.40	5.27	4.38	4.13
	Queue	**9.37**	**3.74**	**1.74**	**1.16**	**0.95**
	Dichotomy	18.99	6.06	3.22	2.41	2.01
ORKUT	CPU	161.40	59.51	24.13	9.72	3.23
	GPU	78.08	38.73	19.60	10.49	5.32
	CPU-GPU	128.58	57.51	23.73	10.32	3.36
	SKJ	397.64	128.28	46.14	19.58	9.46
	Queue	97.28	42.99	20.90	12.05	8.26
	Dichotomy	**27.41**	**13.87**	**6.87**	**3.18**	**1.61**

		Threshold τ_n				
		0.5	0.6	0.7	0.8	0.9
BMS	CPU	41.12	13.67	4.20	1.06	0.50
	GPU	3.51	2.07	1.09	0.83	0.60
	CPU-GPU	27.36	9.51	3.19	1.14	**0.31**
	SKJ	19.99	6.38	2.22	0.88	0.34
	Queue	**3.08**	**1.62**	**0.89**	**0.59**	0.39
	Dichotomy	4.39	2.49	1.37	1.03	0.79
DBLP-300K	CPU	1151.44	569.53	215.72	52.67	5.58
	GPU	17.20	14.18	9.86	3.21	**0.79**
	CPU-GPU	473.15	278.32	135.24	39.87	6.89
	SKJ	1498.01	519.65	122.58	18.55	1.72
	Queue	**10.11**	6.94	4.85	3.23	0.84
	Dichotomy	12.70	**5.35**	**4.38**	**3.04**	2.07
ENRON	CPU	38.12	11.47	3.64	1.06	0.27
	GPU	4.37	1.98	0.99	0.65	0.27
	CPU-GPU	33.03	10.21	3.47	1.09	0.29
	SKJ	36.57	10.02	3.15	1.41	0.67
	Queue	7.69	3.15	1.34	0.70	0.46
	Dichotomy	**3.14**	**1.76**	**0.92**	**0.47**	**0.25**
LVJ	CPU	277.03	70.34	17.21	4.72	**1.35**
	GPU	64.7	36.73	21.98	13.47	6.82
	CPU-GPU	187.73	49.56	13.82	4.46	1.36
	SKJ	235.23	145.51	97.03	50.25	25.67
	Queue	71.96	33.33	16.41	9.48	6.22
	Dichotomy	**54.49**	**26.94**	**9.12**	**3.51**	1.82
TWITTER	CPU	N/A	N/A	4697.91	897.66	63.39
	GPU	**114.01**	**83.47**	**47.7**	**29.86**	9.56
	CPU-GPU	N/A	N/A	3001.98	557.41	46.19
	SKJ	N/A	N/A	1385.29	189.09	16.17
	Queue	143.36	107.95	64.04	37.96	**8.55**
	Dichotomy	136.75	110.51	64.54	42.09	9.62

Table 8: Speedups for the original datasets.

	Threshold τ_n					Overall
	0.5	0.6	0.7	0.8	0.9	
AOL	-	-	1.2	1.02	1.36	1.19
BMS	1.13	1.27	1.22	1.4	-	1.26
DBLP-200K	2.15	1.91	1.56	-	-	1.88
DBLP-300K	1.7	2.65	2.25	1.05	-	1.91
DBLP-1M	1.14	-	-	-	-	1.14
ENRON	1.39	1.12	1.07	1.38	1.08	1.21
KOSARAK	1.3	1.66	1.92	2.26	2.35	1.9
LVJ	1.18	1.36	1.51	1.27	-	1.33
ORKUT	2.84	2.79	2.85	3.05	2	2.71
TWITTER	-	-	-	-	1.11	1.11
Overall	1.6	1.82	1.77	1.74	1.81	

join times reported thus far, in all techniques apart from SKJ, include indexing. On the other hand, SKJ's indexing can be deemed as a cost that can be easily amortized when multiple similarity queries are submitted on the same dataset.

4.3. Performance analysis

We proceed to the performance analysis and comparison between our hybrid solutions and discuss how each one can be used. Furthermore, we highlight the memory cost of the queue technique, especially for datasets with large number of different elements. Finally, we discuss the necessity of choosing a good splitting point and emphasize its impact on the overall performance for the dichotomy technique.

Table 9: Join times for the increased datasets (in seconds).

		Threshold τ_n		
		0.7	0.8	0.9
BMS-25	CPU	7387.55	2875.11	1247.35
	GPU	705.65	420.69	255.16
	CPU-GPU	4752.49	1686.74	725.56
	SKJ	1825.30	561.79	146.16
	Queue	**416.25**	**201.20**	**91.10**
	Dichotomy	832.26	514.00	293.94
ENRON-25	CPU	2176.58	265.70	16.63
	GPU	1115.32	471.59	37.74
	CPU-GPU	1688.64	224.80	20.30
	SKJ	955.03	138.91	31.32
	Queue	405.58	94.87	21.20
	Dichotomy	**293.28**	**91.11**	**11.73**
LVJ-5	CPU	328.44	55.25	9.12
	GPU	527.75	307.2	133.93
	CPU-GPU	252.86	45.69	8.85
	SKJ	**140.11**	39.63	15.62
	Queue	N/A	N/A	N/A
	Dichotomy	152.71	**24.68**	**3.81**

Table 10: Speedups for the increased datasets.

	Threshold τ_n			Overall
	0.7	0.8	0.9	
BMS-25	1.69	2.09	1.6	1.79
ENRON-25	3.25	1.52	1.41	2.06
LVJ-5	-	1.6	2.32	1.94
Overall	1.77	1.74	1.81	

4.3.1. Queue vs Dichotomy

In order to compare our hybrid techniques, we measure the *gap factor*, i.e. the gap between the fastest and the

Table 11: Join time for the synthetic datasets (in seconds).

		Threshold τ_n		
		0.7	0.8	0.9
5M-50K-5	CPU	25.77	20.19	1.10
	GPU	110.39	77.38	33.08
	CPU-GPU	20.17	4.25	0.97
	SKJ	**11.64**	4.07	1.20
	Queue	19.47	7.47	2.76
	Dichotomy	14.86	**3.32**	**0.7**
5M-50K-25	CPU	517.40	123.19	15.37
	GPU	1115.32	471.59	37.74
	CPU-GPU	399.64	93.27	12.40
	SKJ	**133.78**	**31.87**	**5.65**
	Queue	164.93	77.29	22.81
	Dichotomy	192.77	75.10	11.52
5M-500K-5	CPU	8.04	2.30	0.58
	GPU	112.53	79.19	34.31
	CPU-GPU	6.46	2.00	0.63
	SKJ	5.16	2.00	1.09
	Queue	15.87	7.72	2.87
	Dichotomy	**4.96**	**1.40**	**0.25**
5M-500K-25	CPU	71.54	20.13	4.87
	GPU	255.58	172.18	83.06
	CPU-GPU	57.97	16.29	4.30
	SKJ	**31.40**	**10.78**	4.50
	Queue	90.30	46.70	15.16
	Dichotomy	40.48	11.07	**2.76**

		Threshold τ_n		
		0.7	0.8	0.9
10M-50K-5	CPU	104.98	18.47	3.37
	GPU	446.68	308.60	128.90
	CPU-GPU	82.28	15.00	3.18
	SKJ	**46.21**	14.89	3.11
	Queue	74.19	26.28	8.92
	Dichotomy	60.55	**10.46**	**1.92**
10M-50K-25	CPU	2137.76	511.87	55.63
	GPU	1163.30	744.10	338.31
	CPU-GPU	1620.20	370.30	43.67
	SKJ	**574.88**	**132.23**	**16.88**
	Queue	659.10	300.57	83.84
	Dichotomy	820.61	335.10	44.17
10M-500K-5	CPU	30.26	6.95	1.40
	GPU	444.42	321.22	133.07
	CPU-GPU	24.29	5.70	1.54
	SKJ	**16.61**	5.55	2.20
	Queue	56.68	26.64	8.75
	Dichotomy	18.22	**4.21**	**0.69**
10M-500K-25	CPU	285.25	71.06	13.40
	GPU	1019.61	685.91	328.83
	CPU-GPU	226.80	56.11	11.31
	SKJ	**107.05**	**33.25**	9.96
	Queue	352.05	177.90	53.98
	Dichotomy	174.68	42.72	**8.54**

		Threshold τ_n		
		0.7	0.8	0.9
20M-50K-5	CPU	422.20	66.17	11.83
	GPU	1793.57	1238.00	512.68
	CPU-GPU	346.82	54.31	10.56
	SKJ	**199.36**	60.15	9.08
	Queue	284.10	97.01	31.23
	Dichotomy	254.44	**36.26**	**6.30**
20M-50K-25	CPU	9145.80	2168.53	206.27
	GPU	4653.30	2976.64	1348.73
	CPU-GPU	6577.16	1563.72	161.67
	SKJ	**2417.44**	**585.28**	**64.83**
	Queue	2617.97	1185.89	318.88
	Dichotomy	3508.65	1459.78	99.45
20M-500K-5	CPU	114.66	22.11	3.87
	GPU	1783.23	1257.41	528.55
	CPU-GPU	90.35	17.73	3.72
	SKJ	**60.07**	18.48	4.89
	Queue	213.35	95.68	29.29
	Dichotomy	61.82	**10.65**	**1.67**
20M-500K-25	CPU	1226.02	266.27	39.43
	GPU	4087.97	2746.53	1312.94
	CPU-GPU	878.88	202.19	32.50
	SKJ	**390.52**	**112.68**	23.83
	Queue	1393.08	692.26	200.71
	Dichotomy	715.30	132.66	**19.44**

Table 12: Speedups for the synthetic datasets.

	Threshold τ_n			Overall
	0.7	0.8	0.9	
5M-50K-5	-	1.22	1.38	1.3
5M-50K-25	-	-	-	-
5M-500K-5	1.04	1.42	2.32	1.59
5M-500K-25	-	-	1.55	1.55
10M-50K-5	-	1.42	1.61	1.52
10M-50K-25	-	-	-	-
10M-500K-5	-	1.31	2.02	1.67
10M-500K-25	-	-	1.16	1.16
20M-50K-5	-	1.49	1.44	1.46
20M-50K-25	-	-	-	-
20M-500K-5	-	1.43	2.32	1.87
20M-500K-25	-	-	1.22	1.22
Overall	1.04	1.42	1.66	

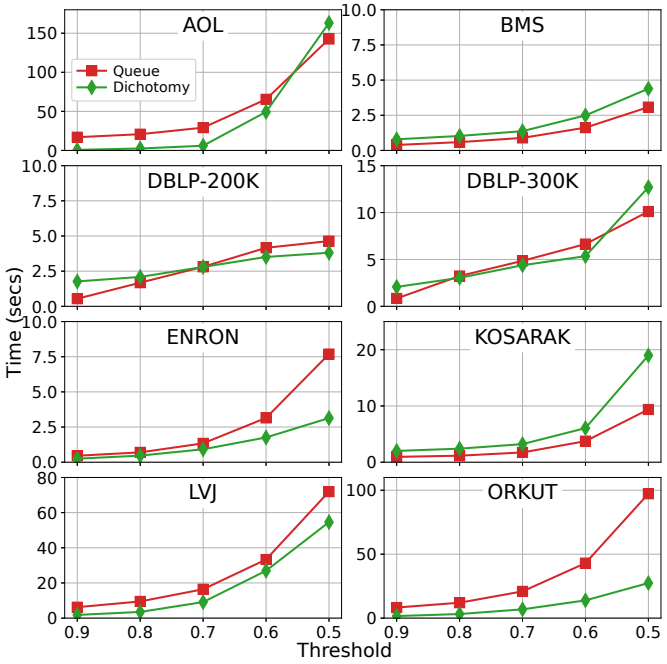

Figure 9: Comparison between the two work allocation strategies for the original datasets.

slowest technique per dataset and threshold between our two hybrid proposals. As shown in Table 13, there is a large variation in the gap factor, especially in very large thresholds, where we measure a gap factor of up to 22.86 for the AOL dataset at $\tau_n = 0.9$ with dichotomy being faster than queue. As the threshold value τ_n decreases, the variation in the gap factor decreases as well.

In general, queue is inferior to dichotomy. However, there are certain cases, such as BMS, KOSARAK and BMS-25 in which queue is faster. For these datasets, the respective indices can be characterized as lightweight both (i) vertically, i.e. the number of inverted lists is relatively small due to the small number of different elements, and (ii) horizontally, i.e. the average length of an inverted list is small because of the small average set size. When this is the case, the hybrid queue technique performs better than the dichotomy technique. On the contrary, the dichotomy technique performs better in the AOL, ENRON, LVJ and ORKUT for the original datasets, in ENRON-25 for the increased datasets and in the majority of the synthetic datasets compared to the hybrid queue technique. All of these datasets, share the common feature of having larger indices.

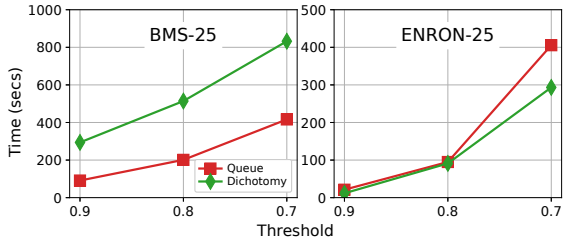

Figure 10: Comparison between the two work allocation strategies for the increased datasets.

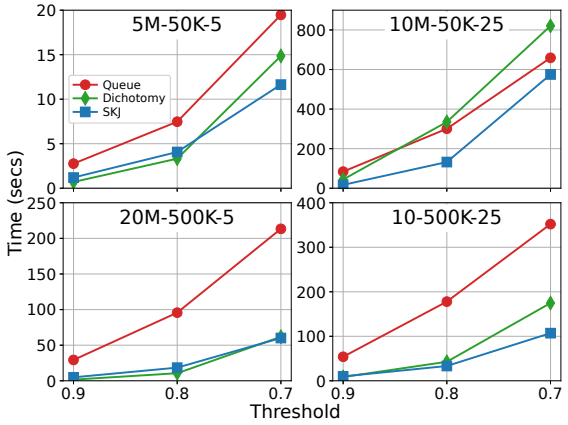

Figure 11: Comparison between the two work allocation strategies for the synthetic datasets.

Table 13: Gap factor per dataset and threshold.

Dataset	Threshold				
	0.5	0.6	0.7	0.8	0.9
AOL	1.14	1.32	4.85	8.41	22.86
BMS	1.42	1.53	1.53	1.74	2.02
DBLP-200K	1.21	1.18	1.00	1.24	3.27
DBLP-300K	1.25	1.29	1.10	1.06	2.46
DBLP-1M	1.61	1.07	1.06	1.13	1.00
ENRON	2.44	1.78	1.45	1.48	1.84
KOSARAK	2.02	1.62	1.85	2.07	2.11
LVJ	1.32	1.23	1.79	2.70	3.41
ORKUT	3.54	3.09	3.04	3.78	5.13
TWITTER	1.04	1.02	1.00	1.10	1.12
BMS-25	-	-	1.99	2.55	3.22
ENRON-25	-	-	1.38	1.04	1.80
LVJ-5	-	-	-	-	-
5M-50K-5	-	-	1.31	2.25	3.94
5M-50K-25	-	-	1.16	1.02	1.98
5M-500K-5	-	-	3.19	5.51	11.48
5M-500K-25	-	-	2.23	4.21	5.49
10M-50K-5	-	-	1.22	2.51	4.64
10M-50K-25	-	-	1.24	1.11	1.98
10M-500K-5	-	-	3.11	6.32	12.68
10M-500K-25	-	-	2.02	4.16	6.32
20M-50K-5	-	-	1.11	2.67	4.95
20M-50K-25	-	-	1.34	1.23	3.20
20M-500K-5	-	-	3.45	8.98	17.53
20M-500K-25	-	-	1.94	5.21	10.32

Further, in most cases, having larger indices favors the CPU prefix filtering, especially for high and very high thresholds. For such thresholds, it is beneficial to delegate most of the workload to the CPU and leave a smaller portion for the GPU; this can be enforced in dichotomy more efficiently. We analyze further the dichotomy workload split in Section 4.3.3.

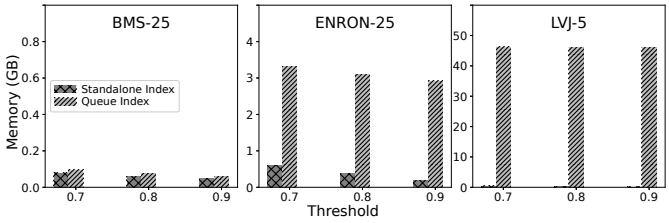

Figure 12: Index memory cost for the increased datasets with $n = 10000$.

However, the correct interpretation of the gap factor needs to take into account the absolute runtimes as presented in Tables 7, 9, 11 and Figures 9, 10, 11. More specifically, lower gap factors in lower thresholds may correspond to more significant runtime difference. For example, on the right plot in Figure 10, the runtime difference for threshold 0.7 is apparent while the gap factor is 1.38; by contrast, for threshold 0.9, the runtime difference is negligible although the gap factor is 1.8.

Additionally, for completeness, we note that both hybrid techniques perform better whenever the prefix filter is effective. When the bitmap filter is the most efficient and the dataset cardinality is relatively small, such as in DBLP-200K and DBLP-300K, employing either dichotomy or queue, using the bitmap filter for the GPU and the prefix for the CPU, seems superior. However, for larger datasets, such as DBLP-1M and TWITTER, standalone GPU bitmap performs better than any hybrid solution for not high thresholds, e.g., equal to or lower than 0.8. This also relates to the discussion in Section 4.3.3.

Finally, depending on the dataset and query characteristics, the impact of occupancy and global memory access efficiency may largely differ among winning cases; details are provided along with the source code.

4.3.2. Queue memory cost

For the queue technique to work properly, the complete index must be constructed in advance at block level. This approach leads to many small indices; in particular, as many as the number of blocks, so that both the CPU and GPU can run the join operation seamlessly without any end waiting for the other. However, this also implies an increase in the memory space required to store these indices, as for each index, the boundaries of the corresponding inverted lists must be specified. Otherwise, in a single standalone index scenario, for each index probe, several binary searches would be needed in order to determine the start and end of an inverted list. This would result in an extra overhead cost, especially in the parallel environment of the GPU.

In Figure 12, we compare the total memory required to store the complete standalone index with the corre-

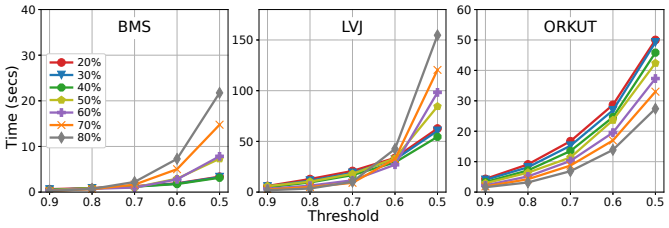

Figure 13: Comparison of different dichotomy splitting points.

sponding one required to store all the small indices for the queue technique. Although for all three datasets the number of blocks is in the order of thousands, specifically for the BMS-25 dataset, there is a small difference in the required memory for both scenarios. This is due to the small number of different elements (1657). However, for ENRON-25 and LVJ-5, the respective numbers of different elements are in the order of millions which leads to a sharp increase in the required memory space for the queue technique. Furthermore, the required memory space for the LVJ-5 dataset exceeds the available memory of our setup and as a result we cannot run the queue technique for this particular dataset.

4.3.3. Dichotomy splitting point

The main goal of the hybrid dichotomy technique is the proper workload split among the CPU and GPU in order to achieve the best possible execution overlap. However, the selection of a good splitting point is not straightforward and, in this work, its automated decision mechanism is left as future work. For each of our experiments, we ran the dichotomy technique by varying the splitting point within the range [0.2,0.8] and reported the lowest timings in Tables 7,9 and 11. We have already mentioned that if we had a mechanism to judiciously select the optimal splitting point in the range of [0,1] then the queue technique would always run slower than dichotomy; moreover, dichotomy would also always dominate the other standalone techniques.

In Figure 13, we present the runtime difference over varying dichotomy splitting points for the BMS, LVJ and ORKUT datasets. The splitting point percentage denotes the amount of workload assigned to the CPU. For the BMS and LVJ datasets, the runtime difference becomes more evident for $\tau_n < 0.7$ where filtering gradually becomes ineffective. As a result, the best performing splitting point for both datasets in lower thresholds is at 40% for the CPU and 60% for the GPU. This behavior is aligned with the findings of [11], summarized in Table 4. However, this is not the case for the ORKUT dataset where, even for higher threshold values, there is more significant relative difference between the various splitting points. Nevertheless, for this dataset, the absolute runtime differences are less significant. Also, for this dataset, it is more beneficial to allocate more work to the CPU due to the effectiveness of the prefix filter, something already discussed above.

Table 14: Hyset comparison against multi-threaded CPU alternatives. Runtimes are in seconds.

Dataset	τ_n	Multi-CPU	Fier et al. [17]	Hyset	Speedup
DBLP-1M	0.7	1623.39	5285.66	33.44	48.54
	0.8	426.38	1308.79	17.13	24.89
	0.9	54.98	98.88	6.91	7.95
KOSARAK	0.7	15.65	24.81	1.74	8.99
	0.8	15.08	19.82	1.16	13.00
	0.9	14.75	19.36	0.95	15.52
ORKUT	0.7	31.72	33.71	6.87	4.61
	0.8	14.9	17.68	3.18	4.68
	0.9	8.28	7.86	1.61	4.88
TWITTER	0.7	2208.75	6103.53	64.04	34.49
	0.8	111.298	1248.98	37.96	2.93
	0.9	74.03	89.18	8.55	8.65
BMS-25	0.7	-	7859.95	416.25	18.88
	0.8	-	2234.39	201.2	11.10
	0.9	-	722.33	91.1	7.92
ENRON-25	0.7	-	3172.74	293.28	10.81
	0.8	-	269.47	91.11	2.95
	0.9	51.24	21.08	11.73	1.79
LVJ-5	0.7	-	257.40	152.71	1.68
	0.8	-	50.98	24.68	2.06
	0.9	-	12.42	3.81	3.26

4.4. Comparison against parallel CPU-based approaches

Over the past years, there have been attempts to employ parallelism for the set similarity join problem via the MapReduce paradigm. In [10], Fier et al. evaluate the state-of-the-art MapReduce techniques for set similarity. As their main result, they highlight the poor performance of MapReduce over a single node solution and also point out its incapability to scale for larger data volumes. More recently, in [17], Fier at al. introduced a multi-threaded CPU framework for the same problem. We note that none of these works are GPU-aware. For completeness, we also implement our own multi-threaded CPU version and we directly experiment with the state-of-the-art proposal found in [17].

Our own multi-threaded implementation is denoted as *Multi-CPU*. Each time reported for the *Multi-CPU* is the best among multiple configurations of the two CPU-based algorithms AllPairs and PPJoin. We split the input dataset into p partitions equal to the number of threads launched and assign an equal number of joins to each thread along with the corresponding portion of the inverted index. This results in higher memory requirements since each thread has its own index. Also, in [17], Fier et al. introduce a more optimized multi-threaded CPU implementation for set similarity join and they claim that they achieve a $2-10$X speedup over single threaded CPU solutions. We compare our hybrid framework with their solution for four original datasets and all the increased datasets for threshold values $\tau_n \geq 0.7$. We report our runtimes in Table 14. As it can be seen, there is not a single case where either our own *Multi-CPU* or the work of Fier et al. [17] is superior to our hybrid framework. In contrast, we observe that our hybrid framework achieves a 10.75X speedup on average against multi-threaded CPU alternatives. The highest speedups reported for DBLP-1M for $\tau_n \in [0.7-0.8]$ and TWITTER

for $\tau_n = 0.7$ are due to the bitmap filter which is more suitable for the GPU. Thus, employing the GPU via our hybrid techniques seems more beneficial in every case. We also note the inability to launch our multi-threaded CPU implementation for the majority of increased datasets due to memory constraints. We also experimented with [10], and without presenting exact numbers due to space constraints, the speedups were between 91.89X and 520X in selected experiments.

The main conclusion of these experiments is that our hybrid solution cannot be outperformed by multi-threaded CPU implementations; however, as discussed next, devising efficient solutions within our hybrid context that are capable of benefiting from the multi-threaded CPU techniques is one of the identified open issues.

5. Discussion

Exact set similarity join is a notoriously expensive operation, for which several techniques and algorithms have been proposed. In a single machine setup, the most prominent solutions incorporate the use of a GPU as previously shown in [11]. To this end, we use the findings of [11] as baseline, and develop a hybrid framework that utilizes both the CPU and GPU to tackle the problem. Below, we provide key insights and discuss some generic observations in line with recent progress in GPU analytics and data management found in literature:

- Although our hybrid framework achieves speedup in the majority of cases, we highlight that the heterogeneity in existing filters hinders the possibility for even larger speedups. Moreover, the prefix filtering dependence on an index structure makes it more CPU-oriented. There are certain cases, especially for lightweight indices as shown in our queue technique, in which the GPU can contribute significantly to prefix filtering. On the contrary, bitmap filtering favors the parallel environment of the GPU exclusively, since it allows a more uniform memory access and compute utilization. As a result, when bitmap filtering is the most effective, involving the CPU does not seem beneficial in most cases.

- We identify our work as a step towards a globally dominant solution. In particular, based on our evaluation, we show that the dichotomy technique lays the foundation for a good workload allocation between the CPU and GPU. Nevertheless, our work has given rise to a new important research issue, namely the development of an automated way to select a good splitting point, alongside a cost model to select the appropriate CPU and GPU technique per scenario. Such a solution should also be able to benefit from CPU multi-thread execution, in line with the work in [17].

- Unlike other parallel paradigms such as MapReduce, for which the initial results are not so encouraging [10], the use of multiple GPUs seems to be the most promising approach for scalability provided that new faster interconnects, such as NVLink 2.0 [18, 19] are employed. This also benefits the use case of cooperative CPU-GPU techniques, since the GPU can access the main memory very fast and with low cost. In addition, execution could scale both vertically, i.e. within a single machine similarly to the work presented in [20], and horizontally in a distributed setting. Our hybrid techniques are orthogonal to the hardware improvements. Nevertheless, all of these remarks underline the necessity of developing novel techniques and algorithms.

- Recently, the work in [21], without investigating set similarity joins explicitly, argues that advanced data analytics should run on CPU only unless the initial data is already stored in GPU's global memory. Our work provides counter-evidence regarding this. Despite any overheads incurred by the CPU-GPU interconnects, careful crafting of CPU-GPU coprocessing schemes for advanced data analytics may lead to speedups of factors 3.25X, as reported in our experiments, even when the data initially resides on the CPU side. As previously, hardware advances regarding NVLink 2.0 may have a big positive impact on the efficiency of hybrid techniques.

6. Related work

Exact Set Similarity Join. There is a substantial body of literature for exact set similarity join. In [9], Mann et al. provide a comprehensive survey on set similarity join for prefix filter based techniques. Recently, Wang et al. [13] propose the SKJ algorithm which is built on top of the prefix filter and encapsulates two skipping techniques to further improve set similarity join. In [22], Deng et al. introduce the SizeAware algorithm which divides the input collection to small and large sets, and process them separately. We consider our dichotomy work allocation strategy close to the core concept of the SizeAware algorithm. For the distributed setting, Fier et al. [10] conduct an experimental survey for set similarity joins on the MapReduce framework and highlight the inability of the evaluated algorithms to scale. More recently, Yang et al. [23] devise a length-based distribution framework for set similarity join on top of the Apache Storm platform. As a result, they avoid pitfalls of previous prefix-based distributed techniques and show promising improvement on runtimes. For the GPGPU paradigm, the authors of [11] present an evaluation of the GPU-enabled techniques. We use the work of [9] and [11] as a baseline for the development of our hybrid framework and improve upon their findings.

Approximate Set Similarity Join. Most of the techniques proposed for approximate set similarity joins resort to data reduction to speedup the join process. In [24], 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. In [25], Li et al. reduces the preprocessing time from the original MinHash by using one permutation hashing. Furthermore, Ji et al. [26] introduce Min-Max hash and reduce the hashing time by half. Recently, Wang et al. [27] propose MaxLogHash to accurately estimate similarities in streaming sets. The main limitations of the above techniques is that they are inherently limited to Jaccard similarity only. In this work, we do not deal with approximate solutions for the set similarity join problem.

7. Conclusion

In this work, we introduce a novel hybrid framework, which encapsulates state-of-the-art CPU and GPU-enabled solutions for the exact set similarity join problem with a view to deriving a higher-level technique that manages to execute fast regardless of changes in the dataset and query characteristics. Through extensive evaluation and performance analysis, we show speedups of up to 3.25x over standalone solutions, and we manage to overcome the main problem of the GPU-enabled set similarity joins thus far, namely that different techniques are dominant under different conditions. We also show that we outperform multi-threaded solutions.

Acknowledgments. The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the HFRI PhD Fellowship grant (Fellowship Number: 1154). In addition, the authors gratefully acknowledge the support of NVIDIA, United States Corporation through the donation of the GPU use through the GPU Grant Program.

References

[1] E. Spertus, M. Sahami, O. Buyukkokten, Evaluating similarity measures: a large-scale study in the orkut social network, in: Proc. SIGKDD, 2005.

[2] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: Proc. ICDE, 2006.

[3] N. Augsten, M. H. Böhlen, Similarity joins in relational database systems, Synthesis Lectures on Data Management 5 (5) (2013) 1–124.

[4] R. J. Bayardo, Y. Ma, R. Srikant, Scaling up all pairs similarity search, in: Proc. of the 16th international conference on World Wide Web, 2007, pp. 131–140.

[5] C. Xiao, W. Wang, X. Lin, J. X. Yu, G. Wang, Efficient similarity joins for near-duplicate detection, ACM Transactions on Database Systems (TODS) 36 (3) (2011) 1–41.

[6] P. Bouros, S. Ge, N. Mamoulis, Spatio-textual similarity joins, Proc. of the VLDB Endowment 6 (1) (2012) 1–12.

[7] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering? an adaptive framework for similarity join and search, in: Proc. of the 2012 ACM SIGMOD Int.Conference on Management of Data, 2012, pp. 85–96.

[8] D. Deng, G. Li, H. Wen, J. Feng, An efficient partition based method for exact set similarity joins, Proceedings of the VLDB Endowment 9 (4) (2015) 360–371.

[9] W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, Proceedings of the VLDB Endowment 9 (9) (2016) 636–647.

[10] F. Fier, N. Augsten, P. Bouros, U. Leser, J.-C. Freytag, Set similarity joins on mapreduce: an experimental survey, Proceedings of the VLDB Endowment 11 (10) (2018) 1110–1122.

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

[12] E. F. Sandes, G. L. Teodoro, A. C. Melo, Bitmap filter: Speeding up exact set similarity joins with bitwise operations, Information Systems 88 (2020) 101449.

[13] X. Wang, L. Qin, X. Lin, Y. Zhang, L. Chang, Leveraging set relations in exact set similarity join, Proceedings of the VLDB Endowment (2017).

[14] C. Bellas, A. Gounaris, Exact set similarity joins for large datasets in the gpgpu paradigm, in: Proc. of the 15th International Workshop on Data Management on New Hardware, DaMoN'19, ACM, New York, NY, USA, 2019, pp. 5:1–5:10.

[15] 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.

[16] R. Vernica, M. J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: SIGMOD Conference, 2010, pp. 495–506.

[17] F. Fier, T. Wang, E. Zhu, J.-C. Freytag, Parallelizing filter-verification based exact set similarity joins on multicores, in: International Conference on Similarity Search and Applications, Springer, 2020, pp. 62–75.

[18] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, K. J. Barker, Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect, IEEE Transactions on Parallel and Distributed Systems 31 (1) (2019) 94–110.

[19] C. Lutz, S. Breß, S. Zeuch, T. Rabl, V. Markl, Pump up the volume: Processing large data on gpus with fast interconnects, in: Proc. of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 1633–1649.

[20] X. Xie, W. Tan, L. L. Fong, Y. Liang, Cumf_sgd: Parallelized stochastic gradient descent for matrix factorization on gpus, in: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, 2017, pp. 79–92.

[21] A. Shanbhag, S. Madden, X. Yu, A study of the fundamental performance characteristics of gpus and cpus for database analytics, in: Proceedings of the 2020 ACM SIGMOD international conference on Management of data, 2020, pp. 1617–1632.

[22] D. Deng, Y. Tao, G. Li, Overlap set similarity joins with theoretical guarantees, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 905–920.

[23] J. Yang, W. Zhang, X. Wang, Y. Zhang, X. Lin, Distributed streaming set similarity join, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 565–576.

[24] 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.

[25] P. Li, A. Owen, C.-H. Zhang, One permutation hashing, in: Advances in Neural Information Processing Systems, 2012, pp. 3113–3121.

[26] J. Ji, J. Li, S. Yan, Q. Tian, B. Zhang, Min-max hash for jaccard similarity, in: 2013 IEEE 13th International Conference on Data Mining, IEEE, 2013, pp. 301–309.

[27] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. Lui, X. Guan, A memory-efficient sketch method for estimating high similarities in streaming sets, in: ACM SIGKDD, 2019, pp. 25–33.