Fast Parallel Set Similarity Joins on Many-core Architectures

Sidney Ribeiro-Junior, Rafael David Quirino, Leonardo Andrade Ribeiro, Wellington Santos Martins

Universidade Federal de Goiás, Brazil
{sydneyribeirojunior, rafaelquirino, laribeiro, wellington}@inf.ufg.br

Abstract. Set similarity join is a core operation for text data integration, cleaning, and mining. Previous research work on improving the performance of set similarity joins mostly focused on sequential, CPU-based algorithms. Main optimizations of such algorithms exploit high threshold values and the underlying data characteristics to derive efficient filters. In this article, we investigate strategies to accelerate set similarity join by exploiting massive parallelism available in modern Graphics Processing Units (GPUs). We develop two new parallel set similarity join algorithms, which implement an inverted index on the GPU to quickly identify similar sets. The first algorithm, called `gSSJoin`, does not rely on any filtering scheme and, thus, exhibits much better robustness to variations of threshold values and data distributions. Moreover, `gSSJoin` adopts a load balancing strategy to evenly distribute the similarity calculations among the GPU's threads. The second algorithm, called `sf-gSSJoin`, applies a block division scheme for dealing with large datasets that do not fit in GPU memory. This scheme also enables a substantial reduction of the comparison space by discarding entire blocks based on size limits. We present variants of both algorithms for a multi-GPU platform to further exploit task parallelism in addition to data parallelism. Experimental evaluation on real-world datasets shows that we obtain up to 109x and 19.5x speedups over state-of-the-art algorithms for CPU and GPU, respectively.

Categories and Subject Descriptors: H.2 [**Database Management**]: Systems; C.1 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

Keywords: Advanced Query Processing, GPU, High Performance Computing, Parallel Set Similarity Join

1. INTRODUCTION

The problem of efficiently answering set similarity join queries has attracted growing attention over the years [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006; Arasu et al. 2006; Bayardo et al. 2007; Vernica et al. 2010; Xiao et al. 2011; Ribeiro and Härder 2011; Wang et al. 2012; Deng et al. 2015; Cruz et al. 2016; Mann et al. 2016; Rong et al. 2017]. Set similarity join returns all pairs of similar sets from a dataset — two sets are considered similar if the value returned by a set similarity function for them is not less than a given threshold. This operation is of great interest and practical importance both in itself and as a basic operator for more advanced data processing tasks. Indeed, set similarity join has been used in a wide range of application domains including data integration [Doan et al. 2012], entity extraction [Agrawal et al. 2008], data cleaning [Chaudhuri et al. 2006], collaborative filtering [Spertus et al. 2005], and data mining [Leskovec et al. 2014].

Set similarity join is a popular approach to dealing with text data, whose representation is typically sparse and high-dimensional. Text data can be mapped to sets and there is a rich variety of set similarity functions to capture various notions of similarity — the well-known Jaccard similarity is a prime example of such functions. Furthermore, predicates involving set similarity functions can be equivalently expressed as an overlap constraint [Chaudhuri et al. 2006]; intuitively, the overlap of similar sets should be high. As a result, set similarity join is reduced to the problem of identifying set pairs with enough overlap.

This research was partially supported by CAPES, CNPq, FAPEG, and FINEP. We thank NVIDIA for equipment donations.

The set overlap abstraction provides the basis for several optimizations. *Prefix filtering* [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006] is arguably the most important of such optimizations. In fact, this technique is employed by all state-of-the-art algorithms considered in a recent experimental evaluation [Mann et al. 2016]. Prefix filtering exploits the threshold value to prune dissimilar set pairs by inspecting only a fraction of them. A filtering-verification framework is typically used in this context, where input sets are *sequentially* processed: in the filtering step, prefix filtering is employed with the support of an inverted index to discard set pairs that cannot meet the overlap constraint; then, in the verification step, the overlap of the surviving pairs is measured and those deemed similar are sent to the output.

Unfortunately, prefix filtering is only effective at high threshold values. Its pruning power drops drastically at lower thresholds, leading to an explosion of the number of set pairs that need to be compared in the verification step. This serious drawback may render set similarity join algorithms based on prefix filtering unsuitable for applications which often require lower thresholds to produce accurate results — important examples are duplicate detection and clustering [Hassanzadeh et al. 2009]. Moreover, filtering effectiveness and, in turn, algorithm performance heavily relies on characteristics of the underlying data distribution [Sidney et al. 2015]. In particular, the filtering effect is severely reduced (or even eliminated) on more uniform data distributions.

To address the challenge of efficiently processing set similarity joins, we can exploit parallelism and take advantage of modern architectures with multiple cores. The multi-core trend affected not only CPUs but also GPUs (graphics processing units), which are known as many-core processors or accelerators. GPUs offer attractive performance to energy consumption and are becoming increasingly more affordable due to mass marketing for gaming. They have a large number of relatively simple, and slower, processing cores. The possibility of using GPUs to perform general purpose computations appeared a few years ago and quickly caught the attention of researchers. However, due to their significant differences when compared to CPUs, novel algorithms and implementation approaches need to be designed. In order to take full advantage of their performance, it is necessary considerable parallelism (tens of thousands of threads) and an adequate control of high latency operations, particularly involving global memory accesses which should be reduced to a minimum.

In this article, we present an exact parallel algorithm and a GPU-based implementation for the set similarity join problem. Our solution takes advantage of data parallelism by processing individual sets in parallel. Each individual operation can be seen as a set similarity search since it finds all sets in a set collection that are similar to a given input set. Furthermore, elements in a set are, in turn, processed in parallel as well, thereby exploiting both inter- and intra-set parallelism. This greatly improves the similarity join processing and can be straightforwardly mapped to modern highly-threaded accelerators like many-core GPUs. The proposed solution, called `gSSJoin` (GPU-based Set Similarity Join), efficiently implements an inverted index, by using a parallel counting operation followed by a parallel prefix-sum calculation. At search time, this inverted index is used to quickly find sets sharing tokens with the input set. We construct an index for the input set which is used for a load balancing strategy to evenly distribute the similarity calculations among the GPU's threads. Finally, set pairs whose similarity is not less than the given threshold are returned to the CPU.

This article is an extended and revised version of a previous conference paper [Ribeiro-Júnior et al. 2016]. As part of the new material, we include a new, improved version of the gSSJoin algorithm, called `sf-gSSJoin` (Size-filtered gSSJoin), which uses size filtering to discard dissimilar pairs before comparing the sets, block division to deal with larger datasets, and a simpler compaction technique to improve performance at high threshold values. Furthermore, we provide an extensive set of experiments in which we compare our algorithm with other parallel solutions on several datasets.

Our contributions are summarized as follows.

—A fine-grained parallel algorithm for both data indexing and set similarity join processing.

—A highly threaded GPU implementation that takes advantage of intensive occupation, hierarchical memory, load balancing, and coalesced memory access.

—A scalable multi-GPU implementation that exploits both data parallelism and task parallelism.

—A block division scheme for dealing with larger datasets that do not fit in the GPU memory, which also enables an aggressive reduction of the comparison space by pruning entire blocks of sets based on size limits.

—An extensive experimental evaluation comparing our solutions with state-of-the-art sequential as well as parallel algorithms on standard real-world textual datasets.

This article is organized as follows. Section 2 covers related work. Section 3 provides background material and Section 4 presents an overview of the architecture and programming model of a GPU. The `gSSJoin` algorithm and its improved version, `sf-gSSJoin`, are presented in Section 5 and Section 6, respectively. Experimental results are described in Section 7 and Section 8 concludes the article.

2. RELATED WORK

There is a substantial body of literature on the efficient computation of set similarity joins. Most proposals focus on the design of CPU-based, intrinsically sequential algorithms [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006; Arasu et al. 2006; Bayardo et al. 2007; Xiao et al. 2011; Ribeiro and Härder 2011; Wang et al. 2012] and, therefore, cannot fully exploit modern many-core architectures. Main optimizations use the threshold and the underlying data characteristics to derive effective filters [Sidney et al. 2015]. Popular filtering schemes are derived from the set overlap constraint, such as size filtering [Sarawagi and Kirpal 2004], prefix filtering [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006], and positional filtering [Xiao et al. 2011]. Other approaches partition the input dataset based on the symmetric set difference such that two sets are similar only if they share a common partition [Arasu et al. 2006; Deng et al. 2015]. Interestingly, overly complex filters can actually increase execution runtime: PPJoin+ [Xiao et al. 2011] and AdaptJoin [Wang et al. 2012], which employ sophisticated filtering strategies, have been shown to be the slowest competitors in the study presented by Mann et al. [2016]. Our proposed solutions either use no filter (i.e., `gSSJoin`) or only size filtering to reduce the comparison space (i.e., `sf-gSSJoin`).

Another popular type of string similarity join employs constraints based on the edit distance, which is defined by the minimum number of character-editing operations (i.e., insertion, deletion, and substitution) to make two strings equal. Because one can derive (q-gram) set overlap bound to the edit distance [Gravano et al. 2001], our algorithms can be used in a pre-processing step to reduce the number of distance computations — edit distance also lends itself to efficient GPU implementation [Chacón et al. 2014].

Lieberman et al. [Lieberman et al. 2008] presented a parallel similarity join algorithm for distance functions of the Minkowski family (e.g., the Euclidean distance). The algorithm first maps the smaller input dataset to a set of space-filling curves and then performs interval searches for each point in the other dataset in parallel. The overall performance of the algorithm drastically decreases as the number of dimensions increases (see Figure 5b in [Lieberman et al. 2008]) because every additional dimension requires the construction of a new space-filling curve. Thus, this approach can be prohibitively expensive on text data, whose representation typically involves several thousands of dimensions.

Besides stand-alone algorithms, set similarity join can also be realized using relational database technology. Previous work proposed expressing set similarity join declaratively in SQL [Ribeiro et al. 2016] or implementing it as a physical operator within the query engine [Chaudhuri et al. 2006]. Considering the increasing adoption of hardware accelerators by relational databases, including commercial products [Meraji et al. 2016], set similarity join can be a candidate operation for off-loading to the GPU in such systems.

Other works focused on performing set similarity join in a distributed platform such as MapReduce [Vernica et al. 2010; Deng et al. 2014; Rong et al. 2017]. In general, these proposals apply a partition scheme to send dissimilar strings to different *reducers* and, thus, avoid unnecessary similarity calculations. Existing partition schemes are based on prefix filtering [Vernica et al. 2010], symmetric set difference [Deng et al. 2014], and vertical partitioning [Rong et al. 2017]. Nevertheless, intensive computations still have to be performed in the reducers. We plan to investigate the integration of our solutions into a distributed setting to accelerate such local computations in future work.

Our algorithms, as well as all other algorithms discussed so far, are *exact*, i.e., they always produce the correct answer. *Approximate* set similarity joins may miss valid output set pairs to trade accuracy for query time. Locality Sensitive Hashing (LSH) is the most popular technique for approximate set similarity joins [Indyk and Motwani 1998], which is based on hashing functions that are approximately similarity-preserving. Cruz et al. [2016] propose an approximate set similarity join algorithm designed for a many-core architecture (GPU). The authors estimate the Jaccard similarity between two sets using MinHash [Broder et al. 1998], an LSH scheme for Jaccard. MinHash can be orthogonally combined with our algorithm to reduce set sizes and, thus, obtain greater scalability.

Similarity search queries retrieve strings similar to a given query string. They could be treated as a special case of join queries, where one of the two input datasets contains a single string. Techniques to increase filtering effectiveness in similarity search queries include exploiting multiple set orderings [Kim and Lee 2012], extracting different signature schemes from strings in the input data and the query string [Qin et al.], and defining a global order to import the input sets into a tree index structure [Zhang et al. 2017]. The work in [Matsumoto and Yiu 2015] proposes a top-k similarity search algorithm for data represented as points in Euclidean space — top-k similarity query returns the k most similar data objects. This algorithm leverages both the GPU and CPU: expensive distance calculations are executed on the GPU, whereas sorting is executed on the host CPU.

To the best of our knowledge, our `gSSJoin` algorithm was the first GPU-based algorithm for exact set similarity join [Ribeiro-Júnior et al. 2016]. In a subsequent work, we proposed `fgssjoin` [Quirino et al. 2017], which employs several filters to reduce the number of candidates. Specifically, `fgssjoin` applies prefix filtering, size filtering, and positional filtering to prune dissimilar pairs of sets in the filtering phase. These filters are very effective on high threshold values, however, its performance declines as lower threshold values are used. Size filtering is also exploited by `fgssjoin` in a block division scheme similar to the one adopted by `sf-gSSJoin`. We experimentally compare the algorithms presented in this article with `fgssjoin` in Section 7.

3. BACKGROUND

In this section, we first formally define the set similarity join problem. Then, we review state-of-the-art optimization techniques and describe a high-level algorithmic framework for set similarity joins. Finally, we discuss the limitations of current solutions.

3.1 Basic Concepts and Problem Definition

Strings can be mapped to *sets of tokens* in several ways. A well-known method is based on the concept of *q-grams*, i.e., sub-strings of length q obtained by "sliding" a window over the characters of the input string. For example, the string *"gSSJoin"* can be mapped to the set of *3*-grams tokens {*'gSS'*, *'SSJ'*, *'SJo'*, *'Joi'*, *'oin'*}.

Given two sets r and s, a set similarity function $sim(r, s)$ returns a value in $[0, 1]$ to represent their similarity; a greater value indicates that r and s have higher similarity. We formally define the set similarity join operation as follows.

Algorithm 1: General set similarity join algorithm.

```
   Input: A sorted set collection C, a threshold τ
   Output: A set S containing all pairs (r, s) s.t. sim(r, s) ≥ τ
1  I_1, I_2, ... I_|U| ← ∅, S ← ∅
2  foreach r ∈ C do
3      foreach t ∈ pref_β(r) do
4          foreach s ∈ I_t do
5              if not filter(r, s)
6                  S ← S ∪ verify(r, s)
7          I_t ← I_t ∪ {r}
8  return S
```

Definition 3.1 *Set Similarity Join.* Given two set collections $\mathcal{R}$ and $\mathcal{S}$, a set similarity function *sim*, and a threshold τ, the *set similarity join* between $\mathcal{R}$ and $\mathcal{S}$ returns all scored set pairs $\langle (r,s), \tau\prime \rangle$ s.t. $(r,s) \in \mathcal{R} \times \mathcal{S}$ and $sim(r,s) = \tau\prime \geq \tau$.

A popular set similarity function is the well-known *Jaccard similarity*. Given two sets r and s, the Jaccard similarity is defined as $J(r,s) = \frac{|r \cap s|}{|r \cup s|}$. A predicate involving the Jaccard similarity and a threshold τ can be equivalently rewritten into a set overlap constraint: $J(r,s) \geq \tau \iff |r \cap s| \geq \frac{\tau}{1+\tau}(|r| + |s|)$. In this article, we focus on the Jaccard similarity, but all concepts and techniques presented henceforth holds for other set similarity functions as well such as Dice and Cosine [Ribeiro and Härder 2011].

Intuitively, the size difference of similar sets should not be large. We can derive bounds for immediate pruning of candidate pairs whose sizes differ enough. Formally, the size bounds of r, denoted by $minsize(r,\tau)$ and $maxsize(r,\tau)$, are functions that map τ and $|r|$ to a real value s.t. $\forall s$, if $sim(r,s) \geq \tau$, then $minsize(r,\tau) \leq |s| \leq maxsize(r,\tau)$ [Sarawagi and Kirpal 2004]. Thus, given a set r, we can define a size-based filter to safely ignore all other sets whose sizes do not fall within the interval $[minsize(r,\tau), maxsize(r,\tau)]$. For the Jaccard similarity, we have $[minsize(r,\tau), maxsize(r,\tau)] = \left[\tau \times |r|, \frac{|r|}{\tau}\right]$.

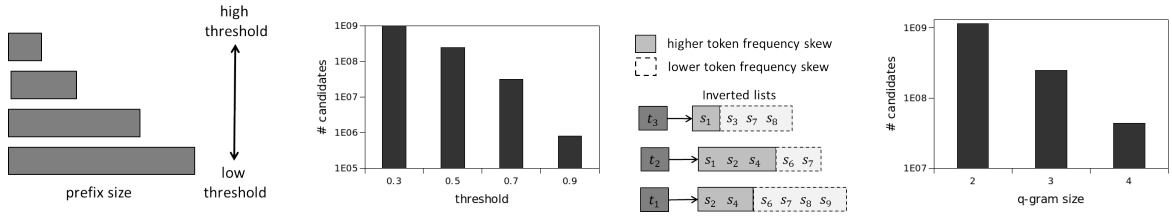
Further, we can use the *prefix filtering principle* [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006] to prune dissimilar sets by examining only a subset of them. We first assume that the tokens of all sets are sorted according to a total order. A prefix $r_p \subseteq r$ is the subset of r containing its first p tokens. Given two sets r and s, if $|r \cap s| \geq \alpha$, then $r_{(|r|-\alpha+1)} \cap s_{(|s|-\alpha+1)} \neq \varnothing$. For a threshold τ, we can identify all candidate matches of a given set r using a prefix of length $|r| - \lceil |r| \cdot \tau \rceil + 1$; we denote such prefix by $pref(r)$.

The original overlap constraint only needs to be verified on set pairs sharing a prefix token. Finally, we pick the token frequency ordering as total order, thereby sorting the sets by increasing token frequencies in the set collection. Thus, we move lower frequency tokens to prefix positions to minimize the number of verifications.

Example 3.1. *Consider the sets* $r = \{\boldsymbol{A}, \boldsymbol{B}, C, \boldsymbol{D}, \boldsymbol{E}\}$ *and* $s = \{\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}, \boldsymbol{E}, F\}$. *Thus, we have* $|r| = |s| = 5$ *and* $|r \cap s| = 4$*; thus* $sim(r,s) = \frac{4}{6} \approx 0.66$. *For a threshold* $\tau = 0.6$*, the set overlap constraint is* $|r \cap s| \geq \frac{0.6}{1+0.6}(5+5) = 3.75$, $[minsize(r,\tau), maxsize(r,\tau)] = [3, 8.3]$ *(values for s are the same),* $pref(r) = \{A, B, C\}$*, and* $pref(s) = \{A, B, D\}$.

3.2 General Algorithm

Most current set similarity join algorithms for main memory employ an inverted index and follow a filtering-verification framework [Mann et al. 2016]. A high-level description of this framework is presented by Algorithm 1. An inverted list I_t stores all sets containing a token t in their prefix. The input collection C is scanned and, for each set r, its prefix tokens are used to find candidate sets in the



(a) Prefix size varying threshold values.

(b) Number of candidates generated with varying threshold values.

(c) Inverted list length with different token frequency distributions.

(d) Number of candidates generated with varying q values.

Fig. 1. Limitations of algorithms based on prefix filtering.

corresponding inverted lists (lines 2–4). Each candidate s is checked using filters, such as positional [Xiao et al. 2011] and length-based filtering [Sarawagi and Kirpal 2004] (line 5); if the candidate passes through, the actual similarity computation is performed in the verification step and r and s are added to the result if they are similar (line 6). Finally, r is appended to the inverted lists of its prefix tokens (line 7). An important observation is that Algorithm 1 is intrinsically sequential: sets, prefix tokens, and candidate sets are processed sequentially, while the inverted index is dynamically created.

Algorithm 1 is actually a self-join. It can be extended to a binary join by first indexing the smaller collection and then going through the larger collection to identify matching pairs.

3.3 Prefix Filtering Limitations

Currently, prefix filtering is the prevalent optimization technique for CPU-based, set similarity join algorithms. As such, all these algorithms benefit from its pruning power, but also suffer from its limitations. In particular, prefix filtering effectiveness is heavily dictated by two factors: threshold value and token frequency distribution [Sidney et al. 2015].

There is a clear correlation between threshold values and similarity join performance. Invariably, execution time increases with lower threshold values. The explanation is that lower threshold values imply larger prefixes as illustrated in Figure 1(a). As a result, more inverted lists have to be scanned (Algorithm 1, lines 3–4) and a larger number of candidate pairs have to be verified. Figure 1(b) shows the number of candidates (in log scale) for decreasing Jaccard thresholds on a 100K sample taken from the DBLP dataset (details about the datasets are given in Section 7). As we decrease the threshold from 0.9 to 0.3, there is an increase of three orders of magnitude in the number of candidates.

Disregarding pruning due to other filters, the frequency of tokens in the prefix determines the number of candidates for a given set. The total token order based on frequency places rare tokens in the prefixes. As a result, inverted lists are shorter and there is much less prefix overlap between dissimilar sets. Of course, the effectiveness of this strategy depends on the underlying token frequency distribution. For a uniform distribution, it behaves not better than an ordinary lexicographical order. Figure 1(c) depicts the impact of the token frequency skew on size of the lists associated with prefix tokens and 1(d) shows the number of candidates (in log scale) for increasing values q-gram size (larger values of q results in more skewed token distribution). We can observe an increase of one order of magnitude in the number of candidates.

The above observations indicate intrinsic limitations of prefix filtering, and, in turn, current set similarity join algorithms. Next, we provide a general overview of the underlying many-core architecture before presenting our massively parallel approach, which avoids such drawbacks.

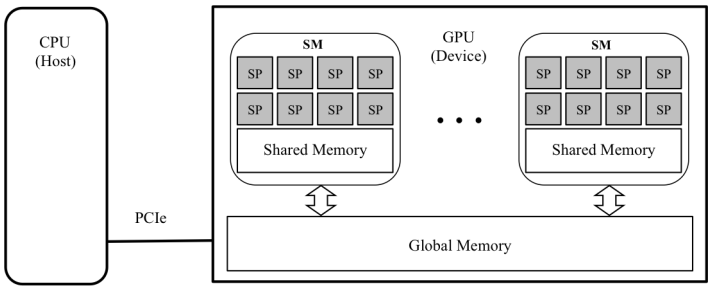


Fig. 2. *Overview of a GPU architecture*

4. GPU ARCHITECTURE AND PROGRAMMING MODEL

This section provides a brief description of both the GPU architecture and its programming model (we refer to [Kirk and Wen-mei 2012] for details).

GPUs were originally designed as special-purpose coprocessors for dedicated graphics rendering. However, in the last years, it has become a powerful accelerator for general purpose computing. Nowadays GPUs are regarded as massively parallel processors with approximately ten times the computation power and memory bandwidth of CPUs. They can perform a large number of independent parallel computations and this performance is improving at a rate higher than that of CPUs and at an exceptionally high performance-to-cost ratio.

Figure 2 shows a simplified diagram of a GPU that is common to most GPUs currently available. A GPU can be regarded as a Multiple SIMD (Single Instruction Multiple Data) processor with each SIMD unit corresponding to a streaming multiprocessor (SM). Inside the SM there are several streaming processor (SP) cores that operate on different data but in a synchronous way. SPs belonging to the same SM share, among others, instruction fetch and decoding, load and store units, the register file and local (shared) memory. A group of threads in an SM is divided into multiple schedule units, called *warps*, that are dynamically scheduled on the SM.

The GPU supports a large number of light-weight threads, e.g. often tens or hundreds of thousands, and, unlike CPU threads, the overhead of creation and switching is negligible. Threads for which data have already been transferred from the global memory can perform calculations while threads waiting for the completion of data transfer have to wait. Thus, in order to hide the global memory's high latency, it is important to have more threads than the number of SPs and to have threads accessing consecutive memory addresses that can be coalesced.

In most platforms, the communication between CPU and GPU takes place through a PCI Express connection. This channel permits the exchange of data between each one's address space but at a much slower speed. Thus, the GPU programming model requires that part of the program runs on the CPU while the computationally-intensive part is processed by the GPU. A GPU program exposes parallelism through data-parallel functions, called kernels, that are offloaded to the GPU. The programmer needs to configure the number of threads to be used. These threads are organized in groups (thread blocks) that are further assembled into a grid structure. When a kernel is launched, the blocks within a grid are distributed on idle SMs and the threads mapped to the SPs.

5. THE GSSJOIN ALGORITHM

In this section, we present a detailed description of our GPU-based algorithm called `gSSJoin`. We start by describing the inverted index creation, that is, the pre-processing step. Then, the actual set similarity join is detailed. Finally, we present the multi-GPU version of `gSSJoin`.

5.1 Pre-processing

Before performing the set similarity join, we map input strings to sets of tokens using q-grams and create an inverted index in the GPU memory, assuming the input collection fits in memory and is static. Let $\mathcal{S}$ be the set collection ordered by set size and $\mathcal{V}$ be the vocabulary of the collection, that is, the set of distinct tokens of the input collection. The input collection is the set $\mathcal{E}$ of distinct token-sets (t, s) pairs occurring in the original collection, with $t \in \mathcal{V}$ and $s \in \mathcal{S}$. An array of size $|\mathcal{E}|$ is used to store the inverted index. Once the set $\mathcal{E}$ has been moved to the GPU memory, each pair in it is examined in parallel, so that each time a token is visited the number of sets where it appears is incremented and stored in the array *count* of size $|\mathcal{V}|$. A parallel prefix-sum is executed on the *count* array by mapping each element to the sum of all tokens before it and storing the results in the *index* array. Thus, each element of the *index* array points to the position of the corresponding first element in the $invertedIndex$, where all (t, s) pairs will be stored sorted by the token. The cardinality of all sets is computed in parallel with each processor contributing partially (incrementing) in the respective position of the array *cardinality* of size $|\mathcal{S}|$. Algorithm 2 depicts the data indexing process.

5.2 Set Similarity Search

After the inverted index construction, gSSJoin can be viewed as a batch of set similarity search operations. Given an input set, the set similarity search consists of two steps. First, the Jaccard similarity of the input set s to all sets have to be computed. Then, the sets similar to the input set are selected. The Jaccard similarity computation takes advantage of the inverted index model, because only the similarities between the input set s and those sets in $\mathcal{S}$ that have tokens in common with s have to be computed. These sets are the elements of the $invertedIndex$ pointed to by the entries of the *index* array associated with the tokens occurring in the input set s.

A straightforward solution to compute the Jaccard similarity is to distribute the tokens of the input set s evenly among the processors and let each processor p access the inverted lists corresponding to tokens allocated to it. However, the distribution of tokens in sets of the collections follows approximately the Zipf's Law [Zipf and Wilson 1949]. This means that few tokens occur in a large number of sets and most tokens occur in only a few sets. Consequently, the size of the inverted lists also varies according to the Zipf's Law, thus distributing the workload according to the tokens of s would cause a great imbalance of the work among the processors.

Thus, we propose a load balance method to distribute the sets evenly among the processors so that each processor computes approximately the same number of Jaccard similarities. In order to facilitate the explanation of this method, suppose that we concatenate all the inverted lists corresponding to tokens in s in a logical vector $E_s = [\,0\,..\,|E_s|-1\,]$, where $|E_s|$ is the sum of the sizes of all inverted lists

Algorithm 2: Indexing

input : The entries array $\mathcal{E}$ where each element is a pair $(token, setId)$
output: The inverted index I

```
1  array of integers count[0...|V| − 1]
2  array of integers index[0...|V| − 1]
3  inverted index I[0...|E| − 1]
4  for each entry e in E, in parallel do
5  |  atomicAdd(count[e.token], 1)
6  end
7  Perform an exclusive sum on count in parallel and store the result in index
8  for each entry e in E, in parallel do
9  |  position ← atomicAdd(index[e.token], 1)
10 |  I[position] ← e.setId
11 end
12 return I, count and index
```

of tokens in s. Given a set of processors $\mathcal{P} = \{p_0, \cdots p_{|\mathcal{P}|-1}\}$, the load balance method should allocate elements of E_s in intervals of approximately the same size, that is, each processor $p_i \in \mathcal{P}$ should process elements of E_s in the interval $[i\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil - 1, |E_s| - 1)]$. Since each processor knows the interval of the indexes of the logical vector E_s it has to process, all that is necessary to execute the load balancing is a mapping of the logical indexes of E_s to the appropriate indexes in the inverted index (array $invertedIndex$). Each processor executes the mapping for the indexes in the interval corresponding to it and finds the corresponding elements in the $invertedIndex$ array for which it has to compute the Jaccard similarity to the input set.

Let $\mathcal{V}_s \subset \mathcal{V}$ be the vocabulary of the input set s. The mapping uses three auxiliary arrays: $count_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$, $start_s[\,0\,..\,|\mathcal{V}_s| - 1\,]]$ and $index_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$. The arrays $count_s$ and $start_s$ are obtained together by copying in parallel $count[t_i]$ to $count_s[t_i]$ and $index[t_i]$ to $start_s[t_i]$, respectively, for each token t_i in the input set s. Once the $count_s$ is obtained, an inclusive parallel prefix sum on $count_s$ is performed and the results are stored in $index_s$.

Algorithm 3 shows the pseudo-code for the complete similarity search. In lines 3–6, the arrays $count_s$ and $start_s$ are obtained. In line 8, the array $index_s$ is obtained by applying a parallel prefix sum on array $count_s$. Next, each processor executes a mapping of each position x in the interval of indexes of E_s associated to it to the appropriate position of the $invertedIndex$. This mapping is described in lines 10-17 of the algorithm. Then, the mapped entries of the inverted index are used to compute the intersection between each set associated with these entries and the input set. The intersections are computed partially by each processor, but the complete intersections are available when all processors have finished this phase. Lines 20-24 show how the Jaccard similarities are computed and compacted. Each processor is responsible for $\frac{|S|}{|\mathcal{P}|}$ similarities. Each Jaccard similarity calculation uses the intersection (calculated in the previous phase) and the *cardinality* of the sets. Similarities above the given *threshold* are flagged and pruned by an exclusive parallel prefix sum. Thus only those similarities are returned to the CPU (line 22).

Algorithm 3: $SimilaritySearch(invertedIndex, s)$

input : $invertedIndex$, $count$, $index$, $cardinality$, $threshold$, $inputset$ $s[\,0\,..\,|\mathcal{V}_s| - 1\,]$.
output: Jaccard similarity array $jac\_sim[\,0\,..\,|\mathcal{S}| - 1\,]$ initialized with zeros.

1 array of integers $count_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$ initialized with zeros
2 array of integers $index_s[\,0\,..\,|\mathcal{V}_s| - 1\,]$
3 **for** *each token* $t_i \in s$, *in parallel* **do**
4 $\quad count_s[i] = count[t_i]$;
5 $\quad index_s[i] = index[t_i]$;
6 **end**
7 Perform an inclusive parallel prefix sum on $count_s$ and stores the results in $index_s$
8 **foreach** *processor* $p_i \in \mathcal{P}$ **do**
9 $\quad$ **for** $x \in [i\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|E_s|}{|\mathcal{P}|}\rceil - 1, |E_S| - 1)]$ **do**
$\quad\quad$ `// Map` x `to the correct position` $indInvPos$ `of the` $invertedIndex$
10 $\quad\quad pos = min(i : index_s[i] > x)$;
11 $\quad\quad$ **if** $pos = 0$ **then**
12 $\quad\quad\quad p = 0$; $offset = x$;
13 $\quad\quad$ **else**
14 $\quad\quad\quad p = index_s[pos - 1]$; $offset = x - p$;
15 $\quad\quad$ **end**
16 $\quad\quad indInvPos = start_s[pos] + offset$
17 $\quad\quad$ uses $s[pos]$ and $invertedIndex[indInvPos]$ in the partial computation of the intersection between s and the set associated to $invertedIndex[indInvPos]$
18 $\quad$ **end**
19 $\quad$ **for** $x \in [i\lceil\frac{|S|}{|\mathcal{P}|}\rceil, min((i+1)\lceil\frac{|S|}{|\mathcal{P}|}\rceil - 1, |S| - 1)]$ **do**
$\quad\quad$ `// Jaccard similarity calculation for` $\mathcal{S}$ `sets using` $\mathcal{P}$ `processors`
20 $\quad\quad$ uses intersection and union (through $cardinality$) to compute the Jaccard similarity
21 $\quad\quad$ flag sets with Jaccard similarity above the $threshold$
22 $\quad$ **end**
23 **end**
24 Perform an exclusive parallel prefix sum on the flagged sets to compact the selected sets
25 **Return** the array: jac_sim with the selected jaccard similarities.

5.3 Multi-GPU Similarity Join

The `gSSJoin` algorithm was designed having in mind a many-core architecture (accelerator) with (global) shared memory. It exploits data parallelism when processing a single input set and uses a single accelerator. It does that by making use of thousands of threads to index the dataset and to find the most similar sets to a given input set. However, when dealing with self-joins, the `gSSJoin` search has to be invoked repeatedly, to deal with the processing of many input sets. The `gSSJoin` algorithm can handle that by processing the queries one after another, once the input data has been indexed. This streaming operation requires that the most similar sets are returned before another input set can be processed. In addition, a set-specific memory allocation is needed for every set. Moreover, machines with more than one accelerator (GPU) can not take advantage of the extra computing power for the `gSSJoin` search. These observations have motivated us to extend the `gSSJoin` to deal with multiple sets in a multi-GPU platform.

In the multi-GPU version, called `mgSSJoin`, task parallelism is exploited in addition to data parallelism. The data indexing step is performed by replicating the input data in each of the g available GPUs and then, in parallel, creating g copies of the inverted index. Thus, each GPU receives the same task and they all produce the same inverted index in their memory. Next, the `gSSJoin` search proceeds by partitioning the m sets (tasks) into the g GPUs. Each GPU receives m/g tasks. This is possible since the tasks (sets) are completely independent of each other. Since the sets are of different size, we preallocate memory based on the biggest set, i.e., the set with the largest number of tokens). This saves us a lot of time since GPU memory allocation can be very costly. Algorithm 4 shows the pseudo-code for the `mgSSJoin`. Note that this algorithm, differently from the previous ones, exploits task parallelism and runs on the CPU. The GPU function (kernel) calls (invocations), for data indexing and `gSSJoin` search, take place in lines 3 and 9 respectively.

6. THE SF-GSSJOIN ALGORITHM

In this section, we present `sf-gSSJoin`, an improved version of our `gSSJoin`, which includes a block division scheme for dealing with larger datasets that do not fit in the GPU memory. This division strategy also allows us to prune entire blocks based on size filtering. As the previous algorithm, `sf-gSSJoin` builds an inverted index to quickly identify sets containing tokens in common.

The `sf-gSSJoin` algorithm proceeds in two phases: indexing and probing. In the indexing phase, the sets are indexed in an inverted index. The similarity join happens in the probing phase when probe sets are compared with indexed sets and the similar pairs are returned. When dealing with datasets with results that do not fit in GPU's memory, `sf-gSSJoin` divides the data into blocks of sets and works with partial inverted indexes. The blocks are indexed successively and probed with all previous blocks. The blocks that were not indexed yet are not probed because it would be redundant

Algorithm 4: $MultiGPUSearch(E)$

input : token-set pairs in $E[0\,..\,|\mathcal{E}|-1]$.
output: A list of the most similar, one for each set.

```
1  for each i ∈ g, in parallel do
2  |   set.gpu.device(i);
3  |   DataIndexing(E);
4  end
5  Allocate memory for the biggest set;
6  for each j ∈ g, in parallel do
7  |   set.gpu.device(j);
8  |   for each set s ∈ (m/g) in parallel do
9  |   |   SimilaritySearch(invertedIndex, s);
10 |   end
11 end
12 Return A list of the most similar sets, one for each set.
```

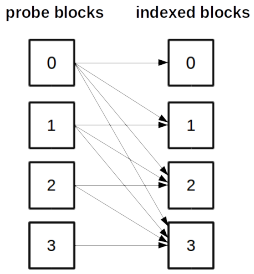


Fig. 3. The block division strategy

Algorithm 5: sf-gSSJoin

input : A collection of sets C, the size of blocks n and a threshold γ
output: A set S with all similar pairs (x, y)

1 Divide C in blocks of size n
2 **for** *each set block* i *in* C **do**
3 $\quad I \leftarrow indexing(C[i * n...min(i * n + n - 1, |C| - 1)])$
4 $\quad$ **for** *each set block* j *in* C **do**
5 $\quad\quad$ **if** $j \leq i$ **and** $|C[min(j * n + n - 1, |C| - 1)]| \geq minsize(C[i * n], \gamma)$ **then**
6 $\quad\quad\quad S \leftarrow S \cup probing(C[j * n...min(j * n + n - 1, |C| - 1)], I, \gamma)$
7 $\quad\quad$ **end**
8 $\quad$ **end**
9 **end**
10 **return** S

work: due to the commutative property of Jaccard, if the pair (x, y) meets the threshold, so does (y, x). The block division strategy is shown in Figure 3 with a dataset divided into 4 blocks. The first block is probed only against itself and the last is probed against all blocks.

The size filtering technique is applied between blocks before the probing phase as well as between sets on the probing phase. When applied between blocks, `sf-gSSJoin` verifies if the last probe set is larger or equal to the `minsize` value of the first indexed set. If it is not, all the previous probe sets cannot be similar too because they are smaller than the last set and all the following indexed sets are also too large to be similar to any probing set. Thus, it is not necessary to probe the block.

The `sf-gSSJoin` algorithm is shown in Algorithm 5. The sets are divided into blocks of size n in line 1. Each block is indexed in line 3 and probed by the previous blocks and by itself in line 6. Then the partial result is included in the final result. The size pruning between blocks is done in line 4. The indexing phase is the same of `gSSJoin` and is described by Algorithm 2.

6.1 Probing

The probing phase achieves massive multi-level parallelism in the GPU by dividing the probe sets between the streaming multi-processors (SMs) and, inside each SM, dividing the inverted lists between the streaming processors (SPs). As can be seen in Algorithm 6, the sets are examined in parallel and the inverted lists related to their tokens are also examined in parallel (lines 2 and 4). Before incrementing the intersection value between the probe set and the sets in the inverted lists, the set id is verified to avoid unnecessary computations and the indexed set size is verified with the probe set *maxsize* (line 5). If the indexed set is smaller than the probe maximum size, they cannot be similar and the intersection increment is not done. This is the size filtering technique applied between sets. It is important to notice that sets that cannot be similar because of their sizes will keep the intersection value equal to 0. The intersection increment is done with an atomic function in line 6 because many threads can be accessing the same sets at the same time.

Algorithm 6: Probing

```
   input : The block of probe sets P, the partial inverted index I and a threshold τ
   output: The set of similar pairs S
 1 matrix of integers M_{|P|×|P|}
 2 for each set x in P, in parallel do
 3 |   for each feature f in x do
 4 |   |   for each set y in f's inverted list I[f], in parallel do
 5 |   |   |   if x.id < y.id and maxsize(x) > |y| then
 6 |   |   |   |   atomicAdd(M[x.id][y.id], 1)
 7 |   |   |   end
 8 |   |   end
 9 |   end
10 end
11 totalSimilars ← 0
12 for each pair (x, y) in M, in parallel do
13 |   if M[x.id][y.id] > 0 then
14 |   |   similarity ← calculateSimilarity(|x|, |y|, M[x.id][y.id])
15 |   |   if similarity ≥ τ then
16 |   |   |   position ← atomicAdd(totalSimilars, 1)
17 |   |   |   S[position] ← (x.id, y.id)
18 |   |   end
19 |   end
20 end
21 return S
```

The second part of the probing phase verifies in parallel the pairs with non zero intersection, in lines 12 and 13, calculates the similarity between them in line 14 and stores the pairs with similarity equal or greater than the chosen threshold in the result array in line 17. An atomic add function is used in line 16 to determine the position where the pair should be stored. Thus only similar pairs will be returned, reducing the amount of data to be copied back to the CPU memory. This data compaction, consequently, reduces the cost of memory copy.

6.2 Multi-GPU Version

The multi-GPU version of `sf-gSSJoin` divides the indexed blocks among the GPUs and sends the probing sets to the respective GPUs. The algorithm is the same as described in Section 5, except that indexed blocks and their probing blocks are divided among GPUs in line 2. This may cause a load imbalance because each indexed block has a different number of probing blocks. For example, if we have two GPUs and two blocks, one GPU will index the first block and probe the same block. The other GPU will index the second block and probe the first and second blocks. But if the number of blocks is big and the set sizes variable, this imbalance is generally negligible since some blocks will not be probed as a result of size pruning.

7. EXPERIMENTS

We now experimentally evaluate our algorithms. The goals of our study are: 1) to evaluate the performance of `gSSJoin` and `sf-gSSJoin` with varying thresholds on different datasets; 2) to compare our algorithms against state-of-the-art CPU-based and GPU-based similarity join algorithms; 3) to compare the performance gain of sf-`gSSJoin` against `gSSJoin`; 4) to measure the speedup gains of our multi-GPU implementation.

Table I. Dataset statistics.

Database	Number of sets	Max length	Mean length	Standard deviation
DBLP	100k	392	116.48	37.15
Spotify	500k	101	28.57	11.1
IMDb	300k	248	51.8	56.51

7.1 Experimental Setup

We used three publicly available, real-world datasets: DBLP [1], a database of Computer Science publications, Spotify[2], a database of songs names and artists, and IMDb[3], a database of movies, series and documentaries titles. For DBLP, we randomly selected 20k publications and extracted their title and authors; then, we generated 4 additional "dirty" copies of each string, i.e., duplicates to which we injected textual modifications consisting of 1–5 character-level modifications (insertions, deletions, and substitutions). For all datasets, we converted all strings to lower-case and eliminated repeated white spaces. Statistics about the datasets are shown in Table I. Finally, we tokenized all strings into sets of 2-grams and 3-grams and sorted the sets as described in Section 3.

We compared `sf-gSSJoin` against `ppjoin`, one of the best-performing CPU-based algorithms according to the evaluation in [Mann et al. 2016], `fgssjoin`, a GPU-based algorithm which showed significant speedups against `ppjoin` [Quirino et al. 2017],and `gSSJoin`, our earlier algorithm [Ribeiro-Júnior et al. 2016]. We used the binary provided by the authors of `ppjoin`[4]. All three parallel algorithms were implemented using the CUDA Toolkit version 7.5. The experiments were conducted on a machine running CentOs 7.2.1511 64-bits, with 2 Intel Xeon E5-2620, 16GB of ECC RAM, and four GeForce Zotac Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores each.

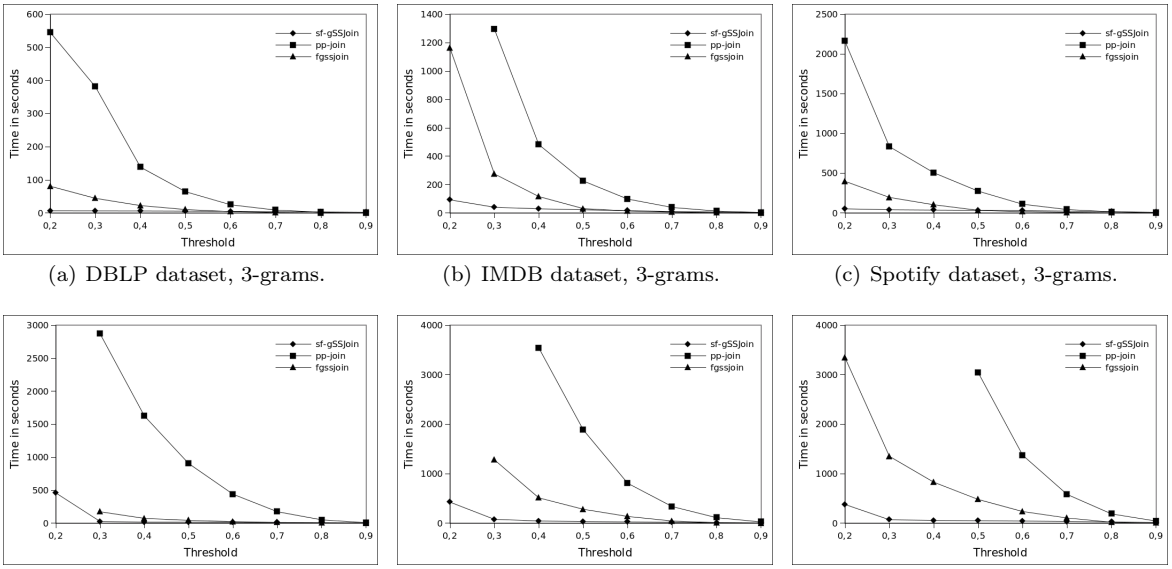
7.2 Performance Results

We now analyze and compare the efficiency of `sf-gSSJoin` against `ppjoin`, `fgssjoin`, and `gSSJoin`. To this end, we measured the runtime performance with varying threshold parameter and for q-grams of size 2 and 3. Figures 4 and 5 show the results. As a general trend, `sf-gSSJoin` outperforms `ppjoin` in almost all scenarios. They exhibit similar performance at high threshold values (0.7–0.9). However, as the threshold decreases, `ppjoin` performance drops dramatically. As discussed in Section 3.3, `ppjoin`, as most set similarity join algorithms, heavily depends on prefix filtering effectiveness to obtain performance. At lower thresholds, prefix filtering becomes ineffective and the verification workload skyrockets. In contrast, `sf-gSSJoin` runtimes increase smoothly, thereby achieving increasing speedups over `ppjoin`.

Furthermore, the performance advantage of `sf-gSSJoin` over `ppjoin` increases with the dataset size. The peak speedup gain for `sf-gSSJoin` is obtained on the DBLP dataset with 2-grams and threshold value of 0.3: in this setting, `sf-gSSJoin` is 109x faster than `ppjoin`. Unfortunately, we could not obtain the `ppjoin` runtimes for all databases at low thresholds due to memory limitations. However, the performance gains clearly increase when threshold values decrease.

As for `ppjoin`, the `fgssjoin` algorithm exhibits increasing runtimes on lower thresholds similar. It outperforms `sf-gSSJoin` at high thresholds (usually above 0,6), but its performance deteriorates as threshold values decrease. This behavior is due to the use of the same filtering techniques of `ppjoin`. As in `ppjoin`, the filtering phase cost increases and more candidates need to be evaluated in the verification phase. The highest speedup gain of `sf-gSSJoin` against `fgssjoin` was 19,5x on the Spotify database using 2-grams with threshold value equal to 0.3.

[1] http://dblp.uni-trier.de/
[2] http://dbis-twitterdata.uibk.ac.at/spotifyDataset/
[3] Information courtesy of IMDb (http://www.imdb.com/interfaces) Used with permission.
[4] http://www.cse.unsw.edu.au/ weiw/project/simjoin.html



(a) DBLP dataset, 3-grams. (b) IMDB dataset, 3-grams. (c) Spotify dataset, 3-grams.

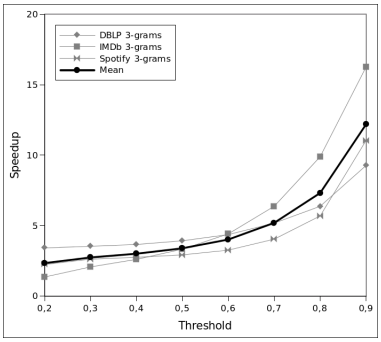
(d) DBLP dataset, 2-grams. (e) IMDB dataset, 2-grams. (f) Spotify dataset, 2-grams.

Fig. 4. ppjoin, fgssjoin and sf-gSSJoin runtimes.

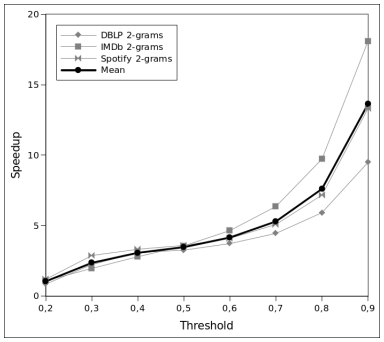
When compared with `gSSJoin`, `sf-gSSJoin` shows a better performance at higher threshold values, when size pruning can discard more blocks of sets. At low thresholds (0.2–0.3) minimum size limits are too low to be effective and they exhibit similar runtimes, but generally `sf-gSSJoin` outperforms our former algorithm. As Figure 5(b) shows, the highest speedup obtained is on the IMDb database with 2-grams and 0.9 as the threshold value. In this scenario `sf-gSSJoin` is 18x faster than `gSSJoin`. The blocking division scheme of `sf-gSSJoin` plays an important role in effectively pruning the comparison space based on size filtering. Moreover, `gSSJoin` indexes all the sets and reads all the sets on the inverted lists when probing a set and verifies if the sets were compared earlier to avoid returning the same pair twice. This verification is needed only when the indexed block and the probe block are the same. Therefore, a large number of dissimilar set pairs are never compared.

We observe that `sf-gSSJoin` and `gSSJoin` suffer a drop in performance for threshold value at 0.2 because of the huge number of set pairs in the result that have to be copied to the CPU memory. On the DBLP database with 2-grams and 0.2 threshold value, `sf-gSSJoin` shows an inferior performance than `gSSJoin` as result of its compaction technique. Using a simple atomic operation to compact the result can be inefficient when a large number of sets are similar. When there is a large number of similar sets, many atomic sums are performed by `sf-gSSJoin` to determine the positions of set pairs on the result array. These atomic operations have a high cost and may lead to an almost sequential execution. The `gSSJoin` compaction technique divides the result array into blocks to avoid the atomic sums overhead.

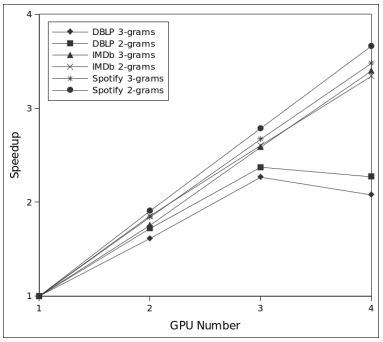
Table II shows the execution time using multiple GPUs on all databases with 0.5 as the threshold value. The data is divided between the GPUs, making our solution scalable. The use of multiple GPUs showed an almost linear speedup increase for larger databases, as can be seen in Figure 5(c). But for small databases or at high threshold values (0.8–0.9) the cost of memory copy between the CPU memory and the GPUs memories is high. And, as mentioned before, if the set sizes distribution is more homogeneous, the irregular number of blocks to be compared can cause a workload imbalance between GPUs and speedups will not be linear.



(a) sf-gSSJoin speedups when compared to gSSJoin using 3-grams.



(b) sf-gSSJoin speedups when compared to gSSJoin using 2-grams.



(c) Multi-GPU Speedup.

Fig. 5. Speedups.

Table II. Execution Time (secs) Using Multiple GPUs, Threshold = 0.5

Number of GPUs:	1	2	3	4
DBLP 2gram	15.47	9	6.52	6.81
DBLP 3gram	5.78	3.6	2.55	2.78
IMDb 2gram	34.11	18.4	13.1	10.21
IMDb 3gram	22.77	13	8.8	6.7
Spotify 2gram	48.16	25.27	17.28	13.15
Spotify 3gram	29.92	16.26	11.2	8.6

8. CONCLUSIONS

In this article, we proposed GPU-based solutions to the set similarity join problem. We present two new parallel algorithms based on an inverted index implemented on the GPU. We exploit a number of strategies to optimize our algorithms including intense GPU occupancy, hierarchical memory, coalesced memory access, load balancing, and a block division scheme. As a result, we achieve significant speedups over state-of-the-art CPU-based and GPU-based algorithms in most settings. We also propose multi-GPU variants to further exploit task parallelism in addition to data parallelism, which exhibit almost linear speedup with the number of GPUs. In future work, we plan to investigate the integration of our GPU-based algorithms into a distributed framework.

REFERENCES

Agrawal, S., Chakrabarti, K., Chaudhuri, S., and Ganti, V. Scalable ad-hoc entity extraction from text collections. *Proceedings of the VLDB Endowment* 1 (1): 945–957, 2008.

Arasu, A., Ganti, V., and Kaushik, R. Efficient Exact Set-Similarity Joins. In *Proceedings of the International Conference on Very Large Data Bases*. Seoul, Korea, pp. 918–929, 2006.

Bayardo, R. J., Ma, Y., and Srikant, R. Scaling up All Pairs Similarity Search. In *Proceedings of the International World Wide Web Conferences*. Banff, Canada, pp. 131–140, 2007.

Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. Min-Wise Independent Permutations (Extended Abstract). In *STOC*. pp. 327–336, 1998.

Chacón, A., Marco-Sola, S., Espinosa, A., Ribeca, P., and Moure, J. C. Thread-cooperative, Bit-parallel Computation of Levenshtein Distance on GPU. In *ICS*. pp. 103–112, 2014.

Chaudhuri, S., Ganti, V., and Kaushik, R. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the IEEE International Conference on Data Engineering*. Atlanta, USA, pp. 5, 2006.

Cruz, M. S. H., Kozawa, Y., Amagasa, T., and Kitagawa, H. Accelerating set similarity joins using gpus. *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems* vol. 28, pp. 1–22, 2016.

Deng, D., Li, G., Hao, S., Wang, J., and Feng, J. MassJoin: A Mapreduce-based Method for Scalable String Similarity Joins. In *Proceedings of the IEEE International Conference on Data Engineering*. Chigago, USA, pp. 340–351, 2014.

Deng, D., Li, G., Wen, H., and Feng, J. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proceedings of the VLDB Endowment* 9 (4): 360–371, 2015.

Doan, A., Halevy, A. Y., and Ives, Z. G. *Principles of Data Integration*. Morgan Kaufmann, 2012.

Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of the International Conference on Very Large Data Bases*. Rome, Italy, pp. 491–500, 2001.

Hassanzadeh, O., Chiang, F., Miller, R. J., and Lee, H. C. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *Proceedings of the VLDB Endowment* 2 (1): 1282–1293, 2009.

Indyk, P. and Motwani, R. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the ACM SIGACT Symposium on Theory of Computing*. Dallas, USA, pp. 604–613, 1998.

Kim, J. and Lee, H. Efficient Exact Similarity Searches Using Multiple Token Orderings. In *Proceedings of the IEEE International Conference on Data Engineering*. Washington, USA, pp. 822–833, 2012.

Kirk, D. B. and Wen-mei, W. H. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

Leskovec, J., Rajaraman, A., and Ullman, J. D. *Mining of Massive Datasets*. Cambridge University Press, 2014.

Lieberman, M. D., Sankaranarayanan, J., and Samet, H. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *Proceedings of the IEEE International Conference on Data Engineering*. Hannover, Germany, pp. 1111–1120, 2008.

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

Matsumoto, T. and Yiu, M. L. Accelerating Exact Similarity Search on CPU-GPU Systems. In *Proceedings of the IEEE International Conference on Data Mining*. Atlantic City, USA, pp. 320–329, 2015.

Meraji, S., Schiefer, B., Pham, L., Chu, L., Kokosielis, P., Storm, A. J., Young, W., Ge, C., Ng, G., and Kanagaratnam, K. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. San Francisco, USA, pp. 1951–1960, 2016.

Qin, J., Wang, W., Lu, Y., Xiao, C., and Lin, X. Efficient Exact Edit Similarity Query Processing with the Asymmetric Signature Scheme, booktitle = sigmod, pages = 1033–1044, year = 2011, address = Athens, Greece.

Quirino, R. D., Ribeiro-Júnior, S., Ribeiro, L. A., and Martins, W. S. fgssjoin: A GPU-based Algorithm for Set Similarity Joins. In *International Conference on Enterprise Information Systems*. Porto, Portugal, pp. 152–161, 2017.

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

Ribeiro, L. A., Schneider, N. C., de Souza Inácio, A., Wagner, H. M., and von Wangenheim, A. Bridging Database Applications and Declarative Similarity Matching. *Journal of Information and Data Management* 7 (3): 217–232, 2016.

Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. gSSJoin: a GPU-based Set Similarity Join Algorithm. In *Proceedings of the Brazilian Symposium on Databases*. Salvador, Brazil, pp. 64–75, 2016.

Rong, C., Lin, C., Silva, Y. N., Wang, J., Lu, W., and Du, X. Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. In *Proceedings of the IEEE International Conference on Data Engineering*. San Diego, USA, pp. 1059–1070, 2017.

Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Paris, France, pp. 743–754, 2004.

Sidney, C. F., Mendes, D. S., Ribeiro, L. A., and Härder, T. Performance Prediction for Set Similarity Joins. In *Proceedings of the ACM Symposium on Applied Computing*. Salamanca, Spain, pp. 967–972, 2015.

Spertus, E., Sahami, M., and Buyukkokten, O. Evaluating similarity measures: a large-scale study in the orkut social network. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Chigago, USA, pp. 678–684, 2005.

Vernica, R., Carey, M. J., and Li, C. Efficient Parallel Set-similarity Joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Indianapolis, USA, pp. 495–506, 2010.

Wang, J., Li, G., and Feng, J. 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*. pp. 85–96, 2012.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. Efficient Similarity Joins for Near-duplicate Detection. *ACM Transactions on Database Systems* 36 (3): 15:1–15:41, 2011.

Zhang, Y., Li, X., Wang, J., Zhang, Y., Xing, C., and Yuan, X. An Efficient Framework for Exact Set Similarity Search Using Tree Structure Indexes. In *Proceedings of the IEEE International Conference on Data Engineering*. San Diego, USA, pp. 759–770, 2017.

Zipf, G. K. and Wilson, L. Human Behavior and the Principle of Least Effort, 1949.