


Set Similarity Joins on Heterogeneous Clusters

Larissa Ramos Marques Silva  [[Implanta IT Solutions LTDA](#) | larissa.ramos@implantait.com.br]

Leonardo Andrade Ribeiro   [[Universidade Fderal de Goiás](#) | laribeiro@inf.ufg.br]

 *Instituto de Informática (INF), Universidade Federal de Goiás (UFG), Alameda Palmeiras, Quadra D, Câmpus Samambaia, Goiânia, GO, Brazil.*

Received: 27 March 2023 • **Published:** 22 December 2023

Abstract

Set similarity join (SSJ) is a fundamental operation widely used in many application scenarios, including data discovery, cleaning, and integration. As this operation is computationally expensive, its runtime can be excessive on large volumes of data. Previous research has focused on improving SSJ scalability using distributed computing or the massive parallelism available in GPUs, but not both. Hence, these efforts cannot fully exploit the processing power of increasingly heterogeneous computing architectures. In this article, we present an approach to evaluating SSJ on a heterogeneous cluster of compute nodes equipped with CPU and GPU. We propose a cost model to distribute the workload between these processors and apply this model to integrate two algorithms, one distributed and the other parallel, in a coprocessing fashion. Experimental results show that our proposal is efficient, scalable, and outperforms previous work.

Keywords: Advanced Query Processing, Distributed Computing, GPU, Heterogeneous Hardware, Set Similarity Join

1 Introduction

Similarity join finds all pairs of similar objects in a dataset. Two objects are considered similar if the value returned by a similarity function applied to these objects is not less than a specified threshold. Similarity join is a fundamental operation in data discovery, cleaning, and integration [Chaudhuri *et al.*, 2006]. For example, similarity joins can be used for *entity matching*, i.e., identifying data instances that refer to the same real-world entity.

Set similarity join (SSJ) maps input objects to sets and assesses the pairwise similarity between objects based on the overlap of their corresponding sets [Chaudhuri *et al.*, 2006]. This special type of similarity join is well-suited to sparse, high-dimensional data; string is a prime example. Despite its simplicity, SSJ is quite effective in many scenarios. For instance, on datasets exhibiting syntactical perturbations, such as those caused by typos, SSJ has shown accuracy comparable to complex solutions adopting representations based on large language models [Suri *et al.*, 2021].

SSJ is a computationally expensive operation. The naive solution evaluates the similarity function on every set pair, clearly leading to a prohibitive runtime on large datasets. There is a host of optimization techniques for SSJ, many of them based on filters to reduce the comparison space [Sarawagi and Kirpal, 2004; Chaudhuri *et al.*, 2006; Bayardo *et al.*, 2007; Xiao *et al.*, 2011; Ribeiro and Härder, 2011; Mann *et al.*, 2016; Fier and Freytag, 2022]. Nevertheless, the runtime of SSJ queries still grows quadratically with input size [Xiao *et al.*, 2011; Ribeiro and Härder, 2011].

The massive parallelism available in modern graphic processing units (GPUs) is attractive to accelerate the execution of SSJ queries. Indeed, prior work reported significant speedups using GPUs as compared to purely CPU-based solutions [Ribeiro-Júnior *et al.*, 2016, 2017; Quirino *et al.*, 2017]. However, GPUs have limited onboard memory ca-

capacity, thereby restricting scalability to large data volumes.

Distributed computing is a natural way to attain scalability in SSJ queries [Oliveira *et al.*, 2017, 2018; Fier *et al.*, 2018]. Load balancing and per-node efficiency are fundamental concerns in this context. Unfortunately, distributed computing platforms such as Apache Spark [Zaharia *et al.*, 2012] only consider data locality for allocating tasks to nodes [Xu *et al.*, 2018]. Underlying hardware capabilities are disregarded in this context, leading to resource underutilization and suboptimal performance.

In this article, we present a proposal to efficiently evaluate SSJ queries using distributed computing and massive parallelism. Specifically, we consider a data processing environment based on a heterogeneous shared-nothing architecture, with nodes equipped with CPU and GPU. Modern hardware platforms for data-intensive operations increasingly use GPUs as accelerators [Rosenfeld *et al.*, 2022], and our work aims at fully exploiting the computational power of such platforms for SSJ query processing. To the best of our knowledge, SSJ on heterogeneous clusters has not been previously investigated in the research literature. The main technical challenge in this context is to deal with the computing power asymmetry among processors. We propose a cost model to guide the workload distribution between CPU and GPU in a coprocessing fashion. Further, we apply this model to incorporate a parallel algorithm into a distributed SSJ algorithm, thereby enabling the effective use of GPU massive parallelism. We conduct an extensive experimental study on publicly available datasets. The results show that our cost model precisely identifies the best workload assignment for different configurations of computational resources. In particular, our solution achieved expressive performance improvements over a baseline, uniform workload distribution, and outperformed previous work.

This article is a significantly extended and revised version of a previous conference paper [Silva and Ribeiro, 2022].

The present article addresses a broader spectrum of cluster configurations, including the following new material: a formal categorization of different cluster configurations by their data processor heterogeneity (Section 3.1); a generalization of the previously proposed cost model seamlessly encompassing all categories of cluster configurations considered in this article (Section 3.2); an extended and more detailed evaluation including new empirical experiments (Section 4). Moreover, we introduce the concept of workload-distribution table, present the algorithm for its construction, and incorporate it into our SSJ algorithm to obtain a refined workload splitting (Section 3.3). Besides these technical contributions, we also include a thorough discussion on state-of-the-art SSJ algorithms and how our work relates to CPU-GPU heterogeneous computing techniques (Section 5).

The remainder of this article is organized as follows. In Section 2, we provide background material. Our proposed solution is presented in Section 3 and experimentally evaluated in Section 4. We discuss relevant related work in Section 5 before we wrap up the conclusions and outline of future work in Section 6.

2 Background

In this section, we first formally define the SSJ problem. Then, we review some important optimization techniques. Finally, we briefly overview frameworks for distributed computing and the architecture of GPUs.

2.1 Problem Definition

We consider the problem of efficiently answering SSJ queries over a set collection. Set elements are referred to as *tokens*. We call the process of mapping data objects to sets *tokenization*. SSJ has a join condition formed by a similarity function and a user-defined threshold. The similarity function receives two sets as input and returns a value in $[0, 1]$ representing their similarity; a larger value indicates a higher similarity. SSJ computes all set pairs in the input collection whose similarity is not less than the threshold.

Definition 1 (Set Similarity Join(SSJ)). Given a set collection \mathcal{C} — each set in \mathcal{C} contains distinct *tokens* sampled from a finite universe \mathcal{U} , a similarity function $sim : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \rightarrow [0, 1]$, and a threshold $\tau \in [0, 1]$, SSJ returns all set pairs $(r, s) \in \mathcal{C} \times \mathcal{C}$ s.t. $sim(r, s) \geq \tau$.

SSJ can be applied to any data object that can be represented as a set of tokens, which covers a vast range of domains, including query logs, click-streams, user preference data, and social media information [Mann *et al.*, 2016]. For example, users can be represented by sets, where tokens are their interests, friends, products purchased, movies watched, albums rated, and so on. String data can be tokenized 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. To this end, the string is (conceptually) extended by prefixing and suffixing it with $q - 1$ occurrences of a special character “\$”, so all its characters participate in exact q *q-grams*. For example,

the string “*similarity*” can be mapped to the set of 3-grams tokens $\{\$ \$ \$, \$ s i, \$ i m, i m i, m i l, i l a, l a r, a r i, r i t, i t y, t y \$, y \$ \$\}$. Finally, as the result of a tokenization method can be a multiset, we further append the symbol of a sequential ordinal number to each occurrence of a token to convert multisets into sets, e.g. the multiset $\{a, b, b\}$ is converted to $\{a \circ 1, b \circ 1, b \circ 2\}$. In the following, we assume that all data objects have already been mapped to sets of tokens.

Similarity functions for sets are based on the overlap, which is normalized to account for the size difference between the two input sets. A popular similarity function is the Jaccard similarity: given two sets r and s , the Jaccard similarity between them is defined as $J(r, s) = \frac{|r \cap s|}{|r \cup s|}$. In this article, we focus on the Jaccard similarity, but all the following techniques hold for other similarity functions, such as Dice and Cosine [Ribeiro and Härder, 2011].

2.2 Optimizations

Predicates based on similarity functions can often be equivalently represented in terms of an *overlap bound* [Chaudhuri *et al.*, 2006]. Thus, we have $J(r, s) \geq \tau \iff |r \cap s| \geq \alpha = \frac{1}{1+\tau} \times (|r| + |s|)$. SSJ is then reduced to the problem of finding set pairs whose overlap is not less than such bound.

The above set-overlap formulation enables the derivation of filters to avoid bulky similarity evaluation for each pair of sets. The *prefix filter* technique [Chaudhuri *et al.*, 2006] allows discarding set pairs that cannot meet the similarity predicate by examining only a subset of them.

Lemma 1 (Prefix Filter Principle). Consider that the tokens of all sets are sorted based on some total token order. Let $pref(x, p)$ be the subset of x containing its first p tokens. Then, for any two sets r and s , the following holds: $|r \cap s| \geq \alpha \implies pref(r, \alpha + 1) \cap pref(s, \alpha + 1) \neq \emptyset$.

We can safely prune set pairs sharing no prefix token as they cannot meet the overlap bound. For the Jaccard similarity and threshold τ , we can identify all candidate matches of a given set r using $pref(r, \lfloor (1 - \tau) \times |r| \rfloor + 1)$. We denote this prefix simply by $pref(r)$. Finally, tokens are ordered by increasing order of their frequency in \mathcal{C} , which moves lower-frequency tokens to prefix positions and, therefore, increases filtering effectiveness.

Another popular optimization technique is the *length filter* [Sarawagi and Kirpal, 2004]. Intuitively, the difference in size between two similar sets cannot be too large. Thus, one can promptly discard set pairs whose sizes differ enough.

Lemma 2 (Length Filter Principle). For any two sets r and s , and a similarity threshold τ , the following holds: $J(r, s) \geq \tau \implies \min\left(\frac{|r|}{|s|}, \frac{|s|}{|r|}\right) \geq \tau$.

Example 1. Consider the set collection $\mathcal{C} = [r, s, t]$, where $r = \{C, D, E, F, G\}$, $s = \{A, B, C, D, E, F, G\}$, and $t = \{D, E, F, G, H, I, J, K\}$, and an SSJ operation with similarity threshold $\tau = 0.7$. We have $pref(r) = \{C, D\}$, $pref(s) = \{A, B, C\}$, and $pref(t) = \{D, E, F\}$. The pair (s, t) can be discarded because $pref(s) \cap pref(t) = \emptyset$. Indeed, $J(s, t) = \frac{4}{7+8-4} \approx 0.36$. The pair (r, t) passes through the prefix filter because $pref(r) \cap pref(t) = \{D\}$,

Algorithm 1: $SSJ(\mathcal{C}, \tau)$

Input: Sorted set collection \mathcal{C} , threshold τ
Output: All pairs (r, s) s.t. $J(r, s) \geq \tau$

```

1  $I_1, \dots, I_{|\mathcal{C}|} \leftarrow \emptyset$ 
2 foreach  $r \in \mathcal{C}$  do
3    $M \leftarrow$  an empty map from set to a similarity score
4   foreach  $tok \in pref(r)$  do // prefix filter
5     foreach  $s \in I_{tok}$  do
6       if  $|s| < |r| \times \tau$  then // length filter
7          $M[s] \leftarrow -\infty$ 
8       else
9          $M[s] \leftarrow M[s] + 1$ 
10       $I_{tok} \leftarrow I_{tok} \cup \{r\}$ 
11   Output ( $Verify(x, M, \tau)$ )

```

but not the length filter because $|r| = 5 < |s| \times \tau = 8 \times 0.7 = 5.6$. Indeed, $J(r, t) = \frac{4}{5+8-4} \approx 0.44$. The pair (r, s) passes through both filters, i.e., $pref(r) \cap pref(s) = \{C\}$ and $|r| = 5 \geq |s| \times \tau = 7 \times 0.7 = 4.9$. Since we have $J(r, s) = \frac{5}{5+7-5} \approx 0.71$, (r, s) is the only pair returned by the SSJ operation over \mathcal{C} .

2.3 General SSJ Algorithm

Most current SSJ solutions adopt a filtering-verification framework [Mann *et al.*, 2016]. In the filtering phase, candidate pairs are collected using an inverted index, which is built as the input set collection is processed. In the verification phase, the overlap of each candidate pair is calculated, and those pairs satisfying the similarity predicate are output. SSJ executes as an index nested loop join, as described in Algorithm 1. An inverted list I_{tok} stores all sets containing a token tok in their prefix (Line 1). For each set r in \mathcal{C} , the prefix filter is applied by using only its prefix tokens for lookups (Line 4) and indexing, where a reference to r is appended to the inverted lists (Line 10). In this way, set pairs sharing no prefix tokens are never considered candidates. The length filter is employed for each set s found in the inverted lists (Line 6). At this point, various other filtering techniques can also be applied to reduce the comparison space, such as positional and suffix filters [Xiao *et al.*, 2011]. If set s passes through the filters, its similarity score is accumulated on a map (Line 9). After the filtering phase, the similarity between r and each of its candidates is fully calculated in the verification phase, and similar pairs are sent to the output (Line 11). Verification can be highly optimized by exploiting the token ordering in a merge-like fashion and the overlap bound to define early stopping conditions [Ribeiro and Härder, 2011]. Finally, the set collection can be pre-sorted according to set size to enable further index reduction either at indexing time [Bayardo *et al.*, 2007] or dynamically during the filtering phase [Ribeiro and Härder, 2011]. It has been shown that the runtime decrease achieved by such optimizations largely compensates for the additional sorting cost.

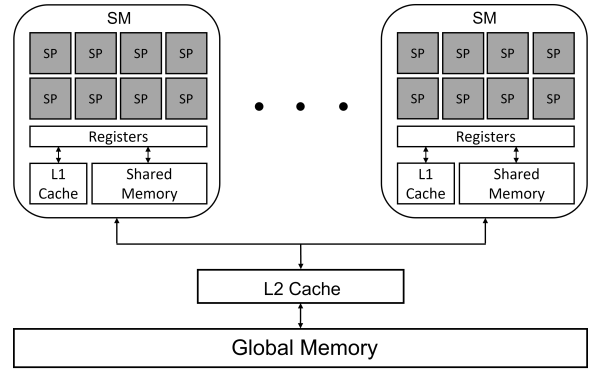


Figure 1. GPU architecture overview.

2.4 Distributed Computing Frameworks

Distributed computing frameworks automatically partition the input data over different nodes and provide a programming model to express complex transformations on this data. MapReduce is one of the earliest and most popular of such frameworks [Dean and Ghemawat, 2004]. Its computing platform provides an abstraction for parallelization, fault tolerance, load balancing, scheduling, and synchronization of data processing tasks. Input data is represented as a set of key/value pairs. The programming API consists of two user-defined functions: $map(k1, v1) \rightarrow list(k2, v2)$ and $reduce(k2, list(v2)) \rightarrow list(k3, v3)$.

Apache Spark is another widely used framework for processing large-scale datasets on shared-nothing architectures. It provides an abstraction for in-memory data storage and parallel processing on clusters called *resilient distributed datasets* (RDDs). More specifically, an RDD is an immutable, partitioned collection of objects created by coarse-grained deterministic operations. In particular, the operations called `flatMap` and `groupByKey` allow expressing the MapReduce programming model.

2.5 Graphic Processing Units

GPUs were originally designed as special-purpose coprocessors for dedicated graphics rendering. With the advent of efficient parallel programming models, such as NVIDIA CUDA and OpenCL, these devices evolved into powerful accelerators for general-purpose computing. Figure 1 depicts an overview of modern GPU architecture. Basically, a GPU contains multiple compute units called *Streaming Multiprocessor (SMs)*. Inside each SM there are several *Streaming Processors (SPs)* cores that operate on different data but in a synchronous way. Memory is organized into a hierarchy. The largest (and slowest) memory is the *global memory*. Data accessed from the global memory is cached in the *L2 cache*, which is shared across all SMs as the global memory, and, optionally, also in the *L1 cache*, which is local to each SM. The *shared memory* is controlled by the programmer for use as a scratchpad and can be accessed by all SPs in an SM. Finally, *registers* store per-thread data.

A GPU program exposes parallelism through data-parallel functions, called *kernels*, that are offloaded to the GPU. The programmer configures the number of threads to be used; unlike CPUs, the overhead of thread creation and switching in GPUs is negligible. These threads are organized in *thread*

blocks that are assembled into a grid structure. When a kernel is launched, the blocks within a grid are distributed on idle SMs, and the threads are mapped to the SPs. A thread block in an SM is divided into multiple schedule units, called *warps*, that are dynamically scheduled on the SM. Warps execute in a *Single Instruction Multiple Threads (SIMT)* model, where its threads execute the same instruction simultaneously but on different data. Data must first be transferred over a system bus for processing on a dedicated GPU. The bandwidth of current GPU interconnects such as PCI Express (PCIe) 3.0 is significantly slower than that of the CPU main memory, leading to a potential transfer bottleneck. The emergence of faster GPU interconnects, such as NVLink 2.0, enables the GPU to access CPU memory with the full memory bandwidth, potentially eliminating this bottleneck.

3 SSJ on Heterogeneous Clusters

In this section, we describe our solution to efficiently execute SSJ queries on heterogeneous clusters equipped with CPU and GPU. We first define levels of cluster heterogeneity w.r.t the available data processors. Then, the main contribution of this article is presented: a CPU-GPU cost model to distribute the workload between CPU and GPU according to their relative processing power. Finally, we employ the proposed model to seamlessly integrate two algorithms, one massively parallel designed for GPUs and the other distributed designed for shared-nothing architectures, thus fully exploiting the available hardware resources.

3.1 Cluster Heterogeneity Levels

We use the term cluster to refer to the traditional notion of a computer cluster, i.e., a collection of independent computer nodes connected together via high-speed network technology. Further, we focus on shared-nothing architectures, i.e., distributed computing architectures in which neither memory nor storage is shared among processors. In this context, we categorize the cluster configurations addressed in this article by the level of heterogeneity of their processors as formally defined below.

Definition 2 (Cluster Heterogeneity Levels). Consider a cluster CT formed by computing nodes N_1, \dots, N_n . Each node N_i has a set of processors $N_i.P = \{P_1, \dots, P_m\}$. A processor P is of type CPU or GPU, i.e., $P.type \in \{CPU, GPU\}$ and has a specification $P.spec$; specifications of the same type of processors are uniform within a node but can vary along different nodes. We formally define the levels of cluster heterogeneity as follows.

- *Homogeneous (HM) cluster*: CT is a HM cluster if $|\bigcup_{i=1}^n \bigcup_{j=1}^m N_i.P_j.type| = 1$.
- *Heterogeneous (HT) cluster*: CT is an HT cluster if $(\forall N_i \in CT, \bigcup_{j=1}^m N_i.P_j.type = \{CPU, GPU\}) \wedge |\bigcup_{i=1}^n \{P.spec : N_i.P.type = CPU\}| = 1 \wedge |\bigcup_{i=1}^n \{P.spec : N_i.P.type = GPU\}| = 1$.
- *Highly Heterogeneous (HHT) cluster*: CT is an HHT cluster if $(\forall N_i \in$

$$CT, \bigcup_{j=1}^m N_i.P_j.type = \{CPU, GPU\}) \wedge (|\bigcup_{i=1}^n \{P.spec : N_i.P.type = CPU\}| > 1 \vee |\bigcup_{i=1}^n \{P.spec : N_i.P.type = GPU\}| > 1).$$

In an HM cluster, all nodes have the same processor type, either CPU or GPU. In HT and HHT clusters, all nodes are equipped with, possibly multiple, CPU and GPU. Processors of the same type in an HT cluster have the same specifications, whereas in an HHT cluster, processors of the same type have varying specifications along different nodes.

Example 2. As a concrete example, consider a cluster CT , whose initial configuration contains two compute nodes; each node has a CPU Intel Xeon E5-2650. We classify CT as an HM cluster. Then, a GPU NVIDIA Tesla K40 is added to CT , leading to a new version CT' , which is classified as an HT cluster. Finally, another version is built, CT'' , by adding a third node with a CPU AMD EPYC 7452 and a GPU NVIDIA Tesla V100. We classify CT'' as an HHT cluster.

Clearly, there exist cluster configurations that do not fit the above definitions. For example, we may have a cluster where some nodes contain both CPU and GPU while others only CPU. Adapting our proposal for dealing with such scenarios is conceivable but left for future work.

Answering SSJ queries on HM clusters has attracted considerable research attention over the years (see Section 5). In contrast, evaluating SSJ on HT and HHT clusters has so far been ignored, which is our focus next.

3.2 Cost Model for CPU-GPU Coprocessing

The crux of a coprocessing strategy on heterogeneous hardware is identifying the best workload distribution. On the one hand, such distribution should avoid resource underutilization by keeping all processors busy. On the other hand, overloading a processor can lead to disastrous performance because the overall execution time is determined by the last processor to finish its workload.

Given the general notion that in-memory execution of traditional database operators is typically memory-bandwidth bound, the work of Shanbhag *et al.* [2020] presents a cost model to predict runtime selection, projection, and join on CPU and GPU. The key insight is that the ratio of operator runtime on CPU to runtime on GPU roughly follows the ratio of memory bandwidth ratio of these two devices. Inspired by this insight, we propose a model for workload distribution between CPU and GPU.

Definition 3 (CPU-GPU Cost Model). Let W^{cpu} and W^{gpu} be the workload fractions of CPUs and GPUs in a cluster CT , respectively; we have $W^{cpu} + W^{gpu} = 1$. Further, let $B_1^{cpu}, \dots, B_k^{cpu}$ and $B_1^{gpu}, \dots, B_l^{gpu}$ be the memory bandwidth of the distinct specification of the CPUs and GPUs, respectively. The workload distribution in CT is defined by the following proportion:

$$\frac{W^{cpu}}{W^{gpu}} = \frac{\frac{1}{k} \sum_{i=1}^k B_i^{cpu}}{\frac{1}{l} \sum_{i=1}^l B_i^{gpu}}.$$

Example 3. Consider the HT cluster CT' in Example 2. The memory bandwidths of the CPU Intel Xeon E5-2650

Algorithm 2: $WDTBuilder(T, W^{cpu})$

input :Token-frequency table T , fraction of the CPU workload W^{cpu}
output :A workload-distribution table \mathcal{W}

```

1  $\mathcal{W} \leftarrow \emptyset, cpuLoad \leftarrow 0$ 
2  $workload \leftarrow sum(T.values())$ 
3 foreach  $(tok, freq) \in T$  do
4   if  $cpuLoad < workload \times W^{cpu}$  then
5      $\mathcal{W}.add(tok, CPU)$ 
6   else
7      $\mathcal{W}.add(tok, GPU)$ 
8    $cpuLoad \leftarrow cpuLoad + freq$ 
9 return  $\mathcal{W}$ 

```

and GPU NVIDIA Tesla K40 are 68 GB/s and 288 GB/s, respectively. Thus, we have $\frac{W^{cpu}}{W^{gpu}} = \frac{68}{288}$. By substituting W^{gpu} in the formula $W^{cpu} + W^{gpu} = 1$, we have $W^{cpu} + \frac{288}{68} \times W^{cpu} = 1 \equiv W^{cpu} \times (1 + \frac{288}{68}) = 1 \equiv W^{cpu} \approx 0.19$. Therefore, the distribution of the workload is 19% for CPU and 81% for GPU for cluster CT' . Now consider the HHT cluster CT'' in Example 2. The memory bandwidths of the CPU AMD EPYC 7452 and the GPU NVIDIA Tesla V100 are 204.8 GB/s and 897 GB/s, respectively. Now we have $\frac{W^{cpu}}{W^{gpu}} = \frac{(68+204.8)/2}{(288+897)/2} = \frac{272.8}{1185}$. By substituting W^{gpu} in the formula $W^{cpu} + W^{gpu} = 1$, we have $W^{cpu} + \frac{1185}{272.8} \times W^{cpu} = 1 \equiv W^{cpu} \times (1 + \frac{1185}{272.8}) = 1 \equiv W^{cpu} \approx 0.18$. Therefore, the distribution of the workload is 18% for CPU and 82% for GPU for cluster CT'' .

3.3 The DSJoin^{gpu} Algorithm

In this section, we illustrate the applicability of our CPU-GPU cost model to coupling distributed computing and massive parallelism for answering SSJ queries on heterogeneous clusters. In particular, we employ the model to integrate two existing SSJ algorithms: DSJoin [Oliveira *et al.*, 2017, 2018], a distributed algorithm designed for CPU-based HM clusters, and *sf-gSSJoin*, a parallel algorithm designed for GPUs [Ribeiro-Júnior *et al.*, 2017]. It is important to notice, however, that our model is not tethered to any particular SSJ algorithm: it only requires that the underlying algorithms allow workload distribution along different devices.

The DSJoin algorithm partitions the input data using prefix tokens as partition keys; only sets within the same partition are compared. This strategy indirectly applies prefix filtering as two sets wind up in the same partition only if they share a prefix token. Similarity evaluation on the partitions uses an inverted index and applies further filters, such as the length filter, analogously to Algorithm 1. The *sf-gSSJoin* algorithm builds a complete inverted index on the GPU — all tokens are indexed, not only prefix tokens — to quickly identify similar sets. The algorithm applies a block division scheme for dealing with large datasets that do not fit in GPU memory; this scheme also enables pruning of the comparison space by discarding entire blocks based on the length filter.

In our previous paper, we directly used the CPU fraction of the workload to send the lists of candidate sets either to the CPU or GPU (see [Silva and Ribeiro, 2022], Algorithm 1).

Algorithm 3: $DSJoin^{gpu}(\mathcal{C}, \tau, \mathcal{W})$

input :Set collection \mathcal{C} , threshold τ , WD table \mathcal{W}
Output: All pairs (r, s) s.t. $J(r, s) \geq \tau$

```

// Partition step
1 foreach  $r \in \mathcal{C}$  do
2    $\lfloor flatMap(funcPart(r)) \rightarrow list(key, r)$ 
3  $list(key, list(r)) \leftarrow groupByKey(list(key, r))$ 
// Verification step
4 foreach  $(key, list(r)) \in list(key, list(r))$  do
5    $sort(list(r))$  // asc set size
6   if  $\mathcal{W}.probe(key) = CPU$  then
7      $\lfloor Output(flatMap(SSJ(list(r), \tau)))$ 
8   else
9      $\lfloor Output(flatMap(sf-gSSJoin(list(r), \tau)))$ 
// Partitioning function
10 Function  $funcPart(r)$ 
11   foreach  $key \in pref(r)$  do
12      $\lfloor list.add(key, r)$ 
13   return  $list(key, r)$ 

```

While intuitive, this strategy can result in a crude workload distribution owing to the non-uniformity of the sizes of the lists. In this article, we introduce the concept of *workload-distribution (WD) table*, which more precisely reflects the workload fraction values derived from the CPU-GPU cost model. We build the WD table during preprocessing as described in Algorithm 2. Given a table with the frequency of prefix tokens and the CPU workload fraction, we first calculate the total workload by summing up all the frequencies; note that such frequencies correspond to the size of the lists of candidates. Then, we simply iterate over token-frequency pairs, associating each token with the CPU flag in the WD table until the CPU load is reached, after which tokens are associated with the GPU flag. Since table T has to be created anyway for ordering the tokens within each set, constructing the WD table incurs negligible overhead in preprocessing.

We now present DSJoin^{gpu}, an algorithm integrating DSJoin and *sf-gSSJoin*. Figure 2 shows the processing steps of DSJoin^{gpu}, which are formalized in Algorithm 3 — although we use Spark operations in the algorithm, namely *flatMap* and *groupByKey*, our solution is generic and can be easily adapted to other distributed platforms such as MapReduce. In the *partitioning step* (Lines 1–3), each input set is passed to the *funcPart* function (Lines 10–13), which extracts the prefix tokens and, in turn, associates each of them to a copy of the set. Then, the *groupByKey* operation is called to group sets with the same key into a list; all set pairs in a group are deemed candidates.

In the *verification step*, the cluster processors perform the pairwise comparison of all candidates in a distributed and independent manner. First, sets associated with the same partition key are sorted in increasing order of their size. Then, DSJoin^{gpu} uses the input WD table to determine where the computation takes place (Line 6), i.e., whether a regular CPU-based SSJ algorithm (Line 7) or *sf-gSSJoin* (Line 9) is invoked. Finally, candidate pairs satisfying the similarity predicate are added to the result set.

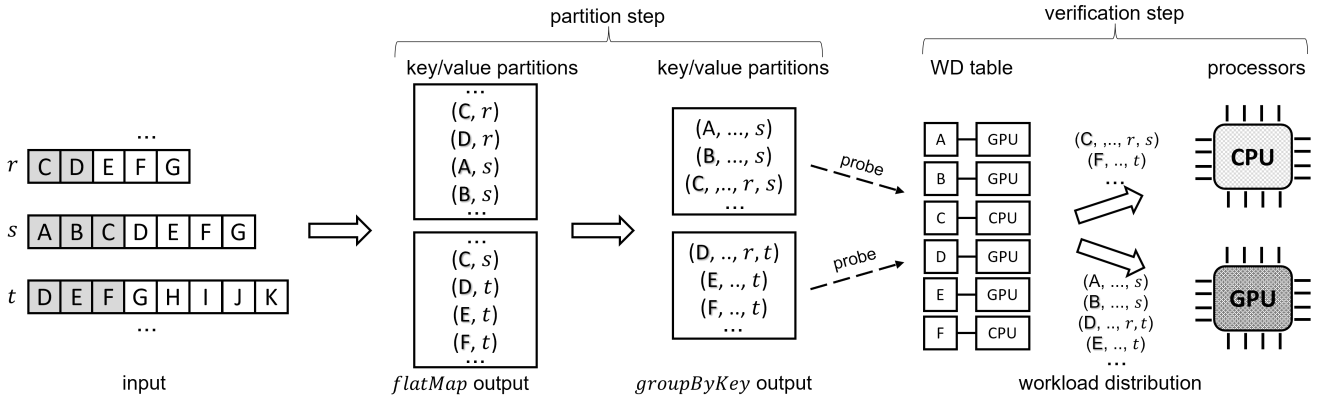


Figure 2. DSJoin^{gpu} processing steps.

As presented, the algorithm can produce duplicate pairs in the result. Two similar sets containing more than one prefix token in common will be sent to different processors and, therefore, appear multiple times in the final result. This problem is avoided by exploiting the global token order. In the verification step, we only compare candidate pairs if the first token in common is equal to the partition key. Otherwise, if this token is greater than the key, then we are sure this pair will be compared elsewhere, and the evaluation can thus be safely interrupted. We have incorporated this checking into both the CPU- and GPU-based algorithms.

4 Experiments

We now present an experimental evaluation of our proposal on HT and HHT clusters. The goals of our study are 1) to evaluate the effectiveness of our cost model in identifying the best workload distribution, 2) to test the scalability of our proposal, and 3) to compare the performance of DSJoin^{gpu} against DSJoin.

4.1 Experimental Setup

We used two publicly available, real-world datasets as sources: DBLP¹, a dataset containing information about Computer Science publications, and IMDB², a dataset containing information about movies and TV shows. In each dataset, we randomly selected entries and extracted the string content of the title attribute (title of papers for DBLP and movies for IMDB). We then generated a number of "dirty" copies of each string by performing 1–5 character-level transformations such as character insertions, deletions, and substitutions. We converted strings to upper-case letters and eliminated repeated white spaces. Each string was then mapped to a set of tokens by using q -grams of size 3, hashing each token into an integer value, and ordering the tokens within a set according to their frequency in the collection. Table 1 shows details about the datasets (minimum, maximum, and average set size, and the number of distinct tokens).

We ran the experiments on a cluster accessed through a central head node, on which processing jobs were then ex-

dataset	min. size	max. size	avg. size	$ \mathcal{U} $
DBLP	5	250	73.23	32513
IMDB	5	122	19.72	37623

Table 1. Dataset statistics.

ecuted through a queue management system called *SLURM Workload Manager*³. We derived three cluster configurations: **HT1**, formed by two compute nodes, each one with CPU Intel Xeon E5-2650 and GPU NVIDIA Tesla K40 CPUs; **HT2**, formed by a compute node with a CPU AMD EPYC 7452 and a GPU NVIDIA Tesla V100; and **HHT**, formed by the integration of HT1 e HT2. The GPUs are connected to the CPUs through PCIe 3.0. The specifications of the CPUs and GPUs were already provided in Example 3. Similarly, the distribution of the workload between CPU and GPU corresponds to the values presented in Example 3: 19% and 81% for CPU and GPU, respectively, on cluster HT1; 18% and 82% for CPU and GPU, respectively, on clusters HT2 and HHT. Overall performance was measured in average wall-clock time over repeated runs.

The algorithms DSJoin and DSJoin^{gpu} were implemented using the following programming languages and technologies: Oracle Java 11, Scala 2.11, Apache Spark 3.0.2, Apache Hadoop 3.2.1, and NVIDIA CUDA. We used the algorithm mpjoin [Ribeiro and Härder, 2011]. To integrate sf-gSSJoin into DSJoin^{gpu}, we used *JCuda*, which is a Java binding library for NVIDIA CUDA.

4.2 Experimental Results

We now report and analyze our experimental results covering the aforementioned evaluation goals. We begin with the evaluation of the proposed cost model. Figure 3 shows the results of DSJoin^{gpu} for varying fractions of the GPU workload, from 50% to 90%, and threshold values, from 0.9 to 0.7; all datasets contain 16M sets. In all settings, the peak performance was achieved when the workload was close to the value obtained from the cost model, i.e., around 80%. Remarkably, this behavior is observed even on the HHT clusters, demonstrating that our cost model satisfactorily generalizes on configurations exhibiting heterogeneity on both processor type and specification. In contrast, the worst results were obtained with the uniform workload distribution, i.e., 50/50, up to 2x slower than the distribution GPU-CPU

¹<https://dblp.uni-trier.de/>

²<https://www.imdb.com/>

³<https://slurm.schedmd.com/documentation.html>

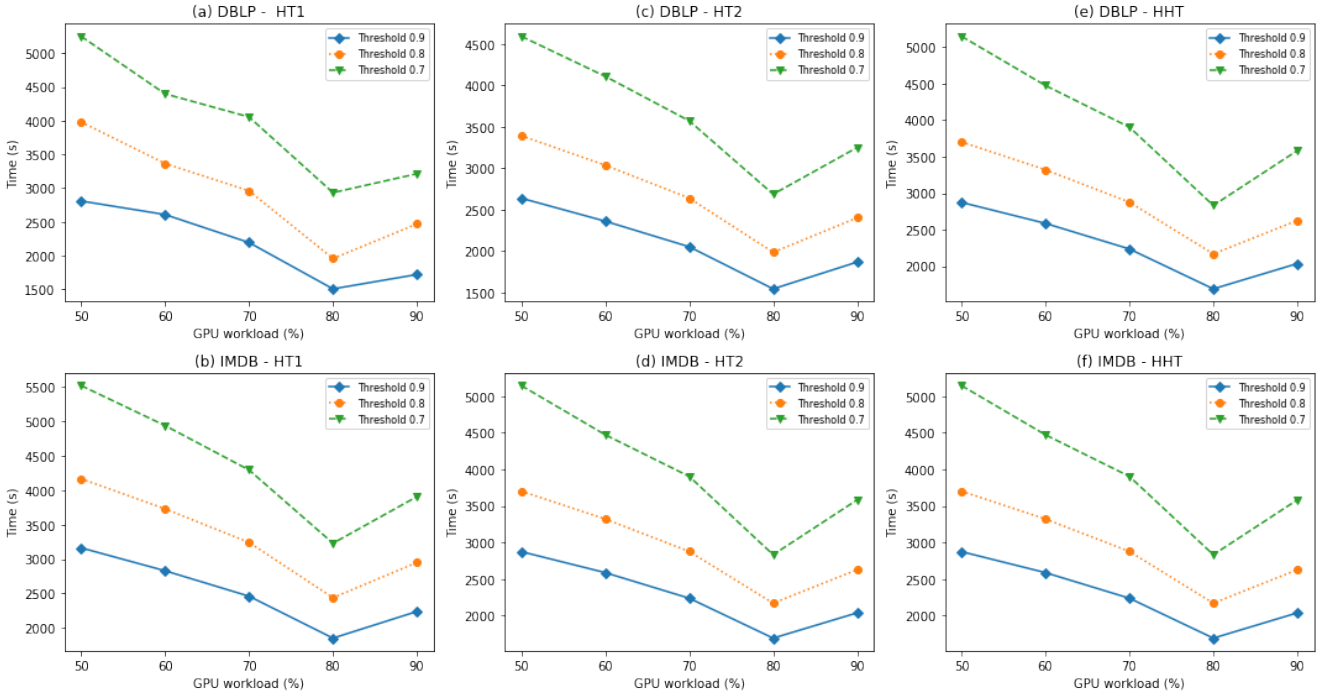


Figure 3. DSJoin^{GPU} runtimes for varying fractions of GPU workload and threshold values.

of 80/20. GPU has superior processing power and memory bandwidth and, therefore, it finished its workload share way before the CPU, which, in turn, slowed down the overall running time. Successive speedups were then achieved as the share of the workload assigned to the GPU increased up to the point when it reached 80%. Afterward, performance dropped as the GPU became overloaded.

An important observation from the speedups obtained is that we were able to overcome the data transfer bottleneck between CPU-GPU with the adopted coprocessing scheme. Some operations, such as selections and ungrouped aggregations, are bandwidth-bound and, thus, suffer from the slow PCIe connection in a coprocessing scheme. In contrast, traditional join operations are considered latency-bound as they typically perform random access to a large internal state, which benefits GPU’s faster memory and larger caches compared to the CPU [Rosenfeld *et al.*, 2022]. Our results suggest that such characteristics are also present in SSJ queries, and, as a result, performance gains can be achieved even when the data initially resides in the CPU memory. Speedups of SSJ queries in CPU-GPU coprocessing scheme were also reported in [Bellas and Gounaris, 2021].

DSJoin^{GPU} follows the same trend as any SSJoin employing filters that exploit the threshold: the higher the threshold, the better performance as the filters become more effective. For prefix filtering, higher thresholds generate smaller prefixes, reducing both communication and computation costs. Conversely, lower thresholds translate into more sets transmitted over the network and processed at the compute nodes. In this context, the block division scheme of sf-GSSJoin comes at hand to deal with larger data volumes on the GPU.

Next, we conducted scalability tests on datasets with a varying number of sets, from 16M to 48M. The threshold was fixed at 0.9. Figure 4 shows the results. The behavior of DSJoin^{GPU} is identical to the previous experiment: the

best performance is achieved with the workload distribution closer to the values returned by our cost model, i.e, 80% GPU and 20% CPU, and the worst was obtained with a uniform distribution. The runtimes do not scale linearly with the number of sets, but this result was expected because the verification workload of SSJ algorithms grows quadratically with the input size [Xiao *et al.*, 2011; Ribeiro and Härder, 2011].

In our last experiment, we compared DSJoin^{GPU} against DSJoin, which is purely CPU-based. The workload distribution of DSJoin^{GPU} was set to 80% GPU and 20% CPU, as it was defined by the cost model and, indeed, is the best distribution according to the previous experiments. The dataset sizes were fixed at 48M, and the threshold value ranges from 0.9 to 0.7. We used two versions of cluster HT1 for this experiment, with one and two compute nodes.

Figure 5 shows the results. DSJoin^{GPU} outperformed DSJoin in all settings. In particular, DSJoin^{GPU} is up to 63% faster than the DSJoin algorithm. Moreover, DSJoin^{GPU} using only one compute node achieved superior performance than DSJoin using two processing nodes in the HT1 cluster for thresholds 0.9 and 0.8 (Figures 5(a)–(b)). The reason for the performance advantage of DSJoin^{GPU} over DSJoin lies in the use of GPU during the verification step; note that allocating 100% of the verification workload to the CPU turns DSJoin^{GPU} into DSJoin. Figure 6 shows the runtime breakdown of the two algorithms. The flatMap and groupByKey operations constitute the partition step of the algorithms, thus encompassing all data transfer over the network. The prefix tokens are the rarest in the entire input set collection; using them as partition keys minimizes the communication load, and, as a result, flatMap and groupByKey have the least impact on the runtime. Further, both algorithms sort candidate lists on the CPU with similar timings. In contrast, the verification step, the largest portion of the overall runtime, is twice as fast in DSJoin^{GPU} compared to DSJoin. This result indi-

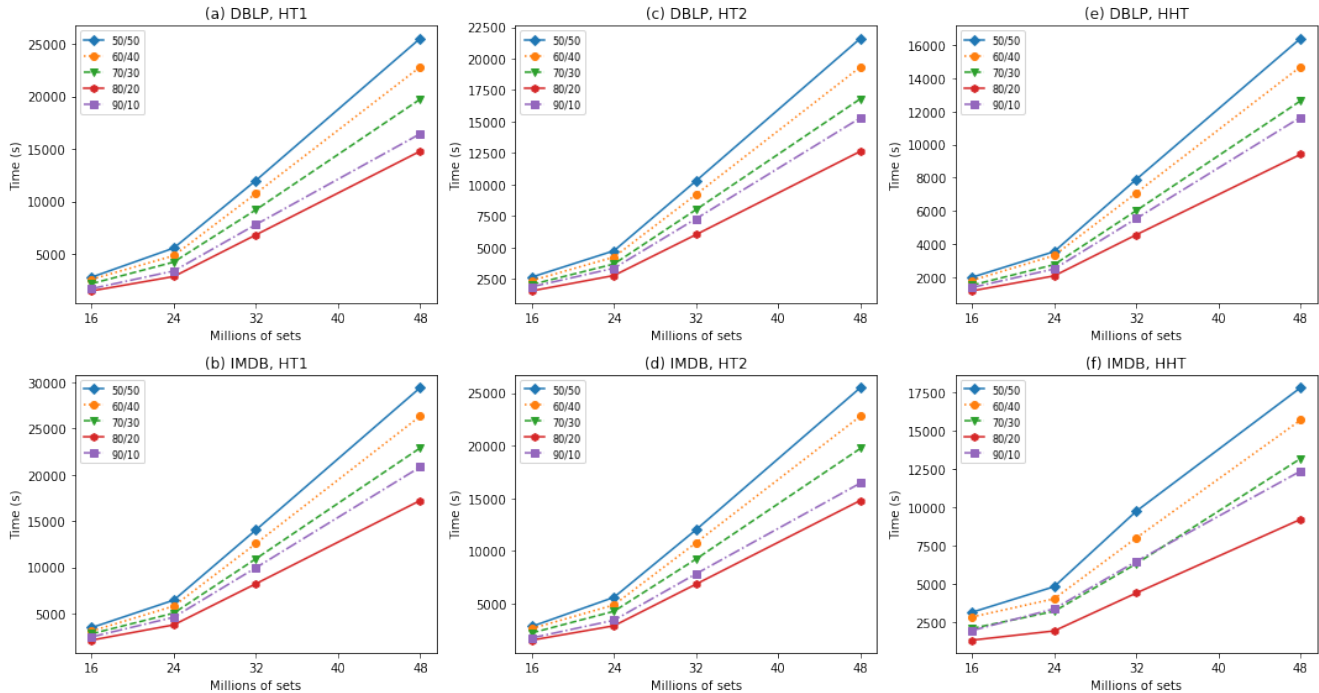


Figure 4. DSJoin^{gpu} runtimes for varying fractions of GPU workload and number of sets.

icates that a judicious workload splitting between CPU and GPU enables effective utilization of both processors, amortizing the overhead incurred by the PCIe bus.

5 Related Work

The efficient evaluation of SSJ queries has been actively investigated by the database community on various computing platforms. Most solutions are designed for CPU and assume memory-resident data [Sarawagi and Kirpal, 2004; Chaudhuri *et al.*, 2006; Bayardo *et al.*, 2007; Xiao *et al.*, 2011; Ribeiro and Härder, 2011]; some works also proposed extensions for disk-resident data [Bayardo *et al.*, 2007; Ribeiro and Härder, 2011]. Mann *et al.* [2016] presented an empirical evaluation of several CPU-based algorithms. More recently, Wang *et al.* [2017] proposed a new inverted list organization to improve the filtering step and exploited the fact that similar sets produce similar results to optimize the verification step. Fier and Freytag [2022] adapted existing SSJ algorithms to exploit multi-threading parallelism available on multicore CPUs.

Ribeiro-Júnior *et al.* [2016] introduced gSSJoin, the first exact SSJ algorithm designed for GPUs. Later on, the same authors presented sf-gSSJoin [Ribeiro-Júnior *et al.*, 2017], which we used in this article, and fgssjoin [Quirino *et al.*, 2017], which applies various filters to reduce the number of candidate pairs processed in the verification. In all three algorithms, SSJ is entirely executed on the GPU. Bellas and Gounaris [2019] presented an SSJ algorithm based on CPU-GPU coprocessing: in a multi-threading scheme, filtering is performed on the GPU while the whole verification is delegated to the GPU. Besides considering a distributed setting, we apply prefix filtering on the CPU during data partitioning and perform the remaining processing on both CPU and

GPU at each compute node. In subsequent work, Bellas and Gounaris [2021] proposed HySet, a framework to execute SSJ concurrently on CPU and GPU in a single-machine setting. Two strategies for distributing the workload between the processors are considered, and the best-performing one, called dichotomy, allocates fixed fractions of the workload to the CPU and GPU, as in our proposal. However, the best fractions for each processor are determined manually on a trial-and-error basis. In contrast, we use a cost model for automatically splitting the workload.

Another line of work considered the evaluation of SSJ on CPU-based HM clusters. An experimental comparison of ten distributed algorithms on this architecture is presented in [Fier *et al.*, 2018]. The best-performing algorithm, VernicaJoin [Vernica *et al.*, 2010], employs a data partitioning strategy based on prefix tokens as DSJoin [Oliveira *et al.*, 2017, 2018] and DSJoin^{gpu}. More recently, Sun *et al.* [2019] proposed a framework for adaptively selecting partition keys, on top of which global and local indexes are built to support similarity processing. Fier and Freytag [2021] employed a cost-based heuristic and a scaling mechanism to avoid intra-node data replication and recomputation. None of these previous works considered HT and HHT clusters.

There is a rich body of research addressing the heterogeneous execution of database queries on CPU-GPU systems. A recent, comprehensive survey is presented by Rosenfeld *et al.* [2022]. The authors categorize existing techniques along six dimensions w.r.t scheduling workload on CPU and GPU: processor usage, scheduling time, scheduling strategy, workload distribution, task granularity, and data partitioning. The former three dimensions cover aspects of schedule decisions, while the latter three focus on the characteristics of the tasks scheduled—in our context, a task refers to a list of set candidates evaluated in the verification step. We briefly discuss how our work fits this categorization in the follow-

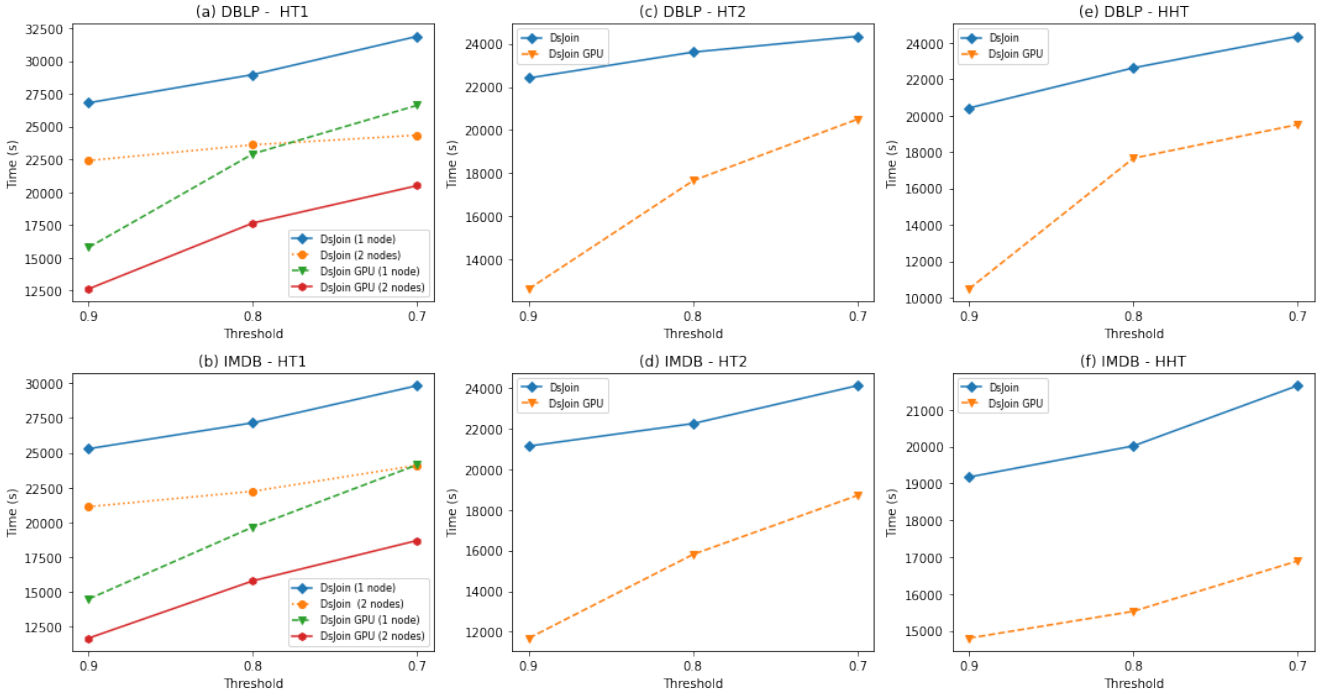


Figure 5. DSJoin^{GPU} vs. DSJoin.

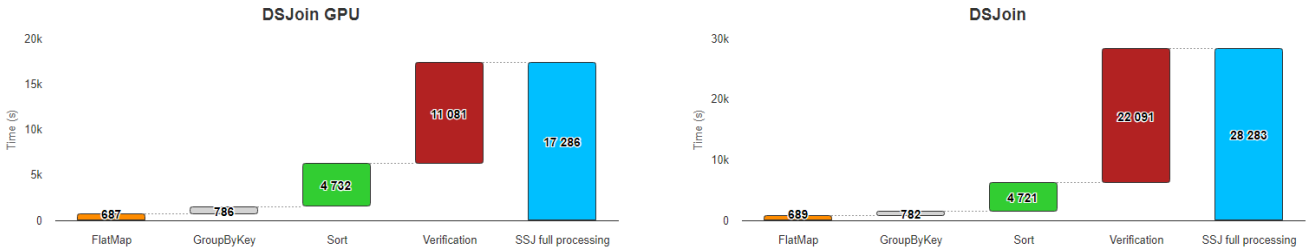


Figure 6. Runtime breakdown of DSJoin^{GPU} and DSJoin.

ing and refer the interested reader to the survey for specific related work on each dimension.

Regarding processor usage, we consider CPUs and GPUs as generic compute resources instead other work which uses them as specialized devices for specific purposes. Scheduling time determines when a task is assigned to a particular processor. The first version of DSJoin^{GPU} determines the type of processor to which the current list of candidates is sent during its execution; this approach is called dynamic in the survey. In the version presented in this article, scheduling is static, i.e., defined in the preprocessing step. Further, our scheduling strategy employs a cost model based on the relative throughput of each processor.

We focus on the execution of the SSJ in isolation. Hence, our workload distribution strategy is simply based on a single data partition. Interestingly, some proposals adopt a strategy that resembles the filtering-verification framework adopted by SSJ algorithms: result candidates are generated on one processor and verified on the other to produce final results. Here, we generate candidates on the CPU in the filtering step and verify them on both the CPU and GPU. Finally, our task granularity is defined by lists of candidates, as already mentioned, and, in turn, data partitioning is horizontal and produces arbitrarily-sized partitions.

6 Conclusion and Future Work

In this article, we presented an approach to efficiently answering SSJ queries on a heterogeneous cluster of compute nodes equipped with CPU and GPU. To deal with the asymmetry in computing power among processors, we devised a cost model based on their memory bandwidth to guide the workload distribution. Further, we applied this cost model to incorporate an SSJ algorithm designed for GPUs into a distributed algorithm, thus, coupling massive parallelism and distributed computing in a coprocessing fashion. We extensively evaluated the integrated algorithm, called DSJoin^{GPU}, on publicly available real-world datasets. Our experimental results have shown that the proposed cost model accurately captures the best workload distribution between CPU and GPU. Moreover, DSJoin^{GPU} is scalable and outperformed an existing distributed algorithm that executes entirely on CPU.

In future research, we will explore the parallelization of additional steps of SSJ. A clear candidate is the sorting of the candidate list, which have a significant impact on the runtime. We can also use the GPU on the pre-processing step of SSJ, including tokenization, set generation, and building the token-frequency and WD tables. We also intend to apply our cost model to enable CPU-GPU coprocessing on other SSJ algorithms, including the multithreading ones. To this end,

the Hyset framework of Bellas and Gounaris is complementary to our work: our cost model can be used to automate the workload splitting of Hyset, while this framework provides us with a common testbed for investigating different SSJ algorithms. Finally, we plan to evaluate our techniques on a wider spectrum of cluster configurations and study the impact of fast GPU interconnects on processing SSJ queries.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This research was partially supported by LaMCAD/UFG.

References

- Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *Proceedings of the International World Wide Web Conferences*, pages 131–140.
- Bellas, C. and Gounaris, A. (2019). Exact Set Similarity Joins for Large Datasets in the GPGPU Paradigm. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 5:1–5:10.
- Bellas, C. and Gounaris, A. (2021). HySet: A Hybrid Framework for Exact Set Similarity Join using a GPU. *Parallel Computing*, 104-105:102790.
- Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the IEEE International Conference on Data Engineering*, page 5.
- Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150.
- Fier, F., Augsten, N., Bouros, P., Leser, U., and Freytag, J. (2018). Set Similarity Joins on MapReduce: An Experimental Survey. *Proceedings of the VLDB Endowment*, 11(10):1110–1122.
- Fier, F. and Freytag, J. (2021). Scaling Up Set Similarity Joins Using a Cost-Based Distributed-Parallel Framework. In *International Conference on Similarity Search and Applications*, pages 17–31.
- Fier, F. and Freytag, J. (2022). Parallelizing Filter-and-verification based Exact Set Similarity Joins on Multi-cores. *Information Systems*, 108:101912.
- Mann, W., Augsten, N., and Bouros, P. (2016). An Empirical Evaluation of Set Similarity Join Techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647.
- Oliveira, D., Borges, F. F., and Ribeiro, L. A. (2017). Uma Abordagem para Processamento Distribuído de Junção por Similaridade sobre Múltiplos Atributos. In *Proceedings of the Brazilian Symposium on Databases*, pages 300–305.
- Oliveira, D. J. C., Borges, F. F., Ribeiro, L. A., and Cuzocrea, A. (2018). Set Similarity Joins with Complex Expressions on Distributed Platforms. In *Proceedings of the Symposium on Advances in Databases and Information Systems*, pages 216–230.
- Quirino, R. D., Ribeiro-Júnior, S., Ribeiro, L. A., and Martins, W. S. (2017). fgssjoin: A GPU-based Algorithm for Set Similarity Joins. In *International Conference on Enterprise Information Systems*, pages 152–161. SciTePress.
- Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, 36(1):62–78.
- Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2016). gSSJoin: a GPU-based Set Similarity Join Algorithm. In *Proceedings of the Brazilian Symposium on Databases*, pages 64–75. SBC.
- Ribeiro-Júnior, S., Quirino, R. D., Ribeiro, L. A., and Martins, W. S. (2017). Fast Parallel Set Similarity Joins on Many-core Architectures. *Journal of Information and Data Management*, 8(3):255–270.
- Rosenfeld, V., Breß, S., and Markl, V. (2022). Query Processing on Heterogeneous CPU/GPU Systems. *ACM Computing Surveys*, 55(1).
- Sarawagi, S. and Kirpal, A. (2004). Efficient Set Joins on Similarity Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM.
- Shanbhag, A., Madden, S., and Yu, X. (2020). A Study of the Fundamental Performance Characteristics of GPU and CPUs for Database Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1617–1632.
- Silva, L. R. M. and Ribeiro, L. A. (2022). Junções por Similaridade usando Processamento Distribuído e Paralelismo Massivo. In *Proceedings of the Brazilian Symposium on Databases*, pages 421–426. SBC.
- Sun, J., Shang, Z., Li, G., Bao, Z., and Deng, D. (2019). Balance-Aware Distributed String Similarity-Based Query Processing System. *Proceedings of the VLDB Endowment*, 12(9):961–974.
- Suri, S., Ilyas, I. F., Ré, C., and Rekatsinas, T. (2021). Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins. *Proceedings of the VLDB Endowment*, 15(3):699–712.
- Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set-similarity Joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 495–506.
- Wang, X., Qin, L., Lin, X., Zhang, Y., and Chang, L. (2017). Leveraging Set Relations in Exact Set Similarity Join. *Proceedings of the VLDB Endowment*, 10(9):925–936.
- Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-Duplicate Detection. *ACM Transactions on Database Systems*, 36(3):15:1–15:41.
- Xu, L., Butt, A. R., Lim, S., and Kannan, R. (2018). A Heterogeneity-Aware Task Scheduler for Spark. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 245–256.
- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28.