gSSJoin: a GPU-based Set Similarity Join Algorithm

Sidney R. Junior, Rafael D. Quirino, Leonardo A. Ribeiro, Wellington S. Martins

INSTITUTO DE INFORMÁTICA
UFG

Agenda

Introduction

Set similarity join returns all pairs of similar sets from a dataset. Sets are considered similar if the value returned by a set similarity function for them is not less than a given threshold.

Applications

- Integration
- Cleaning
- Plagiarism identification
- Data Mining

Current Algorithms

Most state-of-the-art algorithms uses a technique called *prefix filtering* to prune dissimilar sets by inspecting only a fraction of them.

Limitations

Prefix filtering is only effective at high threshold values. When the threshold value is low, the fraction needed to be inspected is bigger and the remaining sets needed to be verified after the pruning step is much higher.

Contributions

- A fine-grained parallel algorithm for both indexing the data and performing the set similarity joins.
- A highly threaded GPU implementation that takes advantage of intensive occupation, hierarchical memory, and coalesced memory access.
- A scalable multi-GPU implementation that exploits both data parallelism and task parallelism.
- Extensive experimental work with standard real-world textual datasets.

Data Representation

We map the strings to sets of tokens called *q-grams*. q-grams are sub-strings of length q obtained by sliding a window over the characters of the input string.

3-grams of *"gSSJoin"*: {*"gSS"*, *"SSJ"*, *"SJo"*, *"Joi"*, *"oin"*}

Formal Definition

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 $sim\,(r, s) = \tau\prime \geq \tau$.

Similarity Function

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.

Similarity Function

$$\textit{Jaccard Similarity}: \; J(r,s) = \frac{|r \cap s|}{|r \cup s|}$$

Overlap Constraint

A predicate involving the Jaccard similarity and a threshold τ can be equivalently rewritten into a set overlap constraint: $J(r,s) \geq \tau \iff |r \cap s| \geq \frac{\tau}{1+\tau}(|r| + |s|)$.

Prefix

For a threshold τ, we can identify all candidate matches of a given set r using a prefix of length $|r| - \lceil |r| \cdot \tau \rceil + 1$.

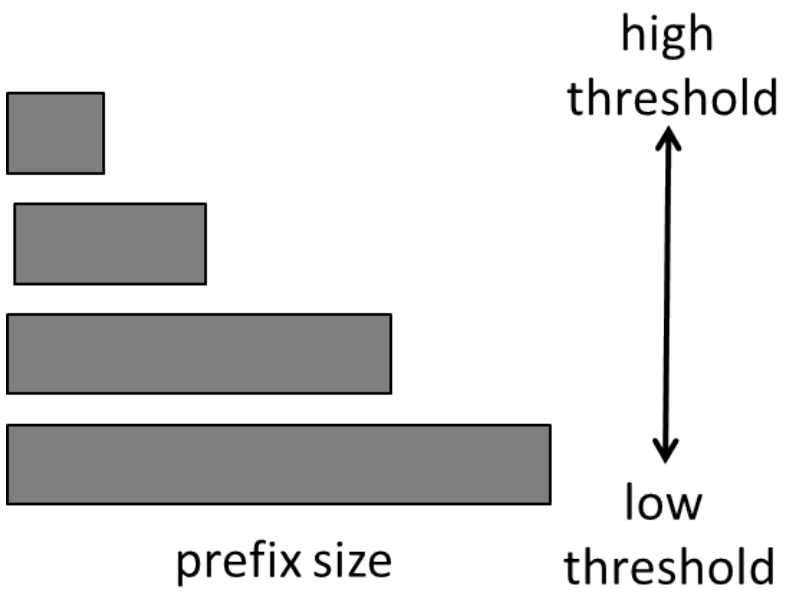
Current Algorithms

Algorithm 1: General set similarity join algorithm.

Input: A sorted set collection $\mathcal{C}$, a threshold τ
Output: A set S containing all pairs (r, s) s.t. $sim\,(r, s) \geq \tau$

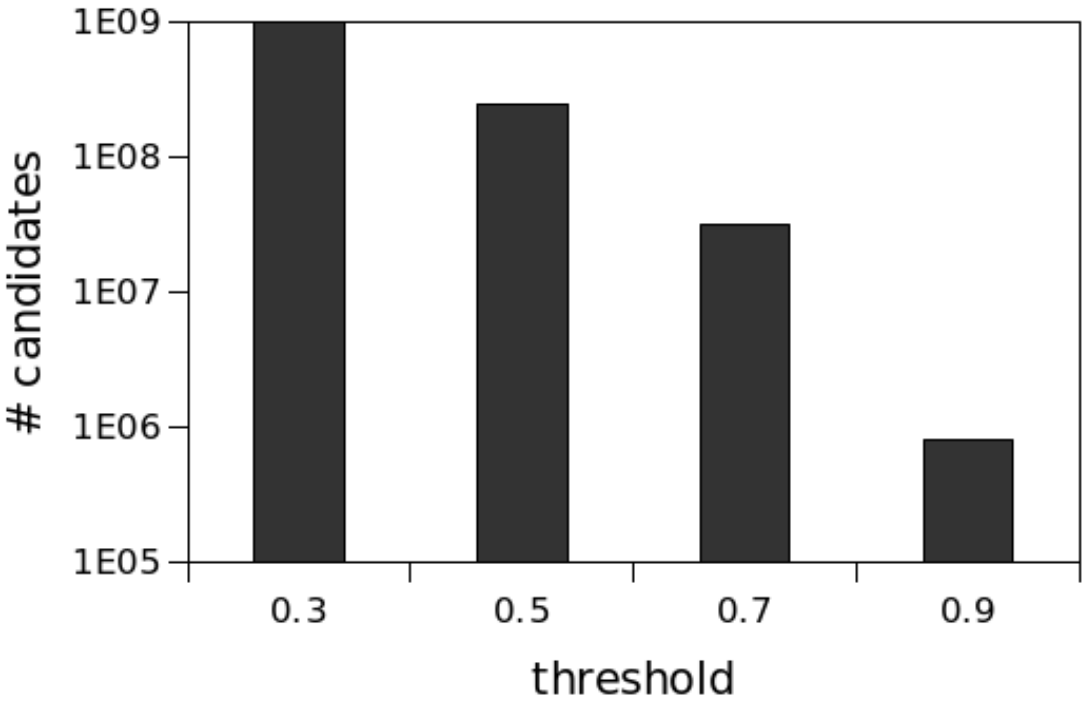
```
1 I_1, I_2, ... I_|U| ← ∅, S ← ∅
2 foreach r ∈ C do
3     foreach t ∈ pref_β (r) do
4         foreach s ∈ I_t do
5             if not filter (r, s)
6                 S ← S ∪ refine (r, s)
7         I_t ← I_t ∪ {r}
8 return S
```

Limitations of Current Algorithms

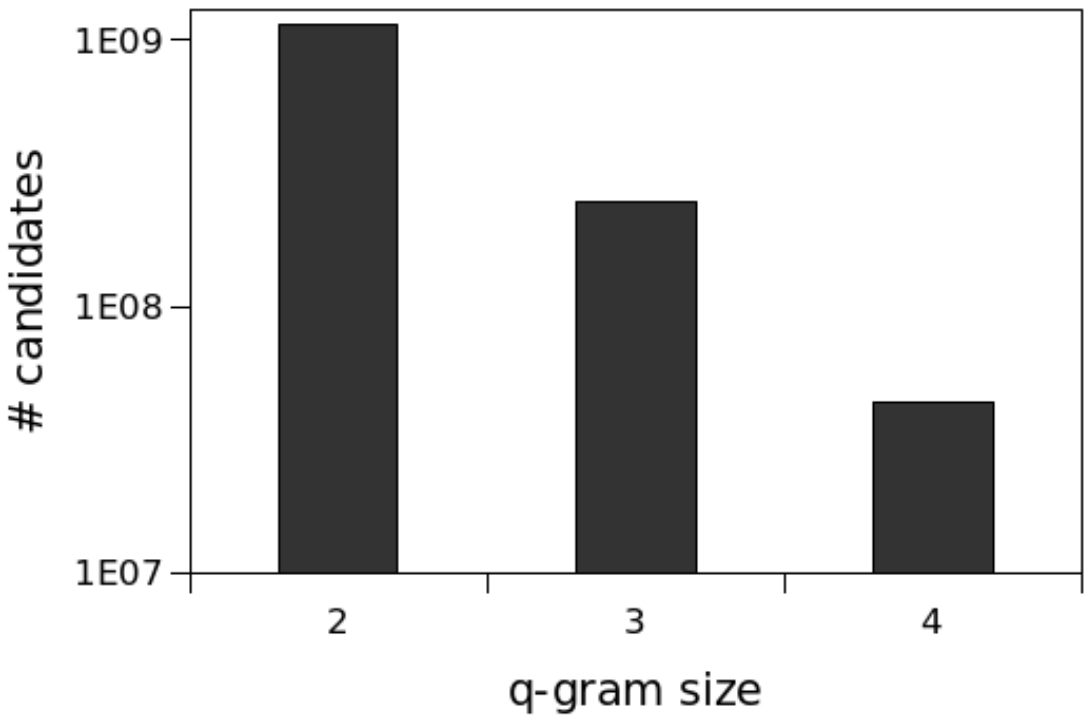


threshold effect

Limitations of Current Algorithms



Limitations of Current Algorithms



The gSSJoin Algorithm

- Exploits massive parallelism instead of filtering
- Builds an inverted index on the pre-processing phase
- Performs a set similarity search for each set

Pre-processing

Algorithm 2: $DataIndexing(E)$

input : token-set pairs in $E[0\,..\,|\mathcal{E}|-1]$.
output: $count$, $index$, $cardinality$, $invertedIndex$.

1 array of integers $count[0\,..\,|\mathcal{V}|-1]$ `// count array, initialized with zeros.`
2 array of integers $index[0\,..\,|\mathcal{V}|-1]$.
3 array of integers $cardinality[0\,..\,|\mathcal{S}|-1]$.
4 $invertedIndex[0\,..\,|\mathcal{E}|-1]$ `// the inverted index`
5 Count the occurrences of each token in parallel on the input and accumulates in $count$.
6 Perform an exclusive parallel prefix sum on $count$ and stores the result in $index$.
7 Access in parallel the pairs in E, with each processor performing the following tasks:
8 **begin**
9 Contribute on the cardinality computation of each set in $cardinality[s]$.
10 Store in $invertedIndex$ the entries corresponding to pairs in E, according to $index$.
11 **end**
12 **Return** the arrays: $count$, $index$, $cardinality$ and $invertedIndex$.

Set Similarity Search

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.

Multi-GPU

- The inverted index is created in all GPUs
- The search sets are divided between the GPUs

Multi-GPU

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.
```

Experimental Setup

We implemented gSSJoin using the CUDA Toolkit version 7.5. The experiments were conducted on a machine running CentOs 7.2.1511 64-bits, with 24 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.

Databases

Table : Dataset statistics.

Database	Number of sets	Max	Mean	Standard deviation
DBLP	100k	205	70	23.9
Netflix	200k	54	30.54	8.2

Performance Results

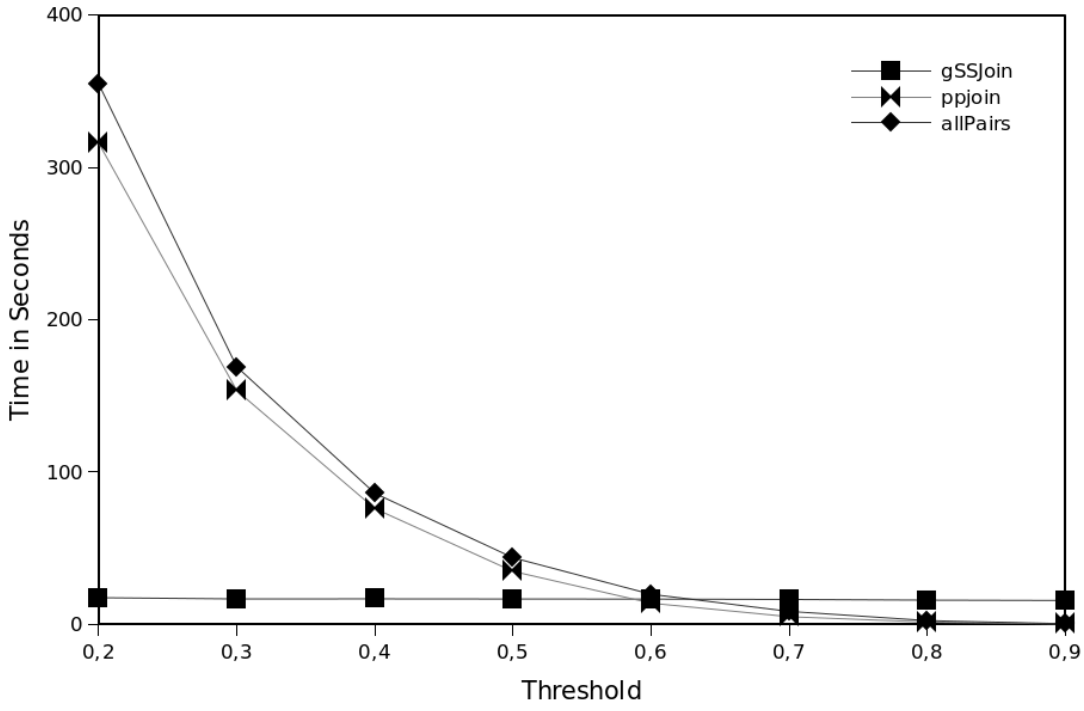


Figure : DBLP dataset using 3-grams.

Performance Results

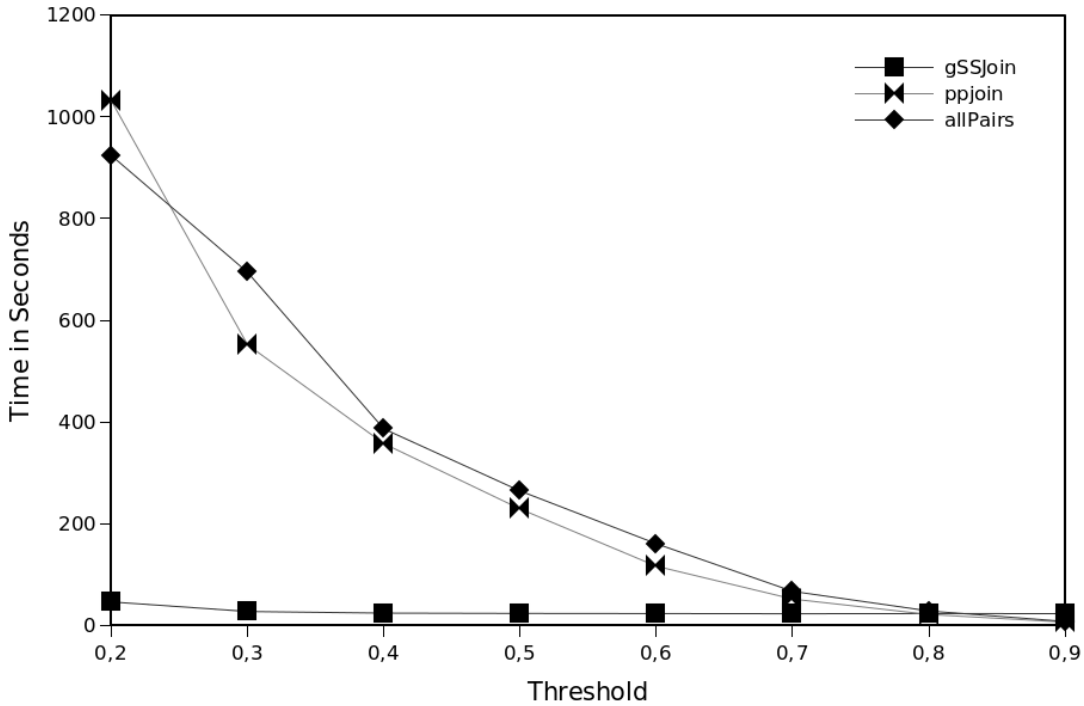


Figure : Netflix dataset using 3-grams.

Performance Results

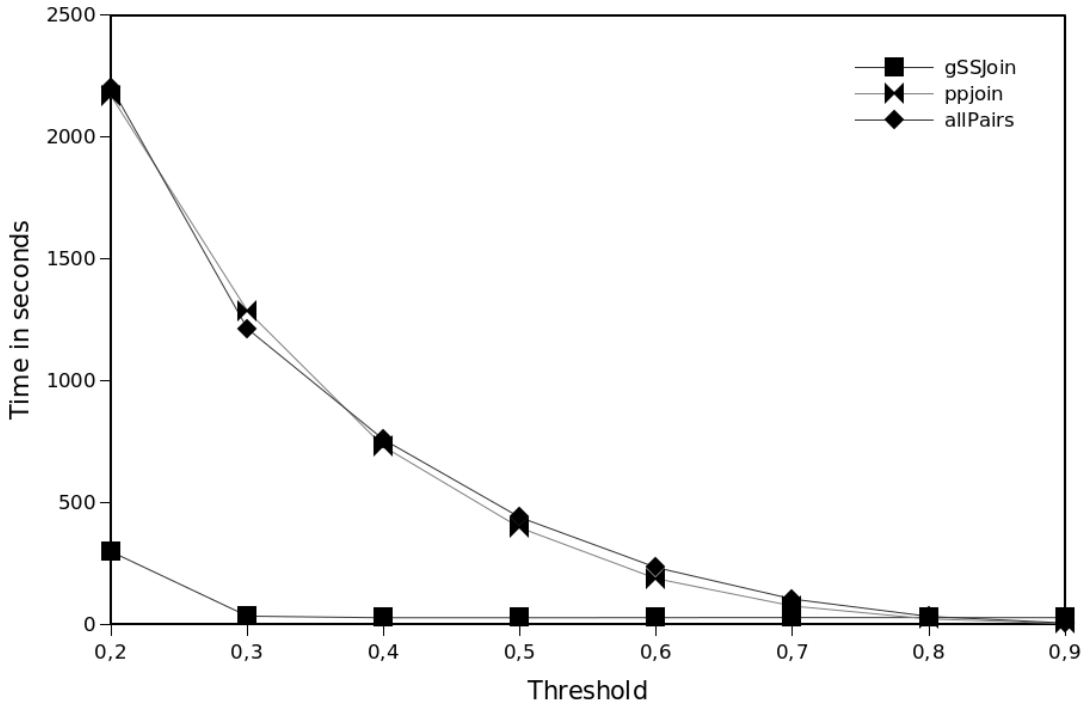


Figure : DBLP dataset using 2-grams.

Performance Results

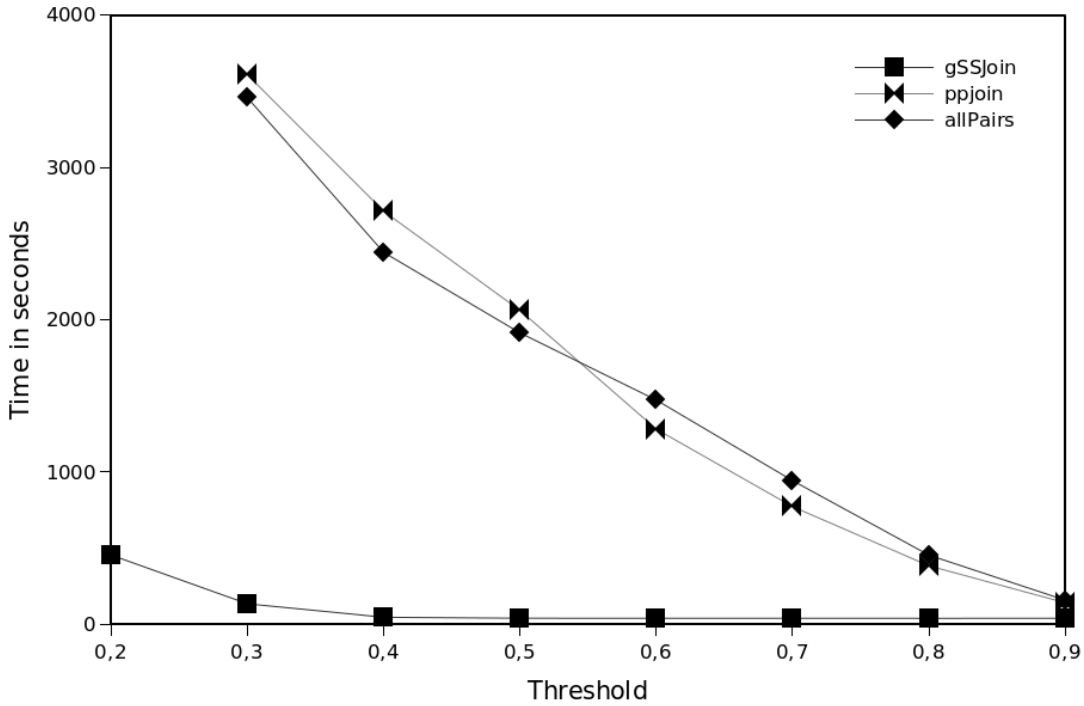


Figure : Netflix dataset using 2-grams.

Performance Results

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

Number of GPUs:	1	2	3	4
DBLP 2gram	27.56	14.37	10.4	7.2
DBLP 3gram	16.73	8.77	6	4.47
Netflix 2gram	41.10	21.63	14.37	11.2
Netflix 3gram	23.52	12.69	8.5	6.5

Related Work

- The state-of-the-art algorithms are intrinsically sequential [Sarawagi and Kirpal, Chaudhuri et al. 2006, Bayardo et al. 2007, Xiao et al. 2011, Ribeiro and Härder 2011, Wang et al. 2012].
- A distributed memory MapReduce framework to perform similarity joins was proposed by [Vernica et at. 2010].
- Parallel solutions such as [Cruz et al. 2015] perform approximate set similarity joins using MinHash and Locality Sensitive Hashing.

Conclusion

The gSSJoin algorithm has a much better performance than the state-of-the-art CPU-based algorithms when the threshold is low and the database is more uniform.

In future work, we plan to incorporate candidate filtering techniques to obtain even greater speed-ups.

Questions?

email: srjsoftware@gmail.com

References

[Cruz et al. 2015]

Cruz, M. S. H., Kozawa, Y., Amagasa, T., and Kitagawa, H. (2015). GPU Acceleration of Set Similarity Joins. In DEXA, pages 384–398.

[Bayardo et al. 2007]

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In WWW, pages 131–140.

[Chaudhuri et al. 2006]

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A primitive operator for similarity joins in data cleaning. In ICDE, page 5.

[Ribeiro and Härder 2011]

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

[Sarawagi and Kirpal 2004]

Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In SIGMOD.

References

[Vernica et al. 2010]

Vernica, R., Carey, M. J., and Li, C. (2010). Efficient Parallel Set Similarity Joins using MapReduce. In SIGMOD, pages 495–506.

[Xiao et al. 2011]

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. (2011). Efficient Similarity Joins for Near-duplicate Detection. TODS, 36(3):15.

[Wang et al. 2012]

Wang, J., Li, G., and Feng, J. (2012). Can We Beat the Prefix Filtering?: An Adaptive Framework for Similarity Join and Search. In SIGMOD, pages 85–96.

GPU Architecture