Parallelizing String Similarity Join Algorithms

Ling-Chih Yao and Lipyeow Lim

University of Hawaiʻi at Mānoa, Honolulu, HI 96822, USA
{lingchih,lipyeow}@hawaii.edu

Abstract. A key operation in data cleaning and integration is the use of string similarity join (SSJ) algorithms to identify and remove duplicates or similar records within data sets. With the advent of big data, a natural question is how to parallelize SSJ algorithms. There is a large body of existing work on SSJ algorithms and parallelizing each one of them may not be the most feasible solution. In this paper, we propose a parallelization framework for string similarity joins that utilizes existing SSJ algorithms. Our framework partitions the data using a variety of partitioning strategies and then executes the SSJ algorithms on the partitions in parallel. Some of the partitioning strategies that we investigate trade accuracy for speed. We implemented and validated our framework on several SSJ algorithms and data sets. Our experiments show that our framework results in significant speedup with little loss in accuracy.

1 Introduction

Given a string-based data collection and a similarity measure, the *string similarity join (SSJ)* problem takes as input two sets of strings and returns all pairs of strings from the two data sets where their similarity value are above a user-specified threshold. Many real-world applications apply SSJ for data cleaning, duplicate detection, data integration and entity resolution. Other less obvious applications include nearest-neighbor-like queries in web search, social media, and even recommender systems. The SSJ problem is well studied in the literature with a plethora of algorithms proposed [15, 5, 9, 3, 8].

With the advent of big data, a modern challenge in SSJ processing is the rapid growth of the size of the data sets. For example, in 2010, the size of the Google N-gram dataset has exceeded 1 trillion data items [13], and the amount of log data that Facebook receives each day exceeds 6TB [2]. A natural question is how to parallelize SSJ processing to handle the large data sets we have today. Two recent studies [13, 12] have utilized MapReduce and Spark for parallel SSJ processing. Nevertheless, there is a large body of work on sequential SSJ algorithms with new ones added every year. Parallelizing any complex algorithm is hard in general and parallelizing each of those sequential SSJ algorithms is clearly a challenge with or without using frameworks like MapReduce and/or Spark.

In this paper, we investigate a parallelization approach for existing SSJ algorithms that first partitions the data, run the existing (in-memory) SSJ algorithms on each partition in parallel, and finally merge the results. In particular, we propose a variety of data partitioning strategies and investigate their impact on parallelizing existing SSJ algorithms. The contribution of this paper are as follows.

– We reduce the unnecessary computation by partitioning dissimilar data into different partitions.
– We show that partitioning strategies can achieve significant speedup with little loss in accuracy.
– We can parallelize any sequential string similarity algorithm without modifications on the original algorithms with our data partitioning strategies.
– Our experimental evaluation show that our partitioning strategies achieve excellent performance on several real-world data sets.

The rest of paper is organized as follows. We give a problem definition and related work in Section 2 and Section 3. We describe our parallelization framework and the different strategies of data partitioning in Section 4. We present the experiment results in section 5. And we conclude the paper in Section 6.

2 Preliminaries

Given two data sets R and S, similarity function sim, and threshold δ, the **string similarity join** operator returns all similar pairs (x, y), where $x \in R$, $y \in S$, and $sim(x, y) > \delta$. We define a string-based record as a set of tokens by mapping each unique string in the data sets to a unique token. We use **token-based similarity functions** to compute the similarity of two records. The popular token-based functions are Jaccard, Cosine, Dice, and Overlap (where x and y denote the input sets of string tokens):

Overlap: $\text{Olp}(x, y) = |x \cap y|$ Jaccard: $\text{Jac}(x, y) = \frac{|x \cap y|}{|x \cup y|}$
Dice: $\text{Dice}(x, y) = \frac{2|x \cap y|}{|x|+|y|}$ Cosine: $\text{Cos}(x, y) = \frac{|x \cap y|}{\sqrt{|x||y|}}$

For example, consider the strings 'Yesterday rained during the afternoon' and 'Yesterday it rained during the afternoon'. We first break them into string tokens x: {Yesterday, rained, during, the, afternoon} and y: {Yesterday, it, rained, during, the, afternoon}, respectively, then we can measure their similarity by similarity functions: $Olp(x, y) = 5$, $Jac(x, y) = \frac{5}{6}$, $Dice(x, y) = \frac{10}{11}$, and $Cos(x, y) = \frac{5}{5.477}$.

3 Related work

There are many SSJ algorithms proposed due to the inefficiency of determining the similar pairs by examining every string pair. The proposed algorithms are classified into two categories: approximate SSJ and exact SSJ.

Approximate SSJ algorithms trade the accuracy for the efficiency by reducing the dimensionality of the input vectors. *Local-Sensitive Hashing (LSH)* is a common technique for reducing the input's dimensionality by hashing input data.In 2012, Satuluri and Parthasarathy [10] utilize a Bayesian approach to optimize a LSH-based SSJ algorithm. In 2017, Sohrabi and Azgomi proposed a parallel SSJ algorithm based on the LHS techniques [11].

Exact SSJ, which output the exact result pairs, has received more attention partly because approximate SSJ algorithms often produce non-trivial errors in practical applications. The common technique is generating signatures and then using inverted index to filter dissimilar pairs. AllPair [1], PPJoin+ [16], and AdaptJoin [14] are popular prefix-filtering algorithms that generate signatures based on the prefix of strings. Partition-based algorithms generate signatures by partitioning strings into non-overlapping subsets [6, 9].

Sequential SSJ algorithms are inadequate for very large data sets such as data from network monitoring, internet of things, internet-scale applications. Vernica et. al. proposed an efficient parallel exact SSJ algorithm based on Hadoop MapReduce [13] in 2010. It partitions data by their prefix and process the partitions in parallel. Sun et. al. proposed Dima, a similarity-based query system on Spark, in 2017 [12]. It partitions data by global and local index, so the target range and data transmission can be decreased.

There are currently many (sequential) SSJ algorithms with new algorithms invented almost every year. Parallelization of the sequential string similarity algorithms remains a very challenging task – many of them do not fit nicely onto the Map-Reduce framework utilized by Hadoop and Spark. Instead of parallelizing each existing sequential SSJ algorithm or inventing a new parallel SSJ algorithm, our work investigate data partitioning (and hence parallelization) strategies for parallel processing of the SSJ problem using *any existing* sequential SSJ algorithm as a sub-routine.

4 Parallelizing String Similarity Join

In this paper, we propose to parallellize the SSJ operation by first partitioning the data into partitions or buckets in a one-time offline phase. Each partition is stored locally in a compute node. In the online SSJ processing phase, we execute the SSJ algorithm at each compute node in parallel. We then consolidate the results from the compute nodes to be returned to the user. A key idea of our framework is to avoid doing cross-partition joins by leveraging a suitable partitioning strategy to partition the data so that similar records will be co-located in the same partition. Hence, there will be some loss of accuracy and our framework will trade-off accuracy for speed. In general we only need to partition data once and the same partitions are reused for multiple queries. Next, we describe the partitioning strategies that are the thrust of this paper.

4.1 Partitioning by the First Token

For this partitioning method, we determine the partition identifier (ID) of a record by using the first token of each record. The assumption is that similar record pairs are very likely to have same first token. Since the number of distinct first tokens can be quite large, we use a variety of techniques to limit the number of partitions: (1) compute a hash value of the first token, (2) use the first letter of the first token, and (3) use the first k letters of the first token. In our experiment, we use the first letter (strategy *First*) , and the first two letters of the first token (strategy *Second*).

4.2 Partitioning by Keyword

Previous research [16, 6, 1] have shown that low frequency tokens have higher power to filter dissimilar data. In this partitioning strategy we use the token with least frequency in each record to determine the partition ID. We first build a dictionary containing all tokens in the data set and their occurrence frequency. For each record, we use the dictionary to find the token with the smallest frequency and use that token to compute the partition ID.

4.3 Partitioning by the Prefix

Prefix filtering is a popular method to filter dissimilar data [16, 6, 1, 14] in SSH algorithms. It is proposed in SSJ algorithm SSJoin [4] and extended to prefix-based framework in ALLPair algorithm [1]. The key idea is to first sort the tokens in each record according to some predetermined order and then if a pair of records do not share any token in their prefix, they must be dissimilar. For this partitioning strategy, we first sort the tokens in each record, and then use a prefix of the record to compute the partition ID.

5 Experiments

For our experiments, we implemented our parallelization framework using the Message Passing Interface (MPI) [7] and Python. For the sequential string similarity algorithm to be executed on each compute node, we run the executable file of the original author's implementation of the algorithm for PPjoin+ [16] and AdaptJoin [14]. We conducted our experiments on a Cray CS300 high performance computing (HPC) clusters with 276 total nodes. Each node contains two 10-core Intel Ivy Bridge, 2.8GHz processors and 118GB usable RAM. We performed extensive experiments using three data sets, two existing SSJ algorithms, different numbers of cores and different thresholds. Due to the space limitation, we only show part of the results.

We use three real-world data sets in our experiments: DBLP, a collection of author names and titles from computer science bibliography; QueryLog, query strings randomly selected from AOL Query Log [14]; and Enron, a collection of the titles and bodies from Enron emails [14].

Table 1 shows the basic information of the data sets. Column Records shows the number of records; columns l_{avg}, l_{min} and l_{max} show the average, minimal, and maximal token size, respectively; columns PPJoin+ and AdaptJoin show the sequential running time of these two algorithms; and column Result Pairs

	Records	l_{avg}	l_{min}	l_{max}	File Size	PPJoin+	AdaptJoin	Result Pairs
DBLP	3,422,478	16.48	6	233	370M	-	60.57s	131,743
QueryLog	1,208,844	20.44	1	500	26m	-	6.44s	20,603
Enron	517,431	133.57	1	3162	435M	-	65.62s	1,332,622

Table 1. Information on the data sets. The result pairs are for threshold $\delta = 0.9$.

	First	Second	Keyword	Prefix
DBLP	7.1 (98.4%)	9.4 (98.4%)	8.2 (96.7%)	3.6 (98.1%)
QueryLog	6.4 (67.9%)	2.6 (66.5%)	8.1 (96.8%)	4.4 (98.9%)
Enron	2.5 (97.6%)	3.5 (97.6%)	5.5 (85.3%)	3.9 (93.0%)

Table 2. The speedup and accuracy in parenthesis of our parallel SSJ framework using different strategies on 11 processors. The AdaptJoin SSJ algorithm with Jaccard threshold 0.9 was used.

show the similar pairs with Jaccard threshold 0.9. Note that PPjoin+ did not generate any results because it exited with a segmentation fault.

Table 2 shows the speedup and accuracy of our parallel SSJ framework. We can see *Keyword* worked best on average with different data sets, and the best speedup happened in *Second* on DBLP. *First* worked worse than *Keyword* but the data partitioning time is much faster. Prefix is slowest (because each record may partition to more than one partitions and cause duplicate computation) but the accuracy is the most stable.

First and *Second* worked well on DBLP and Enron with accuracy more than 97%; but they worked worse on QueryLog with accuracy around 66%. It is because DBLP starts with authors' name and Enron start with the titles of emails, so their first tokens have more filter power. *Keyword* worked worse on Enron as the average token size in Enron is 133.51, one keyword is not enough to filter dissimilar data. Note that accuracy of *Prefix* cannot reach 100% since we only retrieved two tokens instead of whole prefix (for reducing the data partitioning time.)

6 Conclusion

In this paper, we studied data partitioning strategies for parallelizing existing sequential SSJ algorithms. We partitioned the data set into partitions using a variety of strategies based on the first k letters of the first token, the least frequent keyword, and the prefix of each record. These partitions were then processed in parallel with existing SSJ algorithms and the results merged together for the user. We performed extensive experimental validation and observed that different data sets achieve the best accuracy and performance with different strategies. Users can choose different strategies for different data sets depending on the filtering power of the partitioning strategies on that data set. The right strategy can result in significant speedup with little loss of accuracy.

References

1. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW. pp. 131–140. ACM (2007)
2. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In: SIGMOD. pp. 975–986. ACM (2010)
3. Bocek, T., Hunt, E., Stiller, B., Hecht, F.: Fast similarity search in large dictionaries. University (2007)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE. pp. 5–5. IEEE (2006)
5. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB. vol. 23, pp. 426–435 (1997)
6. Deng, D., Li, G., Wen, H., Feng, J.: An efficient partition based method for exact set similarity joins. VLDB 9(4), 360–371 (Dec 2015)
7. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al.: Open mpi: Goals, concept, and design of a next generation mpi implementation. In: European Parallel Virtual Machine/Message Passing Interface Users Group Meeting. pp. 97–104. Springer (2004)
8. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D., others: Approximate string joins in a database (almost) for free. In: VLDB. vol. 1, pp. 491–500 (2001)
9. Li, G., Deng, D., Wang, J., Feng, J.: Pass-join: A partition-based method for similarity joins. VLDB 5(3), 253–264 (2011)
10. Satuluri, V., Parthasarathy, S.: Bayesian locality sensitive hashing for fast similarity search. VLDB 5(5), 430–441 (Jan 2012)
11. Sohrabi, M.K., Azgomi, H.: Parallel set similarity join on big data based on Locality-Sensitive Hashing. Science of Computer Programming 145, 1–12 (2017)
12. Sun, J., Shang, Z., Li, G., Deng, D., Bao, Z.: Dima: A distributed in-memory similarity-based query processing system. VLDB 10(12), 1925–1928 (2017)
13. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using MapReduce. In: SIGMOD. pp. 495–506. ACM (2010)
14. Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: SIGMOD. pp. 85–96. ACM (2012)
15. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. VLDB 1(1), 933–944 (2008)
16. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW. pp. 131–140. ACM (2008)