Index-based R-S Similarity Joins

Spencer S. Pearson, Yasin N. Silva

Arizona State University, Glendale, AZ, USA
{sspearso, ysilva}@asu.edu

Abstract. Similarity Joins are some of the most useful and powerful data processing operations. They retrieve all the pairs of data points between different data sets that are considered similar within a certain threshold. This operation is useful in many situations, such as record linkage, data cleaning, and many other applications. An important method to implement efficient Similarity Joins is the use of indexing structures. The previous work, however, only supports self joins or requires the joint indexing of every pair of relations that participate in a Similarity Join. We present an algorithm that extends a previously proposed index-based algorithm (eD-Index) to support Similarity Joins over two relations. Our approach operates over individual indices. We evaluate the performance of this algorithm, contrast it with an alternative approach, and investigate the configuration of parameters that maximize performance. Our results show that our algorithm significantly outperforms the alternative one in terms of distance computations, and reveal interesting properties when comparing execution time.

1 Introduction

The Similarity Join (SJ) is one of the most useful and studied data processing operators. It has applications in many different situations or domains, such as multimedia applications, sensor networks, marketing analysis, and many others. Many different implementations and algorithms for SJ have been proposed, ranging from on-the-fly algorithms to index-based techniques. Index-based algorithms have the potential to significantly reduce execution time since they store pre-computed information that can be used during query execution. One such technique is the eD-Index [1]. This index enables efficient similarity-aware operations such as similarity search and Self-SJ. In this paper, we present an algorithm that significantly extends this technique to support generic SJ queries over two relations. The main contributions of our work are:

- We implemented the Range Query Similarity Join (RQ-SJ) algorithm using successive similarity search operations for the case of SJ with two relations. This technique was previously proposed in [2] for the case of Self-SJ only.
- We designed and implemented an efficient algorithm, i-SimJoin to extend eD-Index to support SJ operations over two relations using only the individual indices.
- We evaluated the performance of i-SimJoin and RQ-SJ. Our preliminary results show that i-SimJoin significantly outperforms the alternative one in terms of distance computations and interesting properties when comparing execution time.

- We explore ways to tune the eD-Index parameters to improve performance.

The remaining part of the paper is organized as follows. Section 2 presents the related work. Section 3 gives a brief overview of the indexing structures we used in our algorithms, the RQ-SJ algorithm, and a detailed explanation of our i-SimJoin algorithm. Section 4 presents the performance evaluation of the i-SimJoin and RQ-SJ algorithms. Section 5 presents the conclusions and future work directions.

2 Related Work

Significant work has been carried out on the study of Similarity Joins. Much of this work has focused on standalone operators – both index-based and dynamic (on-the-fly) – while some has focused on implementing Similarity Join operators inside of database systems. The distance range join (retrieves all pairs whose distances are smaller than or equal to μ) is one of the most studied Similarity Join types [1,2,3,4,5,6,7,8]. This is the Similarity Join type focused on in this paper. Of the non-index-based approaches, some of the most relevant algorithms are Epsilon Grid Order (EGO) [4], Generic External Space Sweep (GESS) [5], and Quickjoin [6]. These algorithms dynamically partition and cluster the data into smaller, easier to process subsets in such a way that all similar pairs are still captured by the algorithm. The index-based approaches include such algorithms as Pass-Join [7], an algorithm proposed for string data, and the D-Index [2, 3], eD-Index [1] and List of Twin Clusters (LTC) [8], which are indices that can apply to any metric space. Pass-join [7] partitions strings into substrings and used inverted indices in order to efficiently prune dissimilar pairs. LTC [8] is an indexing approach that constructs a combined index for both datasets involved in the Similarity Join. This indexing structure consists of clusters of data points within a fixed radius of given reference points. This structure allows Similarity Join queries with a μ less than or equal to the radius of the clusters to be easily computed. An important disadvantage of this approach is the need to build joint or combined indices for every pair of datasets that can be joined. The D-Index [3] and eD-Index [1] construct an index structure based around separate buckets arranged in a hierarchical structure of levels. This index-structure allows for efficient similarity search and Self-Similarity Join queries. Our work extends on the D-Index and eD-Index, and focuses on algorithms to utilize the eD-Index functionality to efficiently perform Similarity Joins between two relations in metric spaces.

3 The i-SimJoin Algorithm

3.1 The eD-Index

The structure of the D-Index and its extension, the eD-Index, are detailed in [3] and [1] respectively. In brief, the eD-Index makes use of multiple levels where each level is organized into separable buckets and an exclusion set. The top level partitions the initial dataset. Each subsequent level after the first is created by partitioning the exclusion set of the previous level. Separable buckets are constructed by picking *n* pivots and a radius *d* from each pivot. *d* can be different for each pivot and is calculated when constructing the index so as to attempt to balance the number of tuples in each separable bucket. *n* can also vary between levels. Objects are placed in the appropriate separable bucket or the exclusion set based on their distance from the pivots and the global parameters ρ and ε, which determine the maximum query radius the eD-Index can be used to answer efficiently. Objects with a distance between $d + \rho$ and $d - \rho$ from a pivot are placed into the exclusion set. All other objects are placed into a separable bucket determined by the objects' distances from all pivots on that level. All objects with a distance of between $d \pm \rho$ and $d \pm (\rho + \varepsilon)$ from a pivot are duplicated into the exclusion set in addition to being placed in a separable bucket.

3.2 RQ-SJ: Range Query Similarity Join

The Range Query Similarity Join is an algorithm proposed in [2] for the case of Self-SJ only. As part of our work, we implemented and evaluated the performance of this algorithm for the case of SJ for two relations. This algorithm applies successive similarity search operations over the indexed dataset *R*, using all elements of the dataset *S* as the targets of the similarity searches. For each object *s* in *S*, the output is the collection of all objects in *R* that are within μ of *s*.

3.3 i-SimJoin: Index-based Similarity Join

i-SimJoin is an algorithm for performing Similarity Join operations over two datasets indexed using the D-Index (individual indices). The indices are constructed so that they share the same index structure – that is, that the index for relation S uses the same number of levels and the same pivots for each level as relation R does. This allows the indices to be treated as the same logical index containing two separate relations while maintaining the index of each relation as a separate structure. On this logical index, we can apply an extension of a Self SJ operation such as the sliding window algorithm proposed in [1] with the added modification of awareness of which relation the tuples originally came from. This last modification ensures that only matches of pairs between both relations R and S will be returned.

To create the indices for the relations, an index is first created for relation R. The index structure generated from this is then used to create the index for relation S. This allows for the index of relation R to be used for Similarity Join queries with relation S, while still allowing the index on relation R to be independently used for other similarity-aware queries. Other approaches to create logical indices over the two relations while maintaining independent physical indices is a task for future work.

The i-SimJoin algorithm consists of two routines. First, we process the indices simultaneously, treating the corresponding buckets as a logical combined bucket as shown in Algorithm 1. Algorithm 2 is the algorithm that is run on each combined bucket. The getNextObject() function returns the next object in the combined bucket, ordered by the pre-computed distance from the object to the pivot. *upObject* and *loObject* are pointers to the current objects from each relation being compared – the

```
iSimJoin(indices, mu)
Input: indices (logical combined indices from relations), mu (query radius)
Output: all the results of the Similarity Join operation $R \bowtie_{\theta\mu(r,s)} S$

for each CombinedLevel L in indices
 for each CombinedBucket b in L
  b.iSimJoin_bucket(mu)
 end for
end for
indices.exclusionSet.iSimJoin_bucket(mu)
```

Alg. 1. iSimJoin

```
iSimJoin_bucket(mu)
Input: mu (query radius)
Output: all the results of the Similarity Join operation $R \bowtie_{\theta\mu(r,s)} S$ for one logical CombinedBucket

Object loObject = getNextObject()
Object upObject = loObject
while(upObject.relation == loObject.relation)
 upObject = getNextObject()
end while
if(loObject.relation == R)
 markR = loObject
 markS = upObject
else
 markR = upObject
 markS = loObject
end if
while(upObject != NULL)
 while(upObject.distance – loObject.distance > mu)
  loObject = loObject.next()
 end while
 if(loObject.relation == R)
  markR = loObject
 else
  markS = loObject
 end if
 while(loObject.distance <= upObject.distance)
  if(loObject.distance == upObject.distance && loObject.relation == S)
   break
  end if
  if(dist(loObject, upObject) <= mu
   report (loObject, upObject)
  end if
 end while
 upObject = getNextObject()
 if(upObject.relation == R)
  loObject = markS
 else
  loObject = markR
 end if
end while
```

Alg. 2. iSimJoin_bucket

upObject is the highest-ordered of the two objects, while the *loObject* iterates through the current sliding window. A marking system is used to correctly slide the window through the combined bucket.

Lines 1-12 of the iSimJoin_bucket algorithm are the initial setup of the data structures and the marking system. Lines 14-21 advance the rear of the window as it slides through the bucket and marking the new *loObject* appropriately. Lines 22-29 report all similar matches in the current window. Additional checks are done here to prune out dissimilar pairs before performing the final distance calculation on the candidate matches. The check at line 23 ensures that no match will be added twice if the current *upObject* and *loObject* have the same pre-computed distance. Lines 30-36 advance the *upObject* to the next element in the bucket and set *loObject* to the correct marked position for its relation.

4 Performance Evaluation

We implemented the i-SimJoin algorithm as a stand-alone application written in C++. In this section, we present preliminary results comparing the i-SimJoin algorithm to the Range Query (RQ-SJ) algorithm over two relations in terms of number of distance computations and execution time.

All experiments are performed on an Intel Core I-5 1.70 GHz 4-core machine with 6GB of RAM running Linux (OpenSUSE 12.3 64-bit) as the operating system. The dataset used for this experiment is a synthetically generated, 10-dimensional vector dataset with randomly-generated values for each dimension ranging from 0 to 100. This dataset contains 100K tuples per relation, or 200K in total. The Euclidean distance function is used to calculate distances between objects.

The strategy taken for constructing each index was to choose a number of pivots for each level such that the number of objects in the largest separable bucket in that level fell within the range of 5,000 to 10,000 objects, resulting in an index structure with 3 levels and 9 pivots. The value of ρ was 0.5% of the maximum distance and the value of ε was 1.0% of the maximum distance.

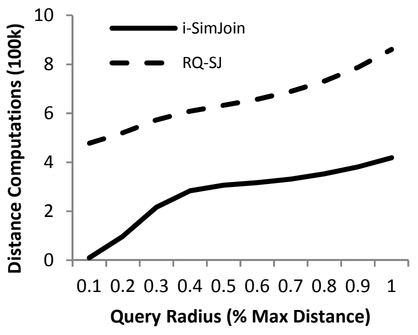

Fig. 1. Comparing Distance Computations while Increasing Query Radius

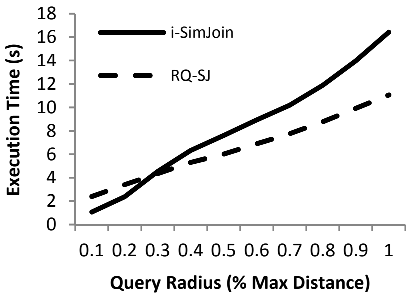

Fig. 2. Comparing Execution Time while Increasing Query Radius

Fig. 1 presents the number of distance calculations needed for each algorithm as the query radius increases. i-SimJoin requires a significantly lower amount of distance computations than the RQ-SJ algorithm, ranging from only 2.1% of the distance computations of the RQ-SJ algorithm at a query radius of 0.1% of the maximum distance, to 48.7% at a query radius of 1.0%.

Fig. 2 presents the execution time of the algorithms, with the execution time of i-SimJoin being comparable to that of the Range Query algorithm. The execution time of i-SimJoin ranges from 44.8% when the query radius is low to 148.3% of that of the Range Query algorithm. Note that i-SimJoin performs better than RQ-SJ in terms of execution time when the number of distance computations required for i-SimJoin is very low compared to that of RQ-SJ.

As part of our initial tests, we also compared the original Self-SJ algorithms proposed in [1] (range query and sliding window algorithms) and extended in this paper. We used the described 10D vector dataset and Euclidean distance function. The results were very similar to the ones reported in this paper for the case of SJ over two relations, i.e., while the sliding window technique performs significantly less distance computations, both algorithms have similar execution times. Note that the work in [1] only reports the number of distance computations and not the execution times.

The comparison of these approaches contrasting both distance computations and execution time is actually quite revealing. These results highlight the fact that the overhead required in processing these algorithms is a significant factor in the execution time. For instance, the i-SimJoin algorithm performs many tests to prune out pairs that are not in the result set. Although these checks do not necessarily involve distance computations, they still contribute to the processing that needs to be done. Since the results reported in this paper were obtained from a 10-dimensional dataset, the distance computations involved were not highly expensive. More complicated distance functions over more complex data types can significantly increase the complexity of the distance computations, and this would be expected to result in the number of distance computations being more significant in terms of the execution time. While i-SimJoin is expected to outperform RQ-SJ for complex data types and distance functions, it is also important to observe that the simple RQ-SJ algorithm can be the most efficient approach for simple data types and distance functions.

5 Conclusions

This paper presents i-SimJoin, an algorithm to perform Similarity Joins on two relations using physically independent indexing structures. Our performance evaluation shows that i-SimJoin requires far fewer distance calculations than an alternative SJ algorithm, and has a comparable execution time. Our future work will include: (1) extensive performance evaluations of more complex data types and distance functions to investigate how this affects the execution times of i-SimJoin and RQ-SJ, and (2) generalization of the i-SimJoin algorithm to the case of multiple SJ predicates.

References

1. V. Dohnal, C. Gennaro, P. Zezula, Similarity join in metric spaces using eD-Index, in: Proceedings of the 25th European Conference on IR Research, ECIR'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 452-467
2. V. Dohnal, C. Gennaro, P. Savino, P. Zezula, Similarity join in metric spaces, in: Proceedings of the 25th European Conference on IR Research, ECIR '03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 452-467.
3. V. Dohnal, C. Gennaro, P. Savino, P. Zezula, D-Index: Distance searching index for metric data sets, in: Multimeda Tools and Applications 21 (2003) 9-33.
4. C. Böhm, B. Braunmüller, F. Krebs, H.-P. Kriegel, Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD '01, ACM, New york, NY, USA, 2001, pp. 379-388.
5. J.-P. Dittrich, B. Seeger, Gess: A scalable similarity-join algorithm for mining large data sets in high-dimensional spaces, in: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, ACM, New York, NY, USA, 2001, pp. 47-56.
6. E. H. Jacox, H. Samet, Metric space similarity joins, ACM trans. Database Syst. 33 (2008) 7:1-7:38.
7. Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. Pass-join: a partition-based method for similarity joins. Proc. VLDB Endow. 5, 3 (November 2011), 253-264.
8. R. Paredes, N. Reyes, Solving similarity joins and range queries in metric spaces with the list of twin clusters, J. of Discrete Algorithms 7 (2009) 18-35.