International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

Duplicate Detection by Progressive Sorted Neighbor Method

1.NEDUNOORI.SANDHYA DEVI

1.PG Scholar, Department of CSE,Vaagdevi College of Engineering,Autonomous,

Bollikunta,Warangal Telangana,Mail.id:nedunoori.sandhya@gmail.com

2.K.GOUTHAM RAJU

2.Associate Professor Department of CSE,Vaagdevi College ofEngineering,Autonomous

Bollikunta,Warangal Telangana

Mail.Id:gouthamraj.kodam@gmail.com

Abstract—Duplicate detection is the process of identifying multiple representations of same real world entities. Today, duplicate detection methods need to process ever larger datasets in ever shorter time: maintaining the quality of a dataset becomes increasingly difficult. We present two novel, progressive duplicate detection algorithms that significantly increase the efficiency of finding duplicates if the execution time is limited: They maximize the gain of the overall process within the time available by reporting most results much earlier than traditional approaches. Comprehensive experiments show that our progressive algorithms can double the efficiency over time of traditional duplicate

detection and significantly improve upon related work.

1 INTRODUCTION

DATA are among the most important assets of a company. But due to data changes and sloppy data entry, errors such as duplicate entries might occur, making data cleansing and in particular duplicate detection indispensable.However, the pure size of today's datasets render duplicate detection processes expensive. Online retailers, for example, offer huge catalogs comprising a constantly growing set of items from many different suppliers. As independent persons change the product portfolio, duplicates arise. Although there is an obvious need for deduplication, online shops without downtime cannot afford traditional deduplication. Progressive duplicate detection identifies most duplicate pairs early in the detection process. Instead of reducing the overall time needed to finish the entire process, progressive approaches try to reduce the average time after which a duplicate is found. Early termination, in particular, then yields more complete results on a progressive algorithm than on any traditional approach. duplicates found by three different duplicate detection algorithms in relation to their processing time: The incremental algorithm reports new duplicates at an almost constant frequency. This output behavior is common for state-of-the-art duplicate detection algorithms. In this work, however, we focus on progressive algorithms, which try to report most matches early on, while possibly slightly increasing their overall runtime. To achieve this, they need to estimate the similarity of all comparison candidates in order to compare most promising record pairs first. With the pair selection techniques of the duplicate detection process, there exists a trade-off between the amount of time needed to run a duplicate detection algorithm and the completeness of the results. Progressive techniques make this trade-off more beneficial as they deliver more complete results in shorter amounts of time. Furthermore, they make it easier for the user to define this trade-off, because the detection time or result size can directly be specified instead of parameters whose influence on detection time and result size is hard to guess. We present several use cases where this becomes important:

1) A user has only limited, maybe unknown time for data cleansing and wants to make best possible use of it. Then, simply start the algorithm and terminate it when needed. The result size will be maximized.

2) A user has little knowledge about the given data but still needs to configure the

cleansing process. Then, let the progressive algorithm choose window/block sizes and keys automatically.

3) A user needs to do the cleaning interactively to, for instance, find good sorting keys by trial and error. Then, run the progressive algorithm repeatedly; each run quickly reports possibly large results.

4) A user has to achieve a certain recall. Then, use the result curves of progressive algorithms to estimate how many more duplicates can be found further; in general, the curves asymptotically converge against the real number of duplicates in the dataset. We propose two novel, progressive duplicate detection algorithms namely progressive sorted neighborhood method (PSNM), which performs best on small and almost clean datasets, and progressive blocking (PB), which performs best on large and very dirty datasets. Both enhance the efficiency of duplicate detection even on very large datasets. In comparison to traditional duplicate detection, progressive duplicate detection satisfies two conditions

II RELATED WORK

Much research on duplicate detection [2], [3], also known as entity resolution and by many other names, focuses on pairs election algorithms that try to maximize recall on the one hand and efficiency on the other hand. The most prominent algorithms in this area are Blocking [4] and the sorted neighborhood method (SNM) [5]. Adaptive techniques. Previous publications on duplicate detection often focus on reducing the overall runtime. Thereby, some of the proposed algorithms are already capable of estimating the quality of comparison candidates [6], [7], [8]. The algorithms use this information to choose the comparison candidates more carefully. For the same reason, other approaches utilize adaptive windowing techniques, which dynamically adjust the window size depending on the amount of recently found duplicates [9], [10]. These adaptive techniques dynamically improve the efficiency of duplicate detection, but in contrast to our progressive techniques, they need to run for certain periods of time and cannot maximize the efficiency for any given time slot. Progressive techniques. In the last few years, the economic need for progressive algorithms also initiated some concrete studies in this domain. For instance, pay-as-you-go algorithms for information integration on large scale datasets have been presented [11]. Other works introduced progressive data cleansing algorithms for the analysis of sensor data streams [12]. However, these approaches cannot be applied to duplicate detection. Xiao et al. proposed a top-k similarity join that uses a

special index structure to estimate promising comparison candidates [13]. This approach progressively resolves duplicates and also eases the parameterization problem. Although the result of this approach is similar to our approaches (a list of duplicates almost ordered by similarity), the focus differs: Xiao et al. find the top-k most similar duplicates regardless of how long this takes by weakening the similarity threshold; we find as many duplicates as possible in a given time. That these duplicates are also the most similar ones is a side effect of our approaches. Pay-As-You-Go Entity Resolution by Whang et al. introduced three kinds of progressive duplicate detection techniques, called "hints" [1]. A hint defines a probably good execution order for the comparisons in order to match promising record pairs earlier than less promising record pairs. However, all presented hints produce static orders for the comparisons and miss the opportunity to dynamically adjust the comparison order at runtime based on intermediate results. Some of our techniques directly address this issue. Furthermore, the presented duplicate detection approaches calculate a hint only for a specific partition, which is a (possibly large) subset of records that fits into main memory. By completing one partition of a large dataset after another, the overall duplicate detection process is no longer progressive. This issue is only partly addressed in [1], which proposes to calculate the hints using all partitions.

3 PROGRESSIVE SNM

The progressive sorted neighborhood method is based on the traditional sorted neighborhood method [5]: PSNM sorts the input data using a predefined sorting key and only compares records that are within a window of records in the sorted order. The intuition is that records that are close in the sorted order are more likely to be duplicates than records that are far apart, because they are already similar with respect to their sorting key. More specifically, the distance of two records in their sort ranks (rank-distance) gives PSNM an estimate of their matching likelihood. The PSNM algorithm uses this intuition to iteratively vary the window size, starting with a small window of size two that quickly finds the most promising records. This static approach has already been proposed as the sorted list of record pairs (SLRPs) hint [1]. The PSNM algorithm differs by dynamically changing the execution order of the comparisons based on intermediate results (Look-Ahead). Furthermore, PSNM integrates a progressive sorting phase (MagpieSort)

and can progressively process significantly larger datasets.

PSNM Algorithm

Algorithm 1 depicts our implementation of PSNM. The algorithmtakes five input parameters: D is a reference to the data, which has not been loaded from disk yet. The sorting key K defines the attribute or attribute combination that should be used in the sorting step. W specifies the maximum window size, which corresponds to the window size of the traditional sorted neighborhood method. When using early termination, this parameter can be set to an optimistically high default value. Parameter I defines the enlargement interval for the progressive iterations. Section 3.2 describes this parameter in more detail. For now, assume it has the default value 1. The last parameter N specifies the number of records in the dataset. This number can be gleaned in the sorting step, butwe list it as a parameter for presentation purposes.

Algorithm 1. Progressive Sorted Neighborhood

Require: dataset reference D, sorting key K, window size

W, enlargement interval size I, number of records N

1: procedure PSNM(D, K, W, I, N)

2: pSize calcPartitionSize(D)

3: pNum dN=ðpSize _W þ 1Þe

4: array order size N as Integer

5: array recs size pSize as Record

6: order sortProgressive(D, K, I, pSize, pNum)

7: for currentI 2 to dW=I e do

8: for currentP 1 to pNum do

9: recs loadPartition(D, currentP)

10: for dist 2 range(currentI, I, W) do

11: for i 0 to jrecsj _ dist do

12: pair hrecs½i_; recs½i þ dist_i

13: if compare(pair) then

14: emit(pair)

15: lookAhead(pair)

In many practical scenarios, the entire dataset will not fit in main memory. To address this, PSNM operates on a partition of the dataset at a time.

IV PROGRESSIVE BLOCKING

In contrast to windowing algorithms, blocking algorithms assign each record to a fixed group of similar records (the blocks) and then compare all pairs of records within these groups. Progressive blocking is a novel approach that builds upon an equidistant blocking technique and the successive enlargement of blocks. Like PSNM, it also presorts the records to use their rank-distance in this sorting for similarity estimation. Based on the sorting, PB first creates and then progressively extends a fine-grained blocking. These block extensions are specifically executed on neighborhoods around already identified duplicates, which enables PB to expose

clusters earlier than PSNM using the block comparison matrix. To create this matrix, a preprocessing step has already sorted the records that form the Blocks 1-8 (depicted as vertical and horizontal axes). Each block within the block comparison matrix represents the comparisons of all records in one block with all records in another block. For instance, the field in the 4th row and the 5th column represents the comparisons of all records in Block 4 with all records in Block 5. Assuming a symmetric similarity measure, we can ignore the bottom left part of the matrix. The exemplary number of found duplicates is depicted in the according fields. In this example, the block comparison

Algorithm 2. Progressive Blocking

Require: dataset reference D, key attribute K, maximum
block range R, block size S and record number N

1: procedure PB(D, K, R, S, N)
2: pSize calcPartitionSize(D)
3: bPerP bpSize=Sc
4: bNum dN=Se
5: pNum dbNum=bPerPe
6: array order size N as Integer
7: array blocks size bPerP as hInteger;Record½ _i
8: priority queue bPairs as hInteger; Integer; Integeri
9: bPairs fh1; 1; i; . . . ;hbNum; bNum; ig
10: order sortProgressive(D, K, S, bPerP, bPairs)
11: for i 0 to pNum _ 1 do
12: pBPs get(bPairs, i _ bPerP, (i þ 1) _ bPerP)
13: blocks loadBlocks(pBPs, S, order)
14: compare(blocks, pBPs, order)
15: while bPairs is not empty do
16: pBPs fg
17: bestBPs takeBest(bbPerP=4c, bPairs, R)
18: for bestBP 2 bestBPs do
19: if bestBP[1] _ bestBP[0] < R then
20: pBPs pBPs [extend(bestBP)
21: blocks loadBlocks(pBPs, S, order)
22: compare(blocks, pBPs, order)
23: bPairs bPairs [pBPs
24: procedure compare(blocks, pBPs, order)
25: for pBP 2 pBPs do
26: hdPairs;cNumi comp(pBP, blocks, order)
27: emit(dPairs)
28: pBP[2] jdPairsj / cNum

At first, PB calculates the number of records per partition pSize by using a pessimistic sampling function in Line 2. The algorithm also calculates the number of loadable blocks per partition bPerP, the total number of blocks bNum, and the total number of partitions pNum. In the Lines 6 to 8, PB then defines the three main data structures: the order-array, which stores the ordered list of record IDs, the blocks-array, which holds

the current partition of blocked records, and the bPairs-list, which stores all recently evaluated block pairs.

V CONCLUSION

This paper introduced the progressive sorted neighborhood method and progressive blocking. Both algorithms increase the efficiency of duplicate detection for situations with limited execution time; they dynamically change the ranking of comparison candidates based on intermediate results to execute promising comparisons first and less promising comparisons later. To determine the performance gain of our algorithms, we proposed a novel quality measure for progressiveness that integrates seamlessly with existing measures. Using this measure, experiments showed that our approaches outperform the traditional SNM by up to 100 percent and related work by up to 30 percent.

REFERENCES

[1] S. E. Whang, D. Marmaros, and H. Garcia-Molina, "Pay-as-you-go entity resolution," IEEE Trans. Knowl. Data Eng., vol. 25, no. 5, pp. 1111–1124, May 2012.

[2] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," IEEE Trans. Knowl. Data Eng., vol. 19, no. 1, pp. 1–16, Jan. 2007.

[3] F. Naumann and M. Herschel, An Introduction to Duplicate Detection. San Rafael, CA, USA: Morgan & Claypool, 2010.

[4] H. B. Newcombe and J. M. Kennedy, "Record linkage: Making maximum use of the discriminating power of identifying information," Commun. ACM, vol. 5, no. 11, pp. 563–566, 1962.

[5] M. A. Hern_andez and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," Data Mining Knowl. Discovery, vol. 2, no. 1, pp. 9–37, 1998.

[6] X. Dong, A. Halevy, and J. Madhavan, "Reference reconciliation in complex information spaces," in Proc. Int. Conf. Manage. Data, 2005, pp. 85–96.

[7] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller, "Framework for evaluating clustering algorithms in duplicate detection," Proc. Very Large Databases Endowment, vol. 2, pp. 1282–= 1293, 2009.

[8] O. Hassanzadeh and R. J. Miller, "Creating probabilistic databases from duplicated data," VLDB J., vol. 18, no. 5, pp. 1141–1166, 2009.

[9] U. Draisbach, F. Naumann, S. Szott, and O. Wonneberg, "Adaptive windows for duplicate detection," in Proc. IEEE 28th Int. Conf. Data Eng., 2012, pp. 1073–1083.

[10] S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles, "Adaptive sorted neighborhood methods for efficient record linkage," in Proc. 7th ACM/ IEEE Joint Int. Conf. Digit. Libraries, 2007, pp. 185–194.

[11] J. Madhavan, S. R. Jeffery, S. Cohen, X. Dong, D. Ko, C. Yu, and A. Halevy, "Web-scale data integration: You can only afford to pay as you go," in Proc. Conf. Innovative Data Syst.
Res., 2007.

[12] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy, "Pay-as-you-go user feedback for dataspace systems," in Proc. Int. Conf. Manage. Data, 2008, pp. 847–860.

[13] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in Proc. IEEE Int. Conf. Data Eng., 2009, pp. 916–927.

[14] P. Indyk, "A small approximately min-wise independent family of hash functions," in Proc. 10th Annu. ACM-SIAM Symp. Discrete Algorithms, 1999, pp. 454–456.

[15] U. Draisbach and F. Naumann, "A generalization of blocking and windowing algorithms for duplicate detection," in Proc. Int. Conf. Data Knowl. Eng., 2011, pp. 18–24.

[16] H. S. Warren, Jr., "A modification of Warshall's algorithm for the transitive closure of binary relations," Commun. ACM, vol. 18, no. 4, pp. 218–220, 1975.

[17] M. Wallace and S. Kollias, "Computationally efficient incremental transitive closure of sparse fuzzy binary relations," in Proc. IEEE
Int. Conf. Fuzzy Syst., 2004, pp. 1561–1565.

[18] F. J. Damerau, "A technique for computer detection and correction of spelling errors," Commun. ACM, vol. 7, no. 3, pp. 171–176, 1964.

[19] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," IEEE Trans. Knowl. Data Eng., vol. 24, no. 9, pp. 1537–1555, Sep. 2012.

[20] B. Kille, F. Hopfgartner, T. Brodt, and T. Heintz, "The Plista dataset," in Proc. Int. Workshop Challenge News Recommender Syst., 2013, pp. 16–23.