IJMTARC – VOLUME – IV – ISSUE - 16 - DEC 2016 ISSN: 2320-1363

Duplicate Detection of Data sets progressively

Ms. T.SUMANA SRI Mr.P.VIJAY

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. As a preview of Section 8.3, Fig. 1 depicts the number of 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 [1]:

Improved early quality. Let t be an arbitrary target time at which results are needed. Then the progressive algorithm discovers more duplicate pairs at t than the corresponding traditional algorithm. Typically, t is smaller than the overall runtime of the traditional algorithm. Same eventual quality. If both a traditional algorithm and its progressive version finish execution, without early termination at t, they produce the same results.

Given any fixed-size time slot in which data cleansing is possible, progressive algorithms try to maximize their efficiency for that amount of time. To this end, our algorithms PSNM and PB dynamically adjust their behavior by automatically choosing optimal parameters, e.g., window sizes, block sizes, and sorting keys, rendering their manual specification superfluous. In this way, we significantly ease the parameterization complexity for duplicate detection in general and contribute to the development of more user interactive applications: We can offer fast feedback and alleviate the often difficult parameterization of the algorithms. In summary, our contributions are the following:

 We propose two dynamic progressive duplicate detection algorithms, PSNM and PB, which expose different strengths and outperform current approaches. We introduce a concurrent progressive approach for the multi-pass method and adapt an incremental transitive closure algorithm that together form the first complete progressive duplicate detection workflow. We define a novel quality measure for progressive duplicate detection to objectively rank the performance of different approaches. We exhaustively evaluate on several real-world datasets testing our own and previous algorithms. The duplicate detection workflow comprises the three steps pair-selection, pair-wise comparison, and clustering. For a progressive workflow, only the first and last step need to be modified. Therefore, we do not investigate the comparison step and propose algorithms that are independent of the quality of the similarity function. Our approaches build upon the most commonly used methods, sorting and (traditional) blocking, and thus make the same assumptions: duplicates are expected to be sorted close to one another or grouped in same buckets, respectively. Paper organization. Section 2 examines related work. Sections 3 and 4 introduce the

PSNM and the PB algorithm, which progressively find duplicates based on windowing and blocking techniques, respectively. Section 5 contributes the Attribute Concurrency multi-pass strategy, which enables PSNM and PB to automatically choose good key attributes. We discuss the incremental transitive closure calculation in Section 6 and define a novel quality measure for progressiveness in Section 7. Section 8 comprehensively evaluates our algorithms, showing that they can double the efficiency of traditional duplicate detection algorithms. Section 9 concludes this paper and discusses future work.

2 RELATED WORK

Much research on duplicate detection also known as entity resolution and by many other names, focuses on pairselection 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 and the sorted neighborhood method (SNM) . 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 .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 . 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]. Other works introduced progressive data cleansing algorithms for the analysis of sensor data streams. 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]. 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. The algorithms presented in our paper use a global ranking for the comparisons and consider the limited amount of available main memory. The third issue of the algorithms introduced by Whang et al. relates to the proposed pre-partitioning strategy:

3 PROGRESSIVE SNM

The progressive sorted neighborhood method is based on the traditional sorted neighborhood method 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.

3.1 PSNM Algorithm

Algorithm 1 depicts our implementation of PSNM. The algorithm takesfive 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..

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 N=ðpSize□W þ1Þ de 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 W=Ide do 8: for currentP 1 to pNum do 9: recs loadPartition(D, currentP) 10: for dist2range(currentI, I, W) do 11: for i 0 to recsjj □dist do 12: pair recs½i□; recs½iþdist□hi 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. The PSNM algorithm calculates an appropriate partition size pSize, i.e., the maximum number of records that fit in memory, using the pessimistic sampling function calcPartitionSize(D) in Line 2: If the data is read from a database, the function can calculate the size of a record from the data types and match this to the available main memory. Otherwise, it takes a sample of records and estimates the size of a record with the largest values for each field. In Line 3, the algorithm calculates the number of necessary partitions pNum, while

considering a partition overlap of W □1 records to slide the window across their boundaries. Line 4 defines the order-array, which stores the order of records with regard to the given key K. By storing only record IDs in this array, we assume that it can be kept in memory. To hold the actual records of a current partition, PSNM declares the recs-array in Line 5. In Line 6, PSNM sorts the dataset D by key K. The sorting is done by applying our progressive sorting algorithm Magpie, which we explain in Section 3.2. Afterwards, PSNM linearly increases the window size from 2 to the maximum window size W in steps of I (Line 7). In this way, promising close neighbors are selected first and less promising far-away neighbors later on. For each of these progressive iterations, PSNM reads the entire dataset once. Since the load process is done partition-wise, PSNM sequentially iterates (Line 8) and loads (Line 9) all partitions. To process a loaded partition, PSNM first iterates overall record rank-distances dist that are within the current window interval currentI. ForI ¼ 1 this is only one distance, namely the record rank-distance of the current main-iteration. In Line 11, PSNM then iterates all records in the current partition to compare them to their dist-neighbor.

3.2 Progressiveness Techniques Window interval.

PSNM needs to load all records in each progressive iteration and loading partitions from disk is expensive. Therefore, we introduced the window enlargement interval I in Line 7 and 10. It defines how many dist-iterations PSNM should execute on each loaded partition. For instance, if we set I ¼ 3, the algorithm loads the first partition to sequentially execute the rank-distances 1 to 3, then it loads the second partition to execute the same interval and so on until all partitions have been loaded once. Afterwards, all partitions are loaded again to run dist 4 to 6 and so forth. This strategy reduces the number of load processes. However, the theoretical progressiveness decreases as well, because we execute comparisons with a lower probability of matching earlier. So I constitutes a trade-off parameter that balances progressiveness and overall runtime. Partition caching. As we cannot assume the input to be physically sorted, the algorithm needs to repeatedly reiterate the entire file searching for the records of the next partition, which contains

the currently most promising comparison candidates. So, all records need to be read when loading the next partition. To overcome this issue, we implemented Partition Caching within the loadPartition(D, currentP) function in Line 9: If a partition is read for the first time, the function collects the requested records from the input dataset and materializes them to a new, dedicated cache file on disk. When the partition is later requested again, the function loads it from this cache file, reducing the costs for PSNM's additional I/O operations (and for possible parsing efforts on the file-input). Look-ahead. After sorting the input dataset, we find areas of high and low duplicate density, particularly if duplicates occur in larger clusters, i.e., groups of records that are all pair-wise duplicates. The Look-Ahead strategy uses this observation to adjust the ranking of comparison candidates at runtime: If record pairði;jÞhas been identified as a duplicate, then the pairsðiþ1;jÞandði;j þ1Þhave a high chance of being duplicates of the same cluster. Therefore, PSNM immediately compares them instead of waiting for the next progressive iteration. If one of the look-ahead comparisons detects another duplicate, a further look-ahead is recursively executed. In this way, PSNM iterates larger neighborhoods around duplicates to progressively reveal entire clusters. To avoid redundant comparisons in different look-aheads or in a following progressive iteration, PSNM maintains all executed comparisons in a temporary data structure. This behavior is implemented by the lookAhead(pair) function in Line 15 of our PSNM implementation. Since the look-ahead works recursively, it may perform comparisons that are beyond the given maximum window size W. Hence, it can find duplicates that cannot be found by the traditional Sorted Neighborhood Method. For easier comparison, we limited the maximum look-ahead rank-distance to W in our evaluation. In summary, PSNM automatically preferslocally promising comparisons in the otherwise static execution order by adaptively comparing record pairs in the neighborhood of previously detected duplicates. MagpieSort.

4 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. Sections 8.3 and 8.4 directly compare the performance of

PB and PSNM showing that PB is indeed preferable for datasets containing many large duplicate clusters.

4.1 PB Intuition

how PB chooses comparison candidates 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 ð4;5Þ delivered nine duplicates. Because of the equidistant blocking, all blocks have the same size. This eases the progressive extension process that we describe in the following. Only the last block might be smaller, if the dataset is not divisible by the desired block size. In the initial run, PB defines the blocking and executes all comparisons within each block. For the first progressive iteration, the algorithm then selects those block pairs that delivered the most duplicates in the initial run. In the example, these are the block pairs ð2;2Þ and ð5;5Þ. Because these two block pairs represent the areas with the currently highest duplicate density, the PB algorithm chooses ð1;2Þ and ð2;3Þ to progressively extend the first block pair and ð4;5Þ andð5;6Þto extend the second block pair. Having compared the four new block pairs, PB starts the second iteration. In this iteration, ð4;5Þ and ð5;6Þ are the best block pairs and, hence, extended. The results of this iteration then influences the third iteration and so on. In

this way, PB dynamically processes those neighborhoods that are expected to contain most new duplicates. In case of ties, the algorithm prefers block pairs with a smaller rank-distance, because the distance in the sort rank still defines the expected similarity of the records. The extensions continue until all blocks have been compared or a distance threshold for all remaining block pairs has been reached.

4.2 PB Algorithm

Algorithm 2 lists our implementation of PB. The algorithm accepts five input parameters: The dataset reference D specifies the dataset to be cleaned and the key attribute or key attribute combination K defines the sorting. The parameter R limits the maximum block range, which is the maximum rank-distance of two blocks in a block pair, and S specifies the size of the blocks. We discuss appropriate values for R and S in the next section. Finally, N is the size of the input dataset.

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 pSize=Sbc 4: bNum N=Sde 5: pNum bNum=bPerPde 6: array order size N as Integer 7: array blocks size bPerP as Integer;Record½□ hi 8: priority queue bPairs as Integer;Integer;Integer hi 9: bPairs 1;1;hi ;... ; bNum;bNum; hi fg 10: order sortProgressive(D, K, S, bPerP, bPairs) 11: for i 0 to pNum□1 do 12: pBPs get(bPairs, i□bPerP, (iþ1)□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(bPerP=4 bc , bPairs, R) 18: for bestBP2bestBPs 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 pBP2pBPs do 26: dPairs;cNum hi comp(pBP, blocks, order) 27: emit(dPairs) 28: pBP[2] dPairsjj / 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. Thereby, a block pair is represented as a triple of blockNr1;blockNr2;duplicatesPerCompariso n hi . We implemented the bPairs-list as a priority queue, because the algorithm frequently reads the top elements from this list. In the followingLine10,thePBalgorithmsortsthedat asetusingthe progressive MagpieSort algorithm. Afterwards, the Lines 11 to 14 load all blocks partition-wise from disk to execute the comparisonswithineachblock. After the preprocessing, the PB algorithm starts progressively extending the most promising block pairs (Lines 15 to 23). In each loop, PB first takes those block pairs bestBPs from the bPairs-list that reported the highest duplicate density. Thereby, at most bPerP=4 block pairs can be taken,because the algorithm needs to load two blocks per bestBP and each extension of a bestBP delivers two partition block pairs pBPs in Line 20. However, if such an extension exceeds the maximum block range R, the last bestBP is discarded. Having successfully defined the most promising block pairs, Line21loads thecorresponding blocksfromdisktocompare the pBPs in Line 22. The compare(blocks, pBPs, order)procedure is listed in Lines 24 to 28. For all partition block pairs pBP, the procedure compares each record of the first block to all records of the second block. The identified duplicate pairs dPairs are then emitted in Line 27. Furthermore, Line 28 assigns the duplicate pairs to the current pBP to later rank the duplicate density of this block pair with the density in other block pairs. Thereby, the amount of duplicates is normalized by the number of comparisons, because the last block is usually smaller than all other blocks. In Line 23, the algorithm adds the previously compared pBPs to the bPairslist to use them in the next progressive iteration. If the PB algorithm is not terminated prematurely, it automatically finishes when the list of bPairs is empty, e.g., no new block pairswithinthemaximumblockrangeRcanbef ound.

4.3 Blocking Techniques Block size.

A block pair consisting of two small blocks defines only few comparisons. Using such small blocks, the PB algorithm carefully

selects the most promising comparisons and avoids many less promising comparisons from a wider neighborhood. However, block pairs based on small blocks cannot characterize the duplicate density in their neighborhood well, because they represent a too small sample. A block pair consisting of large blocks, in contrast, may define too many, less promising comparisons, but produce better samples for the extension step. The block size parameter S, therefore, trades off the execution of non-promising comparisons and the extension quality. In preliminary experiments, we identified five records per block to be a generally good and not sensitive value. Maximum block range. The maximum block range parameter R is superfluous when using early termination. For our evaluation, however, we use this parameter to restrict the PB algorithm to approximately the same comparisons executed by the traditional sorted neighborhood method. We cannot restrict PB to execute exactly the same comparisons, because the selection of comparison candidates is more fine-grained by using a window than by using blocks. Nevertheless, the calculation of R as R ¼bwindowSize S ccauses PB to execute only minimally fewer comparisons. Extension strategy. The extend(bestBP) function in Line 20 of Algorithm 2 returns some block pairs in the neighborhood of the given bestBP. In our implementation, the function extends a block pair ði;jÞ to the block pairs ðiþ1;jÞ andði;j þ1Þas shown in Fig. 2. More eager extension strategies that select more block pairs from the neighborhood increase the progressiveness, if many large duplicate clusters are expected. By using a block size S close to the average duplicate cluster size, more eager extension strategies have, however, not shown a significant impact on PB's performance in our experiments. The benefit of detecting some cluster duplicates earlier was usually as high as the drawback of executing fruitless comparisons.

5 ATTRIBUTE CONCURRENCY

The best sorting or blocking key for a duplicate detection algorithm is generally unknown or hard to find. Most duplicate detection frameworks tackle this key selection problem by applying the multi-pass execution method. This method executes the duplicate detection algorithm multiple times using different keys in each pass. However, the execution order among the different keys is arbitrary. Therefore,

favoring good keys over poorer keys already increases the progressiveness of the multi-pass method. In this section, we present two multi-pass algorithms that dynamically interleave the different passes based on intermediate results to execute promising iterations earlier. The first algorithm is the attribute concurrent PSNM (AC-PSNM), which is the progressive implementation of the multi-pass method for the PSNM algorithm, and the second algorithm is the attribute concurrent PB (AC-PB), which is the corresponding implementation for the PB algorithm.

5.1 Attribute Concurrent PSNM

The basic idea of AC-PSNM is to weight and re-weight all given keys at runtime and to dynamically switch between the keys based on intermediate results. Thereto, the algorithm precalculates the sorting for each key attribute. The precalculation also executes the first progressive iteration for every key to count the number of results. Afterwards, the algorithm ranks the different keys by their result counts. The best key is then selected to process its next iteration. The number of results of this iteration can change the ranking of the current key so that another key might be chosen to execute its next iteration. In this way, the algorithm prefers the most promising key in each iteration. Algorithm 3 depicts our implementation of AC-PSNM. It takes the same five parameters as the basic PSNM algorithm but a set of keys Ks instead of a single key. First, AC-PSNM calculates the partition size pSize and the overall number of partitions pNum. During execution, each key is assigned an own state. To encode these states, the algorithm defines three basic data structures in Lines 4 to 6: an orders-array, which stores the different orders, a windowsarray, which stores the current window range for each key, and a dCounts-array, which stores the keys' current duplicate counts. To initialize these data structures, Line 7 iterates all given keys. For each key, the algorithm uses MagpieSort in Line 8 to create the corresponding order. Simultaneously, it calculates and counts the duplicates of the key's first progressive iteration. In Line 9, AC-PSNM then stores the number 2 as the recently used window range for the current key.

Algorithm 3. Attribute Concurrent PSNM
Require: dataset reference D, sorting keys Ks, window size W, enlargement interval

size I and record number N 1: procedure AC-PSNM(D, Ks, W, I, N) 2: pSize calcPartitionSize(D) 3: pNum N=ðpSize□W þ1Þ de 4: array orders dimension Ksj j□ N as Integer 5: array windows size Ksjj as Integer 6: array dCounts size Ksjj as Integer 7: for k 0 to Ksj j□ 1 do 8: orders½k□; dCounts½k□ hi sortProgressive(D, I, Ks½k□, pSize, pNum) 9: windows½k□ 2 10: while9w2windows : w < W do 11: k findBestKey(dCounts, windows) 12: windows½k□ windows½k□þ1 13: dPairs process(D, I, N, orders½k□, windows½k□, pSize, pNum) 14: dCounts½k□ dPairsjj

After initialization, AC-PSNM enters the main loop in Line 10. This loop continues until the maximum window size W has been reached with all keys. In the loop's body, the algorithm first selects the key k that delivered the most duplicates in the last iteration by consulting the dCountsarray in Line 11. To execute the next progressive iteration for k, the algorithm first increases the corresponding window range by one. Then, it calls the process(...) function that runs the PSNM algorithm with only the specified rankdistance. Afterwards, Line 14 updates the duplicate count of the current key with the amount of newly found duplicates. Due to the update, AC-PSNM might select another best key in the next iteration. In this way, the algorithm dynamically re-ranks the sorting keys. Note that the process(...) function in Line 13 handles record comparisons slightly different than MagpieSort in Line 8. Since the initialization uses the keys in arbitrary order, MagpieSort counts all duplicates that are found in the first iterations to treat all keys equally. Afterwards, the process(...) function reports only new duplicates that have not been found before with a different key. This change in behavior guarantees that the progressive main loop always chooses the currently most promising key. Counting only new duplicates also causes the algorithm to automatically rank those keys last, whose orders are subsumed by other keys' orders. For instance, "postcode" might displace "city" as a key in an address dataset, because it usually generates a similar but more fine-grained order.

5.2 Attribute Concurrent

PB Instead of scheduling progressive iterations of different keys, AC-PB directly schedules the bPair-comparisons of all keys: AC-PB first calculates the initial block pairs and their duplicate counts for all keys (see

Fig. 2 in Section 4.1); then, it takes all block pairs together and ranks them regardless of the key, with which the individual blocks have initially been created. This approach lets AC-PB rank the comparisons even more precisely than AC-PSNM.

6 CONCLUSION AND FUTURE WORK

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. For the construction of a fully progressive duplicate detection workflow, we proposed a progressive sorting method, Magpie, a progressive multi-pass execution model, Attribute Concurrency, and an incremental transitive closure algorithm. The adaptations AC-PSNM and AC-PB use multiple sort keys concurrently to interleave their progressive iterations. By analyzing intermediate results, both approaches dynamically rank the different sort keys at runtime, drastically easing the key selection problem. In future work, we want to combine our progressive approaches with scalable approaches for duplicate detection to deliver results even faster. In particular, Kolb et al. introduced a two phase parallel SNM , which executes a traditional SNM on balanced, overlapping partitions. Here, we can instead use our PSNM to progressively find duplicates in parallel.

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.

Author's Details

.P.VIJAYA RAGHAVULU received M.Tech(CSE) Degree from School of Information Technology, Autonomous, and Affiliated to JNTUA, Anathapur. He is currently working as Assistant Professor in the Department of Computer Science and Engineering in Modugula Kalavathamma Institute of Technology for Women, Rajampet, Kadapa,AP . His interests includes Object Oriented Programming, Operating System, Database Management System, Computer Networking, Cloud Computing and Software Quality Assurance.

Ms. T.SUMANA SRI She is currently pursuing M.tech Degree in Computer Science and Engineering specialization in Modugula Kalavathamma Institute of Technology for Women, Rajampet, Kadapa,AP **.**