Progressive Identification of Duplicity

Mohd Shoaib Amir Khan*

* M. Tech Computer Science and Engineering, S.R.M University, Kattankulathur, INDIA
Guided by: Mrs. K. Deeba

Abstract- Databases contains very large datasets, where various duplicate records are present. The duplicate records occur when data entries are stored in a uniform manner in the database, resolving the structural heterogeneity problem. Detection of duplicate records are difficult to find and it take more execution time. In this literature survey papers various techniques used to find duplicate records in database but there are some issues in this techniques. To address this Progressive algorithms has been proposed for that significantly increases the efficiency of finding duplicates if the execution time is limited and improve the quality of records.

Index Terms- Duplicate record detection, Progressive Sorted Neighborhood Method, Progressive Blocking, Entity resolution.

I. Introduction

Data's are among the most important assets of a company. But due to data changes and copied data entry, errors such as duplicate entries might occur, making data cleansing and particular duplicate detection indispensable. However, the size of today's datasets makes 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.

Databases play an important role in today's IT based economy. Many industries and systems depend on the accuracy of databases to carry out operations. Therefore, the quality of the information stored in the databases, can have significant cost implications to a system that relies on information to function and conduct business. With the ever increasing volume of data, data quality problems abound. Multiple, yet different of the same real-world objects in data, duplicates, are one of the most intriguing data quality problems.

Progressive identification of duplicity 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.
Several use cases are:

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.

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.

Two novel algorithms are proposed 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.

II. RELATED WORK

Much research on duplicate detection [2], [3], also known as entity resolution and by many other names, focuses on pair-selection 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 run for certain periods of time and can- not 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].

2.1 Disadvantages

1. 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.
2. Needs to process large dataset in short time
3. Quality of data set becomes increasingly difficult.

III. PROPOSED SYSTEM

In an error-free system with perfectly clean data, the construction of a comprehensive view of the data consists of linking in relational terms, joining two or more tables on their key fields. Unfortunately, data often lack a unique, global identifier that would permit such an operation. Furthermore, the data are neither carefully controlled for quality nor defined in a consistent way across different data sources.

1. Two algorithms are proposed, namely progressive sorted neighborhood method (PSNM), which performs best on small and almost clean datasets.
2. Progressive blocking (PB), which performs best on large and very dirty datasets. Both enhance the efficiency of duplicate detection even on very large datasets.
3. 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 PSNM algorithm differs by dynamically changing the execution order of the comparisons based on intermediate results.
4. 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.

3.1 Advantages

1. To detect the duplicate data over short time in the real world.
2. High Accuracy
3. Time taken is minimized for detecting duplicate data (overall time).
4. Percentage (%) of duplicate data occurs in the uploaded content.
5. Total number of duplicate words.

3.2 ROGRESSIVE SORTED NEIGHBORHOOD METHOD(PSNM)

The progressive sorted neighborhood 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 progressive iterations.

3.2.1 PSNM Algorithm

Algorithm 1 depicts implementation of PSNM. The algorithm takes 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.

```
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)
```

The PSNM algorithm calculates an appropriate partition size pSize, i.e., the maximum number of records that fit in memory, using the pessimistic joining 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. After- wards, 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. For I ¼ 1 this is only one distance, namely the record rank-distance of the cur- rent main-iteration. In Line

11, PSNM then iterates all records in the current partition to compare them to their neighbor. The comparison is executed using the compare(pair) function in Line 13. If this function returns "true", a duplicate has been found and can be emitted. Furthermore, PSNM evokes the lookAhead(pair) method, which we explain later, to progressively search for more duplicates in the current neighborhood. If not terminated early by the user, PSNM finishes when all intervals have been processed and the maximum window size W has been reached.

3.3 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 pre- sorts 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 key attribute , combination K defines the sorting. The parameter R limits the maximum block range, which is the maxi- mum 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.

Fig. 1. PB in a block comparison matrix

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
 4:   bNum
 5:   pNum bpSize=S c dN =S e 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
```

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 and ð5; 6Þ to 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 dis- tance 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.

IV. PROPERTIES 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. How- ever, 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.

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 pre-calculates the sorting for each key attribute. The pre-calculation 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. 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 dN =ðpSize W þ 1Þe
4:  array orders dimension jKsj    N as Integer
5:  array windows size jKsj as Integer
6:  array dCounts size jKsj as Integer
7:  for k     0 to jKsj    1 do
8:   horders½k  ; dCounts½k  i       sortProgressive(D,  I,Ks½k  , pSize, pNum)
9:  windows½k       2
10:  while 9 w 2 windows : w < W do
11:  k  findBestKey(dCounts, windows)
12:  windows½k       windows½k þ 1
13:  dPairs       process(D, I, N, orders½k ,windows½k , pSize, pNum)
14: dCounts½k jdPairsj
```

V. CLOSURE

Due to careful pair-selection and the use of similarity thresh- olds, the result of a duplicate detection run is usually not transitively closed: the record pairs ða; bÞ and ðb; cÞ might be recognized as duplicates but ða; cÞ is (yet) missing in the result. Traditional duplicate detection algorithms, therefore, calculate the transitive closure of all results in the end [16]. As this calculation is blocking in nature, it hinders progressiveness. Therefore, we propose to calculate the transitive closure incrementally while the detection algorithm is running.

A suitable incremental transitive closure algorithm has already been introduced by Wallace and Kollias [17]. The proposed algorithm incrementally adds new duplicates, which are given as pairs of record identifiers, to an internal data structure that serves to calculate transitive relations from current results. The proposed data structure comprises two sorted lists of duplicates—one sorted by first records and one sorted by second records. If n is the number of records in the result, the proposed data structure exhibits an insert complexity of Oðn þ logðnÞÞ and a read complexity of OðlogðnÞÞ. As these complexities would introduce a significant performance drawback to our progressive work- flow, we instead store the duplicates in an index structure: We directly map each record identifier to a set of record identifiers representing a duplicate cluster. To add a new duplicate, we lookup the two contained records and point them to the same cluster, in which we add both records. Because of the map's overhead, this data structure requires approximately 75 percent more memory. However, inserts and reads can be easily done in constant time.

VI. LITERATURE SURVEY

Duplicate Record Detection: A Survey

Often, in the real world, entities have two or more representations in databases. Duplicate records do not share a common key and/or they contain errors that make duplicate matching a difficult task. Errors are introduced as the result of transcription errors, incomplete information, lack of standard formats, or any combination of these factors. In this paper, a thorough analysis of the literature on duplicate record detection. It cover similarity metrics that are commonly used to detect similar field entries, and we present an extensive set of duplicate detection algorithms that can detect approximately duplicate records in a database. It also cover multiple techniques for improving the efficiency and scalability of approximate duplicate detection algorithms. It conclude with coverage of existing tools and with a brief discussion of the big open problems in the area. Index Terms—Duplicate detection, data cleaning, data integration, record linkage, data reduplications, instance identification, database hardening, name matching, identity,uncertainty, entity resolution, fuzzy duplicate detection, entity matching.

Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem

The problem of merging multiple databases of information about common entities is frequently encountered in KDD and decision support applications in large commercial and government organizations. The problem study is often called the Merge/Purge problem and is difficult to solve both in scale and accuracy. Large repositories of data typically have numerous duplicate information entries about the same entities that are difficult to cull together without an intelligent ―equation theory that identifies equivalent items by a complex, domain-dependent matching process. It has developed a system for accomplishing this Data Cleansing task and demonstrate its use for cleansing lists of names of potential customers in a direct marketing-type application. Results for statistically generated data are shown to be accurate and effective when processing the data multiple times using different keys for sorting on each successive pass. Combing results of individual passes using transitive closure over the independent results, produces far more accurate results at lower cost.

Supervised learning approach for distance based record linkage as disclosure risk evaluation

In data privacy, record linkage is a well-known technique to evaluate the disclosure risk of protected data. It is used to evaluate the number of linked records between a data set and its protected version. In this paper , an overview of the work that has been done during the last months. It describes the development of a supervised learning method for distance-based record linkage, which determines the optimum parameters for the linkage process. It also present an evaluation and a comparison between three different alternatives of such method.

Framework for Evaluating Clustering Algorithms in Duplicate Detection

The presence of duplicate records is a major data quality concern in large databases. To detect duplicates, entity resolution also known as duplication detection or record linkage is used as a part of the data cleaning process to identify records that potentially refer to the same real-world entity. It present the Stringer system that provides an evaluation framework for understanding what barriers remain towards the goal of truly scalable and general purpose duplication detection algorithms.

This paper uses Stringer to evaluate the quality of the clusters (groups of potential duplicates) obtained from several unconstrained clustering algorithms used in concert with approximate join techniques. The work is motivated by the recent significant advancements that have made approximate join algorithms highly scalable. Its extensive evaluation reveals that some clustering algorithms that have never been considered for duplicate detection, perform extremely well in terms of both accuracy and scalability.

Pay-As-You-Go Entity Resolution

Entity resolution (ER) is the problem of identifying which records in a database refer to the same entity. In practice, many applications need to resolve large data sets efficiently, but do not require the ER result to be exact. For example, people data from the Web may simply be too large to completely resolve with a reasonable amount of work. As another example, real-time applications may not be able to tolerate any ER processing that takes longer than a certain amount of time. This paper investigates how we can maximize the progress of ER with a limited amount of work using ―hints, which give information on records that are likely to refer to the same real-world entity. A hint can be represented in various formats (e.g., a grouping of records based on their likelihood of matching), and ER can use this information as a guideline for which records to compare first. It introduce a family of techniques for constructing hints efficiently and techniques for using the hints to maximize the number of matching records identified using a limited amount of work. Using real data sets, we illustrate the potential gains of our pay-as-you-go approach compared to running ER without using hints.

VII. CONCLUSION AND FUTURE WORK

The progressive sorted neighborhood method and progressive blocking 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 later. This paper surveys different research papers that proposed various algorithms for detection of duplicate records. The suggested functions properly combine the best evidence available in order to identify whether two or more distinct record entries are replicas (i.e., represent the same real-world entity) or not. This is extremely useful for the no specialized user, who does not have to worry about setting up the best set of evidence for the replica identification task.

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.

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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

AUTHORS

First Author – Mohd Shoaib Amir Khan, M. Tech Computer Science and Engineering, S.R.M University, Kattankulathur, INDIA, Email: 1.shoaibamirkhan@gmail.com +919087860635