www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 6 Issue 2 Feb. 2017, Page No. 20372-20375
Index Copernicus Value (2015): 58.10, DOI: 10.18535/ijecs/v6i2.36

Efficiency Improvement in Data Detection

Qadri Syeda Asra Arshia

PG Scholar, Dept. of CNE, Khurana Sawant Institute of Engineering and Technology, Hingoli, Maharashtra, India.

Abstract— ***There are different methods to process duplicate detection in databases, but to process the data quickly at the same time maintaining the quality of database become difficult. In this paper PB and PSNM algorithms are presented to improve the efficiency of duplicate data detection in databases keeping the time to a shorter level.***

Index Terms — Data Cleaning, Blocking, Data Integrity Process, Duplicate Records.

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 renders 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 de duplication, online shops without downtime cannot afford traditional de duplication. 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 completes results on a progressive algorithm than on any traditional approach.
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. 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. Thus, data quality is often compromised by many factors, including data entry errors (e.g.,studet instead of student), missing integrity constraints (e.g., allowing entries such as Employee Age=567), and multiple conventions for recording information.

2. RELATED WORK

Steven Euijong Whang, David Marmaros, [1] 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 recor ds that are likely to refer to the same real-world entity. A hint can be represented in various formats, and ER can use this information as a guideline for which records to compare first. We 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. An ER process is often extremely expensive due to very large data sets and compute-intensive record comparisons.
The proposed a pay-as-you-go approach for Entity Resolution (ER) where given a limit in resources (e.g., work, runtime) we attempt to make the maximum progress possible. In [8] the World Wide Web is witnessing an increase in the amount of structured content – vast heterogeneous collections of structured data are on the rise due to the Deep Web, annotation schemes like Flickr, and sites like Google Base. While this phenomenon is creating an opportunity for structured data management, dealing with heterogeneity on the web-scale presents many new challenges. In this paper, we highlight these challenges in two scenarios – the Deep Web and Google Base.

We contend that traditional data integration techniques are no longer valid in the face of such heterogeneity and scale. We propose new data integration architecture, PAYGO, which is inspired by the concept of data spaces and emphasizes pay-as-you-go data management as means for achieving web-scale data integration.
[10] Similarity join is a useful primitive operation underlying many applications, such as near duplicate Web page detection, data integration, and pattern recognition. Traditional similarity joins require a user to specify a

DOI: 10.18535/ijecs/v6i2.36

similarity threshold. In this paper, we study a variant of the similarity join, termed top-k set similarity join. It returns the top-k pairs of records ranked by their similarities, thus eliminating the guess work users have to perform when the similarity threshold is unknown. An algorithm, top k-join, is proposed to answer top-k similarity join efficiently. It is based on the prefix filtering principle and employs tight upper bounding of similarity values of unseen pairs. Experimental results demonstrate the efficiency of the proposed algorithm on large-scale real datasets. Given a similarity function, a similarity join between two sets of records returns pairs of records from two sets such that their similarities are no less than a given threshold. In this paper, we study the problem of answering similarity join queries to retrieve top-k pairs of records ranked by their similarities. Existing approaches for the traditional similarity joins with a given threshold will have to make guesses on the similarity threshold and incur much redundant calculation. We propose an efficient algorithm that computes the answers in a progressive manner.

[2] Ahmed K. Elmagarmid, Panagiotis G.Ipeirotis, Vassilios S.Verykios 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, we present a thorough analysis of the literature on duplicate record detection. We 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. We also cover multiple techniques for improving the efficiency and scalability of approximate duplicate detection algorithms. We conclude with coverage of existing tools and with a brief discussion of the big open problems in the area. The problem that we study has been known for more than five decades as the record linkage or the record matching problem in the statistics community. The goal of record matching is to identify records in the same or different databases that refer to the same real-world entity, even if the records are not identical.

In paper [9] Duplicate detection is the task of identifying all groups of records within a data set that represent the same real-world entity, respectively.

This task is difficult, because (i) representations might differ slightly, so some similarity measure must be defined to compare pairs of records and (ii) data sets might have a high volume making a pair-wise comparison of all records infeasible. To tackle the second problem, many algorithms have been suggested that partition the data set and compare all record pairs only within each partition. One well-known such approach is the Sorted Neighborhood Method (SNM), which sorts the data according to some key and then advances a window over the data comparing only records that appear within the same window.

3. PSNM ALGORITHM

Algorithm 1 depicts our 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. 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. For now, assume it has the default value 1. The last parameter N specifies the number of records in the data-set. This number can be gleaned in the sorting step, but we list it as a parameter for presentation purposes.

3.1 Algorithm:

Progressive Sorted Neighborhood Require: dataset reference D, sorting key K, window size W, enlargement interval size I, number of records N

```
Step 1: procedure PSNM(D, K, W, I, N)
Step 2: pSize← calcPartitionSize(D)
Step 3: pNum ← [N/pSize-W + 1)]
Step 4: array order size N as Integer
Step 5: array recs size pSize as Record
Step 6: order ←sortProgressive(D, K, I, pSize, pNum)
Step 7: for currentI←  2 todW=Iedo
Step 8: for currentP ← 1 to pNum do
Step 9: recs←  loadPartition(D, currentP)
Step 10: for dist belongs to range(currentI, I, W) do
Step 11: for i ← 0 to |recs|_ dist do
Step 12: pair← <recs[i], recs[i + dist]>
Step 13: if compare(pair) then
Step 14: emit(pair)
Step 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. 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 dist neighbor. The comparison is executed using the com-pare(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.

4. PROGRESSIVE BLOCKING ALGORITHM

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.
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 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.

4.1 PB Algorithm:

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

```
Step 1: procedure PB(D, K, R, S, N)
Step 2: pSize ← calcPartitionSize(D)
Step 3: bPerP ← [pSize/S]
Step 4: bNum ← [N/S]
Step 5: pNum ← [bNum/bPerP]
Step 6: array order size N as Integer
Step 7: array blocks size bPerP as <Integer; Record[]>
Step 8: priority queue bPairs as <Integer; Integer; Integer>
Step 9: bPairs ←{<1,1,->, . . . ,<bNum, bNum,->}
Step 10: order ←sortProgressive(D, K, S, bPerP, bPairs)
Step 11: for i ←0 to pNum - 1 do
Step 12: pBPs ← get(bPairs, i . bPerP, (i+1) . bPerP)
Step 13: blocks ← loadBlocks(pBPs, S, order)
Step 14: compare(blocks, pBPs, order)
Step 15: while bPairs is not empty do
Step 16: pBPs← {}
Step 17: bestBPs← takeBest([bPerP/4], bPairs, R)
Step 18: for bestBP belongs to bestBPs do
Step 19: if bestBP[1] _ bestBP[0] < R then
Step 20: pBPs← pBPs U extend(bestBP)
Step 21: blocks ←loadBlocks(pBPs, S, order)
Step 22: compare(blocks, pBPs, order)
Step 23: bPairs ←bPairs U pBPs
Step 24: procedure compare(blocks, pBPs, order)
Step 25: for pBP belongs to pBPs do
Step 26: <dPairs,cNum> comp(pBP, blocks, order)
Step 27: emit(dPairs)
Step 28: pBP[2] ←|dPairs|/ 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 hblockNr1; blockNr2; duplicatesPerComparisoni. We implemented the bPairs-list as a priority queue, because the algorithm frequently reads the top elements from this list.
In the following Line 10, the PB algorithm sorts the dataset using the progressive MagpieSort algorithm. Afterwards, the Lines 11 to 14 load all blocks partition-wise from disk to execute the comparisons within each block. 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, Line 21 loads the corresponding blocks from disk to compare 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 bPairs-list 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 pairs within the maximum block range R can be found.

5. EXPERIMENTAL RESULTS:

We consider restaurant information from an internal operational data warehouse an d introduce errors. Because we start from real data all characteristics of real data: variations in the lengths of strings, numbers of tokens in and frequencies of attribute values, co-occurrence patterns, etc. are preserved. Since, we know the duplicate tuples and their correct counterparts in the erroneous dataset; we can evaluate duplicate elimination algorithms. The execution bottleneck for duplicate detection is commonly the attribute correlation with likeness measures between the record sets which is quite expensive one. To dodge this restrictively expensive analysis of all sets of records, a basic method is to precisely segment the records into smaller subsets and quest for copies just inside of these allotments. Two contending methodologies are regularly referred to: Blocking techniques allotment records into disjoint subsets, for occurrence utilizing zip code as apportioning key. Sorted-neighborhood based strategies that sort the information as per some key, for example, last name, and afterward slide a window of altered size over the sorted information and look at sets just inside of the window.

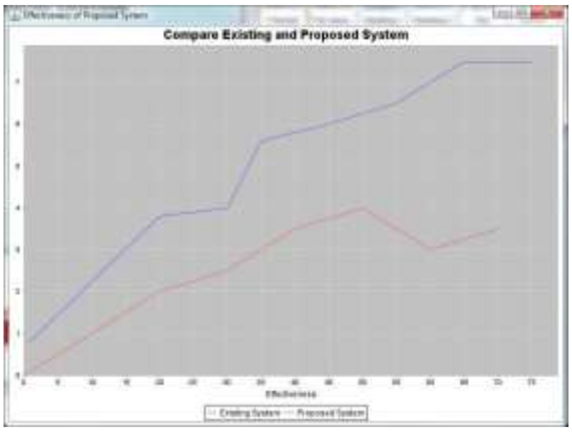

Fig.1. Based on the effectiveness metric for our proposed work with existing work.

6. 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. 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. Hernandez 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.

9. Shen H, Zhang Y, Improved approximate detection of duplicates for data streams over sliding windows, Journal of Computer Science and Technology, Volume 23(6), 2008, pp. 973-987.

10. Thorsten Papenbrock, Arvid Heise, and Felix Naumann, “Progressive Duplicate Detection”, IEEE Transactions on Knowledge and Data Engineering, Vol. 27, No. 5, May 2015.