An Efficient Data Cleaning Algorithm Based on Attributes Selection

Ling He, Zhongnan Zhang[1], Yize Tan, Minghong Liao
Software School, Xiamen University
422 South Siming Road
Xiamen, Fujian, China 361005

***Abstract* – In data cleaning, detecting approximately duplicate records in data warehouse is an important task. However, due to the wide range of possible data inconsistencies and the sheer data volume, determining whether two records are equal is not a simple arithmetic predicate. This paper proposes an improved algorithm of approximately duplicate records cleaning based on attributes selection after analyzing the existing basic sorted-neighborhood method (SNM) and multi-pass sorted neighborhood method (MPN), which are widely used in this field.**

Keywords –Approximately duplicate records; data cleaning; SNM

I. Introduction

Real-world data are always dirty. Overall in these data sets, there inevitably exists redundant data, missing data, uncertain data, inconsistent data, and so on. These data are referred to as "dirty data". Different data sources often contain redundant data in different representations. Given the “garbage in, garbage out” principle, dirty data will not be able to provide data miners with correct information. It is difficult for managers to make well-informed decisions based on information derived from such data [1]. Hence data cleaning is needed, and the aim of data cleaning is to detect errors and inconsistencies that exist in the data sets, remove or correct them, and improve the data quality [2].

Duplicate records cleaning is one of the most critical issues during the data cleaning process. Duplicate records do not share a common key; they contain errors that make duplicate matching difficult. The emphasis of detection and elimination of duplicate records is the match/merge problem [3]. However, due to data representation are different in different data sources or errors because of various reasons, determining whether two records are equal is not a simple arithmetic predicate, which makes the duplicate records cleaning very complicated [4].

Based on the existing duplicate records identification algorithm SNM and MPN, this paper proposes an improved algorithm which analyzes attributes and sort the dataset multiple times to make duplicate records more clustered. Our algorithm also gives a special weight to each attribute and introduces the concept of effective weight so that to make the comparison more accurate. Moreover, the filtering mechanism is introduced to improve the efficiency of detection. In part II, we analyses the existing algorithms. In part III, an improved algorithm is proposed. In part IV, the experimental results are presented. Finally in part V, we make a conclusion and some future work.

II. Related Works

Sort/merge method is the standard method to detect duplicate records in database. Its basic idea is to sort the data set and then compare the adjacent records to see whether they are similar or not. Hernandez et al. [4] proposed to make multiple sorts with different keywords, calculate the similarity of adjacent records separately and finally combine multiple computing results to complete the records matching process. Li et al. [5] raised the concept of variable sliding window and proposed to adjust the window’s size promptly according to the result of comparison between the similarity and the threshold to reduce match missing and improve the efficiency. Qiu et al. [6] proposed a clustering algorithm based on N-Gram and it could automatically correct the inserting of the word and remove errors during detection to improve the detection accuracy. Monge et al. [3, 7] tried to sort all records at first and then use the priority queues which contain a number of clusters to sequentially scan all the sorted records so that to do clustering dynamically.

The idea of the basic sorted-neighborhood method can be summarized in three steps as follows:

1. Create Sort Keys: compute a key for each record in the dataset by extracting relevant fields or portions of fields.
2. Sort Data: sort the entire data set based on the keys created in step one.
3. Merge: move a fixed size window through the sequential list of records and only compare each record with other records within the window. If the size of the window is *w*, every new record entering the window is compared with the previous *w*-1 records to find “matching” records.

The drawbacks of the SNM lie in the fact that it depends heavily on the sort keys [5]. The accuracy of detecting duplicate records largely depends on the created sort keywords, which have a direct impact on the matching efficiency and accuracy. If keywords are not selected properly, a lot of duplicate records may be missed. For example, since two duplicate records may be far away in the physical location after sorting, they may never locate in the same sliding

[1] Correspondence to: zhongnan_zhang@xmu.edu.cn. This work is partially supported by the Fundamental Research Funds for the Central Universities (No. 2010121071) and the Natural Science Foundation of Fujian Province of China (No. 11101060).

window at the same time and cannot be recognized as duplicate records. Secondly, it is hard to decide the sliding window size *w*. If *w* is too large, the comparison time will increase and some of the comparisons are not necessary; if *w* is too small, some duplicate records cannot be detected. When the sizes of clusters of all duplicate records in the data set vary greatly, no matter how to choose the size of *w* could not be appropriate. Thirdly, for the whole matching process, the algorithm's time complexity is O(*NlogN*), where *N* is the total number of records in the data set [8].

Single-pass algorithms go through the creating key, sort, and merging process once. Multi-pass algorithms go through the process several times, however each time they select a different key to sort. Hernandez et al. [4, 11] proposed the MPN algorithm. Its basic idea is to carry out SNM algorithm several times independently. Each time it creates a different sort keyword and uses a relatively small sliding window. In the merging step, the algorithm assumes that the similarity of the records is transitive and calculates the transitive closure [8]. Transitive closure means that if record *R1* is a duplicate of record *R2* and record *R2* is a duplicate of record *R3*, record *R1* should be a duplicate of record *R3*. By calculating the transitive closure with the duplicate records identified by each scan, the algorithm can obtain a more complete set of duplicate records and solve the problem of pretermission partially.

III. Improved Algorithm

A. Algorithm Description

This paper proposes an improved algorithm. Firstly, considering that the window size *w* is hard to select in SNM algorithm, we analyze attributes and sort the dataset multiple times to make duplicate records more clustered so that they may fall into the same sliding window simultaneously. Thus it just needs to slide a small fix-sized window through the sorted records and this can solve the problem of pretermission, not only improving the accuracy of the matching but also the efficiency of the matching. Secondly, when matching the fields, the algorithm gives a special weight to each attribute and introduces the concept of effective weight, makes the weight multiple the similarity of the corresponding non-empty attributes and combines them to obtain the overall record's similarity, and uses this value to judge whether two records are duplicate or not. Finally, it introduces the filtering mechanism to improve the efficiency of detection. During the process of attributes selection, we justify the selection by checking the similarity within *m* specially selected windows. This will help us use as less as possible times of selection to get the appropriate attribute for sorting.

The algorithm is divided into three steps and the process of the algorithm is shown in Fig. 1.

Line 1 and line 2 is step 1, attribute analysis; from line 3 to line 5 is step 2, data sort; line 6 is step 3, duplicate records detection. Each step will be described in detail later.

The flow chart which describes the algorithm intuitively is shown in Fig. 2.

```
Input: data set, k, m, w, LowThreshold
Output: duplicate records processed data sets

1. data preprocessing
2. select attributes to constitute sort sets and mode sets
3. do
4.     sort the data with attributes in sort sets
5. until (time of sort >= k | | SortGE() == true)
6. detect duplicate records
```

Fig. 1. Process of the algorithm

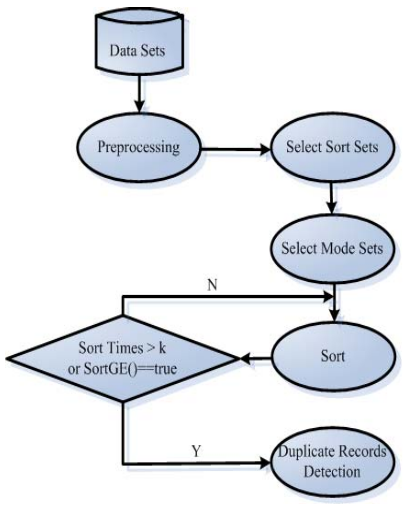

Fig. 2. Flow chart of the algorithm

1) Attribute Analysis

Firstly, the algorithm preprocesses the attributes. It treats space and punctuation as delimiters, splits the string of the attribute into words, and sorts the words lexicographically. For multi-language data, comparisons of characters between different languages not only don't make any sense but also waste a lot of time. Therefore, the algorithm uses natural segmentation and compares characters that belong to the same language.

Second, the algorithm selects attributes. As mentioned, only selecting one attribute as keyword will let the attribute have direct effect on the efficiency and accuracy of matching. If selecting all attributes, the execution time of the algorithm may be too long and some attributes may have a negative impact on the accuracy of matching.

Select Sort Sets: some important properties should be selected and under normal circumstances, the number of important attribute values should be as large as possible. According to this principle, we select *k* attributes which have the largest number of values to constitute the sort set R_k.

Select Mode Sets: on the contrary, select *m* attributes which have the highest number of replication to constitute the mode set R_m. This set is used below in the function *SortGE*.

2) Sort Data

In the second step, the algorithm uses the attributes in sort sets R_k in sequence to make multiple sorts to the record set R, and calls the function *SortGE* after each sort as a filtering mechanism to improve the efficiency of detection. The sort process continues until either the times of sort exceed k or the function *SortGE* returns true.

Function *SortGE*: the function *SortGE* is shown as follows in Fig. 3.

```
Input: sorted data set, m, w, LowThreshold
Output: true/false

1.  flag = true;
2.  take m window snapshot with the size of w
3.  for i = 1 to m do begin
4.      for j = 1 to w-1 do
5.          for k = j+1 to w do
6.              flag =RecordMatch(Rj, Rk, LowThreshold);
7.              if (flag == false)
8.                  return false;
9.  end;
10. return true;
```

Fig. 3. Description of function *SortGE*

In line 2, *SortGE* takes m window snapshots with size of w according to mode set R_m. The starting position of each window is where the most frequently appearing value of the attribute in mode set R_m appears in the data set in the first time. From line 4 to 8 the function compares the records pair wise in each window snapshot and calculates the similarity of the two records. If any two records' similarity is greater than *LowThreshold*, it means that the sort within the corresponding window is good enough. The function *RecordMatch* will be introduced in Fig. 5. If for m window snapshots, the sorts within them are all good enough, the function returns true and the sort for the whole record sets is good enough.

3) Duplicate Records Detection

In order to detect and eliminate duplicate records in the data set, the first problem needed to be solved is how to determine whether two records are duplicate or not. So it needs to compare the corresponding attributes of the records and calculate the similarity. If the similarity of two records exceeds a given threshold, the two records are considered duplicate. Otherwise, they are different. Evaluating similarity of the field, namely field matching problem, is the core [9].

Attribute matching algorithm: attribute matching is to determine whether two attribute values representing the same semantic entity can replace each other. It's the basis of the record matching. In 1981, Smith and Waterman proposed the famous Smith-Waterman algorithm [12]. The basic idea of the algorithm is to use the dynamic programming method and introduce the conception of gap, space and penalty to determine the matching value of two strings.

Some definitions used in Smith-waterman algorithm are as follows:

Score function: function $f(x, y)$ signifies the comparing score of x and y, reflecting the level of similarity of character x and y. In general, if x and y are the same, $f(x, y)$ is positive, otherwise the value is zero or negative.

Gap: in a single sequence any continuous interval as long as possible is called a gap; the number of vacancies ('-') in the gap is called the length of the space and is expressed as *spaces*. The number of vacancy in the sequence is expressed as *gaps*.

Penalty: The introduction of gap is to compensate for the insertion or deletion caused by the mutation. When a gap is inducted, the matching score will be deducted. The usual gap penalty function can be expressed as:

$$Penalty = W_g + W_s \times spaces \qquad (1)$$

where W_g represents the penalty of initializing a gap and W_s represents the penalty for adding one space to a gap [12].

For two attribute sequences $S(s_1,s_2,\ldots,s_p)$ and $T(t_1,t_2,\ldots,t_q)$, their corresponding lengths are p and q. According to the method of dynamic programming, we construct a size of $(p+1)\times(q+1)$ matrix D to store the possible matching score. The similarity comparison function *Sim* is shown in Fig. 4.

```
Input: two attribute sequences S, T
Output: the match score

1. Initialize matrix D[p+1][q+1] with 0
2. for i = 1 to p do begin
3.     for j = 1 to q
4.         if(Si == Tj)
5.             D[i][j] = D[i-1][j-1]+f(Si,Tj);
6.         for(x = 1 to i-1)
7.             row = max(D[i-x][j]-Wx);
8.         for(y = 1 to j-1)
9.             column = max(D[i][j-y]-Wy);
10.        D[i][j] = max(D[i][j], row, column);
11. end;
12. return D[p][q]
```

Fig. 4. Description of function *Sim*

In line 1, the function firstly initializes the matrix D with value zero. From line 2 to 11, the loop is a process of filling the matrix $D[p][q]$. From line 4 to 5, as mentioned above, function $f(x, y)$ signifies the comparing score of x and y, reflecting the level of similarity of character x and y. So if the two corresponding values in sequence S and T are the same, the value of matrix is calculated as line 5. The other two calculating formulas are shown in line 7 and 9. In line 7, the value Wx means the penalty of the gap with the length of x added to sequence S. In line 9, the value Wy means the penalty of the gap with the length of y added to sequence T. In line 10, the function calculates the value of $D[i][j]$, which is the maximum of the results calculated in line 5, 7 and 9 respectively. In the end, the function returns the value of $D[p][q]$, which is the matching score of the two attributes, namely the results of their similarity.

Record matching: For all records, the contributions of different attributes are unequal. In [10], each attribute C_i in the key is given a special weight W_i. Thus the similarity rate can be worked out according to the weights. The *null* values of attributes should also be taken into consideration. In order to eliminate the effect caused by the loss of attribute's value, the concept of effective weight is introduced. Assuming that there are *n* attributes participating in comparison and *R1* and *R2* represent two records, two records' similarity rate can be computed as follows:

$$Tsim(R1,R2)=\sum_{i=1}^{n}Valid[i]*W_i*Sim(R1_i,R2_i) \qquad (2)$$

in which $W_1+W_2+\ldots+W_n=1$. The function *Sim* has been introduced in Fig. 4. The record matching process is shown in Fig. 5 as follows:

```
Input: record R1, R2, LowThreshold
Output:  true/false

1.  for i = 1 to n do begin
2.     if ((R1.Field[i] == null && R2.Field[i] != null) ||
            (R1.Field[i] != null && R2.Field[i] == null))
3.          Valid[i] == 0;
4.     else
5.          Valid[i] == 1;
6.  end;
7.  remainWeight = 1.0;
8.  for i = 1 to n do begin
9.     validWeightMatch +=
            Valid[i]*weight[i]*Sim(R1.Field[i], R2.Field[i]);
10.    remainWeight -= weight[i];
11.    if (validWeightMatch >= LowThreshold)
12.         return true ;
13.    else if ((validWeightMatch + remainWeight) <
                 LowThreshold)
14.         return false;
15. end;
16. if (validWeightMatch >= LowThreshold)
17.     return true;
18. else
19.     return false;
```

Fig. 5. Description of function *RecordMatch*

Only when both records are not null or null at the same time for the attribute *i*, the comparison result for that attribute can be counted and *Valid*[*i*] is set to 1. Otherwise *Valid*[*i*] is 0. From line 1 to 6, the function fills the array *Valid* to ensure the null attributes will affect the result of the comparisons. From line 8 to 15, the function calculates the similarity value of records *R1* and *R2* as the formula shows. During the comparing, the function firstly compares the records with the attributes from the sort sets R_k. Selecting the attributes of the sort set to be compared first is because that their weights are higher and they are more representative in the similarity of records. In line 7, the function firstly initializes *remainWeight* with the value of 1.0, which represents the total weight of remaining attributes during the comparison. In line 10, during the comparing, it modifies *remainWeight* dynamically. In line 11 and 12, if the value *validWeightMatch* is large enough, the comparison terminates and returns true; otherwise, in line 13 and 14, it shows that the matching score is too low and there is no need to compare any more, the comparison also terminates and returns false.

B. *Algorithm Analysis*

The total execution time *T* of the algorithm consists of three parts: the time of attribute preprocessing, represented as *T1*; the time of multiple sorts and verification of the sort results, as *T2*; the time of duplicate records detection, as *T3*. In step 1 of the algorithm, it just needs to traverse the data set one time to create attribute sets R_k and R_m, so $T1=O(N)$, where *N* is the total number of records in the data set. In step 2, the algorithm makes multiple sorts, and calls the function *SortGE* after each sort. As the number of sorts is no more than *k*, *T2* is no more than $kNlogN+kmw^2$, in which *m* is the number of window snapshots, *w* is the size of the sliding window, and *m* and *w* are much less than *N*. Therefore, $T2=O(NlogN)$. In step 3, the time complexity of record matching is *wnN*, in which *n* is the number of attributes for each record and $T3=O(N)$. So the time complexity of the algorithm is $O(NlogN)$ if *n* is much less than *logN*.

IV. Experimental Results

For our improved algorithm, we calculate the recall rate and the false recognition rate. The recall rate refers to the percentage of the identified duplicate records among all the real duplicate records. The false recognition rate is the percentage of falsely identified duplicate records among the identified duplicate records.

All the experiments are performed on a 2.93GHz Intel Core i3 processor computer with 3GB memory, running on Windows XP professional SP3. Our algorithm is implemented in VC6.0. The three experimental data sets respectively contain 1000, 5000 and 10000 records. They are generated by the method provided by Pudjijono [13]. Each record has 16 attributes. We select 5 attributes to constitute the sort set and 5 attributes to constitute the mode set.

The first experiment makes comparisons of recall rate and false merge rate with fixed window size and similarity threshold. The sliding window size *w* is set to 6 and the similarity threshold *LowThreshold* is set to 0.65.

The results are shown in TABLE I.

TABLE I
Recall and False Merge rate of Different Data Sets

Record number	Recall Rate	False Merge Rate
1000	91.8%	0%
5000	98.3%	0%
10000	94.5%	0%

From TABLE I, we can see that this algorithm has high stability. When the number of records increases, the false recognition rate is still low while the recall rate is high enough. Meanwhile, recall rate and false merge rate do not have a large fluctuation for different date sets.

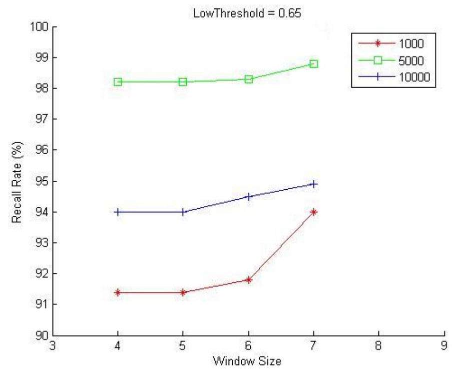

Fig. 6. Recall rate with different window size *w*

Fig.6 shows different recall rates with different window sizes and a fixed value of *LowThreshold* as 0.65. We can see that as the increase of the window size, the recall rate will also increase so that the algorithm can identify more duplicate records and improve the effectiveness of the algorithm.

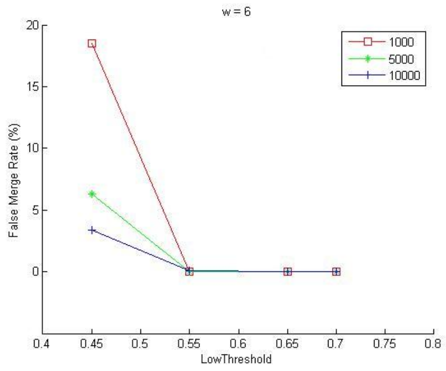

Fig. 7. False merge rate with different threshold

Fig.7 shows different false merge rates with different thresholds and a fixed value of window size *w* as 6. We can see that as the increase of the threshold, the false recognition number of duplicate records decreases fast and finally the false recognition rates become zero for all three data sets.

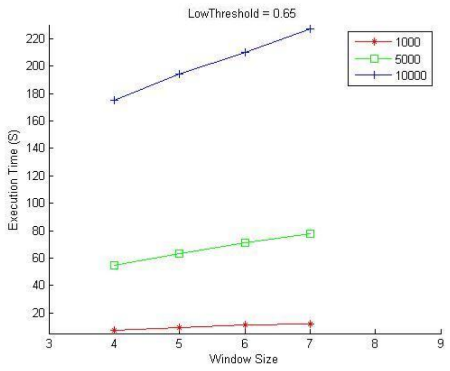

Fig. 8. Execution time with different window size *w*

Fig.8 shows the execution time with different window sizes and the *LowThreshold* is set as 0.65. It shows that with the increase of the window size, execution time's growth rate is comparatively gentle and the algorithm's performance is stable.

V. CONCLUSION

Detecting approximately duplicate records is a difficult task of data cleaning. This paper makes three aspects of improvements against SNM algorithm: it makes attribute analysis and multiple sorts to force duplicate records more clustered so that to improve the accuracy and efficiency of the matching; in the step of field matching, it gives each attribute a specific weight and introduces the concept of effective weight to improve the matching accuracy; the introduction of filtering mechanisms to record matching improves the accuracy of the duplicate records detection effectively.

By analyzing the algorithm, we find that transitive closure is likely to cause false identification. For example, record *R1* and record *R2* are similar and record *R2* and record *R3* are similar, but record *R1* and record *R3* may vary widely. Unfortunately the transitive closure considers that records *R1* and record *R3* are duplicate records. Further work still needs to be done to improve at least this deficiency.

REFERENCES

[1] Z. Guo and A. Zhou, "Research on Data Quality and Data Cleaning: a Survey," Journal of Software, vol. 13, 2002, pp. 2076-2082.

[2] E. Rahm and H.H. Do, "Data cleaning: Problems and current approaches," IEEE Data Engineering Bulletin, vol. 23, 2000, pp. 3-13.

[3] A.E. Monge, "Matching algorithm within a duplicate detection system," IEEE Data Engineering Bulletin, vol. 23, 2000, pp. 14-20.

[4] M.A. Hernandez and S.J. Stolfo, "Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem," Journal of Data Mining and Knowledge Discovery, vol. 2, 1998, pp. 9-37.

[5] J. Li and N. Zheng, "Improvement on the algorithm of data cleaning based on MPN," Computer Applications and Software, vol. 25, 2008, pp. 245-246.

[6] Y. Qiu, Z. Tian, W. Ji, and A. Zhou, "An efficient approach for detecting approximately duplicate database records," Chinese Journal of Computers, vol. 24, 2001, pp. 69-77.

[7] A.E. Monge and C.P. Elkan, "An efficient domain-independent algorithm for detecting approximately duplicate database records," In Proceeding of the ACM-SIGMOD Workshop on Data Mining and Knowledge Discovery. Tucson: ACM, 1997, pp. 23-29.

[8] A.E. Monge and C.P. Elkan, "An efficient domain-independent algorithm for detecting approximately duplicate database records," University of California, 1997.

[9] A.E. Monge and C.P. Elkan, "The field matching problem: algorithms and applications," Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining. Oregon: AAAI Press, 1996, pp. 267-270.

[10] K. Karadimitriou and J.M. Tyler, "The centroid method for compressing sets of similar images," Pattern Recognition Letters 19, 1998, pp. 585-593.

[11] M.A. Hernandez and S.J. Stolfo, "The Merge/Purge Problem for Large Databases," In Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Jose, California, 1995, pp. 127- 138.

[12] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," Journal of Molecular Biology, vol. 147, 1981, pp. 195-197.

[13] A. Pudjijono, "Probabilistic Data Generation," Master Thesis, Australian National University, 2008.