Random-Based Algorithm for Efficient Entity Matching

Pingfu Chao[1,2], Zhu Gao[2], Yuming Li[1,2], Junhua Fang[1,2], Rong Zhang[1,2,⋆], and Aoying Zhou[1,2]

[1] Institute for Data Science and Engineering
[2] Shanghai Key Laboratory of Trustworthy Computing
{51121500001,10132510331,51141500019,52131500020}@ecnu.cn,
{rzhang,ayzhou}@sei.ecnu.edu.cn

Abstract. Most of the state-of-the-art MapReduce-based entity matching methods inherit traditional Entity Resolution techniques on centralized system and focus on data blocking strategies for structured entities in order to solve the load balancing problem occurred in distributed environment. In this paper, we propose a MapReduce-based entity matching framework for Entity Matching on semi-structured and unstructured data. Each entity is represented by a high dimensional vector generated from description data. In order to reduce network transmission, we produce lower dimensional bit-vectors called signatures for those entity vectors based on Locality Sensitive Hash (LSH) function. Our LSH is required for promising cosine similarity. A series of random algorithms are designed to ensure the performance for entity matching. Moreover, our design contains a solution for reducing redundant computation by one round of additional MapReduce job. Experiments show that our approach has a huge advantages on both processing speed and accuracy compared to the other methods.

1 Introduction

Nowadays, the rapid growth of web data and User Generated Content (UGC) changes the way we used to collect and manage information. By the hands of the huge amount of web users, data become much easier to be generated. For instance, in a C2C (Customer to Customer) online business site, it becomes easy to start an online store and generate personalized structured/unstructured descriptions for its listed goods. And it is pervasive that different sellers own the same commodity with variety of descriptions together with diverse schemas. This results in the difficulty in product managing which may affect product or price comparison. Additionally, this kind of UGC is becoming large. Then it is urgent to design an efficient distributed entity matching framework by using of this UGC to identify entities that represent the same items.

Though there are already a bunch of entity matching algorithms, we face new challenges which make the traditional methods infeasible. First of all, most of the

⋆ Corresponding author.

R. Cheng et al. (Eds.): APWeb 2015, LNCS 9313, pp. 509–521, 2015.
DOI: 10.1007/978-3-319-25255-1_42

user generated data are semi-structured or unstructured data, which come from variety data sources and in different formats/schemas. Second, a great quantity of typos occurred in UGC data dramatically reduce the data quality. Third, high computation cost occurs due to the large web data size. Traditional well designed entity matching algorithms are usually for structured data. When they confront with these three challenges including hybrid data structure without uniform schema, low data quality and huge data size, their performance is also challenged.

Lots of works have tried resolving those challenges separately: 1) Document similarity metrics[20] are introduced to measure the similarity between unstructured data, such as online documents. They provide standards of similarity measurements for unstructured data and semi-structured data. 2) Tokenization technique[18] is used to reduce the negative influence to data quality caused by typos and human mistakes. It has become an important step in data cleaning and for improving the accuracy of entity matching. 3) Data blocking strategies are designed to split data into multiple parts for parallelization and lowering computation cost. Inspired by those work, we propose a flexible parallel entity matching framework on MapReduce, which aims to resolve entity matching on unstructured data with higher efficiency and lower cost. That is for these unstructured data, our algorithm shall boost the processing speed while promise load balance and reduce network transmission cost. Those are the

The main contributions can be summarized as follows:

- We sketch out a random-based framework for entity matching based on MapReduce, and introduce our random algorithm based on this framework. We propose to generate low dimensional bit-vector signatures for entities, that are calculated from the high dimensional feature vectors by a specific Locality Sensitive Hash (LSH) function[16]. It helps to boost computation efficiency and reduce storage and network transmission costs dramatically.
- We propose a random-based permutation method inspired by PLEB[6] algorithm for increasing match accuracy, which can be well adapted on MapReduce framework. Besides, the permutation method ensures load balance during the matching process.
- We analyze the cause of redundancy problem in our algorithm, which is pervasive in many of the blocking-based algorithms. We solve this problem by adding an extra MapReduce job to reduce the redundancy rate.
- We evaluate our approaches and demonstrate their efficiency in comparison to existing blocking-based matching methods on both semi-structured and unstructured data.

2 Related Work

The idea of Entity matching (along with Record linkage and deduplication) is first introduced by geneticist Howard Newcombe in [15] who presents odds ratios of frequencies and the decision rules for delineating matches and mismatches.

Fellegi and Sunter[4] provide the formal mathematical foundations of record linkage. At the end of 20th century, since the data size grew rapidly, the main problem for entity matching changed from improving calculation accuracy to handling huge amount of data. Blocking strategy was introduced to solve this problem for it can filter majority of the entity pairs with low similarity before similarity comparison. Meanwhile, the proposal of MapReduce[3] also gave us a better platform to solve this problem.

A number of blocking-based entity matching algorithms under MapReduce framework have been presented to help dealing with big data sets[7,8,19,14]. These works are based on a assumption that there is only one key for an entity and use a map/reduce phase to handle the problem. They design different blocking strategies for map phase and then do the matching step in reduce phase. Some of the most influential works includes sorted neighborhood[12] and load-balanced entity matching[11,10]. The sorted neighborhood is a blocking technique by sorting all entities according to blocking keys, assigning a window size w and comparing entities in the window while sliding. This gives us some inspiration of the pair generation method. However, this part of work and its joint works[9] didn't mention the load balancing problem on MapReduce.

Both *BlockSplit* and *PairRange* blocking strategies mentioned in [11] focus on solving the imbalance problem. But they rely on a data analysis phase before matching job. This phase scan the input entities to collect a list of all possible pairs, then make a blocking plan to evenly divide those pairs into multiple blocks by the above two strategies. It can perfectly solve the load balancing problem on MapReduce by adding an expensive cost before matching process. Therefore, both of these two strategies are suitable for processing skewed data, but far more slow to deal with regular data or data with enormous size. Another algorithm has been presented for document-similarity computation[1] which have the same background to our work, we will compare our performance with this algorithm.

3 MapReduce-Based Entity Matching Framework

In order to get good matching accuracy for high dimensional data, the distributed entity matching framework is expected to have the following functionalities: a) Fast Entity Similarity Calculation method, b) Efficient Candidate Entity Pair Generation algorithm and c) Redundant pairs removing plan. Figure 1 shows the framework of our algorithm.

3.1 Random-Based Similarity Calculation

Cosine similarity is appropriate for measuring similarity of semi-structured and unstructured entities. However, high dimensional vectors may cause the dimensional-curse on entity matching calculation. Locality Sensitive Hashing (LSH) function which keep the property of cosine similarity, proposed by Charikar in [2], provides an option for high dimensional vectors similarity calculation distributively.

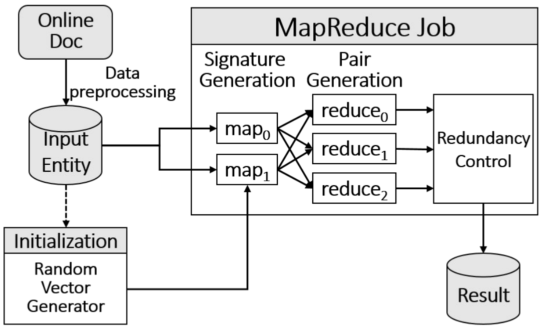


Fig. 1. Framework of random-based entity matching on MapReduce

Theorem: Suppose we have a collection of vectors in a k dimensional vector space (that is N^k). We generate a random vector r of unit length from this k dimensional space, and define a hash function h_r as Eqn.1. Then for vectors u and v, we have the corresponding relationship as calculated by Eqn.2. Goemans and Williamson[5] prove that this hash function can promise the cosine similarity in a high probability.

Since high dimensional vectors computation is time-consuming, dimension reducing is generally done for improving calculation performance. Note that the above equation is probabilistic in nature. Hence, we start to generate d numbers of random vectors from N^k and get R={r}, with $|R| = d \ and \ d \ll k$. For each vector u in N^k, we can get a d-bit vector $\{h_{r1}(u), h_{r2}(u), ..., h_{rd}(u)\}$ by using hash function $h_r(u)$ for u and each vector $r_i \in R$. In such a way, we represent each vector u by a d-bit signature S_u. Then dimension k is successfully reduced to d whilst it still preserves the cosine similarity, where $d \ll k$. This d-dimension signature keeps features of the original vector. Then the huge deviation between two signatures means big difference between two entity vectors.

Cosine similarity between any two vectors is achieved by Eqn. 3. On the other hand, if we use the similarity between signatures to represent the probability in Eqn. 3, we can observe:

$Pr[h_r(u) = h_r(v)] = 1 - (hamming \ distance)/d.$

Thus, it converts the problem of finding cosine similarity between vectors to the problem of calculating hamming distance between signatures. It is more efficient in both processing speed and memory utilization. So in the following descriptions, hamming distance has the same meaning as cosine similarity.

$$h_r(u) = \begin{cases} 1 & r.u \geqslant 0 \\ 0 & r.u < 0 \end{cases} \quad (1)$$

$$Pr[h_r(u) = h_r(v)] = 1 - \frac{\theta(u,v)}{\pi} \quad (2)$$

$$\cos(\theta(u,v)) = \cos(1 - Pr[h_r(u) = h_r(v)])\pi \quad (3)$$

3.2 Entity Pairs Generation

As shown above, highly similar entities usually have similar signatures. Then sorting by lexicography will make them close to each other except the deviations occur in the first few bits of their signatures. In order to reduce this kind of deviation and increase the opportunity of getting close for similar signatures, we propose to do random permutation for these signatures as proposed by PLEB (Point Location in Equal Balls), which was first mentioned in [6] and improved in [2]. This algorithm takes random permutations to signatures and sort the permuted signatures. It aims to find vectors with short hamming distances.

A random permutation is considered as a random jumble of the bits of each signature, so the prefix deviation of two similar signatures can be prevented in some of their permutations. We apply several rounds of permutations on signature set. At each round, it generates a set of permuted signatures. Signature pairs with lower hamming distance are expected to get close in some of the sorting lists. Accordingly, we can find the top m closest neighbors for each signature and generate our entity pair candidates. Implementation details can be found in the following content.

3.3 Entity Matching on MapReduce

We aim at solving the entity matching problem on semi-structured and unstructured data. We first tokenizing the input data and generating high dimensional feature vectors based on the words frequency. We use these vectors as our data input. The goal is to find all matching pairs among these entity vectors. We define that two entities are matched when the hamming distance between their feature vectors is lower than a predefined threshold. We can also output the top N similar vectors by an additional sorting process on the result set.

Figure 1 shows matching framework on MapReduce. Before doing the matching process, we carry out three steps of preprocessing on the source data to get our expected input. Initially, we split the input entities into tokens using the Part-Of-Speech Tagger[17]. Then we generate a dictionary containing all k different tokens occurred in the data set. Finally, for each entity u, a k-dimension vector V_u is generated, in which the nth dimension represent the word frequency of the nth token in entity u. The input of our method is a set of (*key*, *value*) pairs made up of entity ID E_u and its k-dimension vector V_u. In addition, another standard vector set R is introduced into the MapReduce job, which contains d ($d << k$) numbers of k bits random vectors of unit length, that is $\{r_1, r_2, ..., r_d\}$.

Figure 2 illustrates the workflow of our MapReduce job. Map phase contains steps as followings:

1. Initially, we apply $h_r(u)$ to each input entity (E_u, V_u) using R as the standard vector set. We pair V_u with every vector r_i in R ($1 \leqslant i \leqslant d$) and calculate $h_{ri}(u)$, then we get a d bit binary signature S_u for V_u represented as: $\overline{S_u} = \{h_{r1}(u), h_{r2}(u), ..., h_{rd}(u)\}$.
2. After converting the input vectors into signatures S_u, we randomly permute them t times. The permutation function can be approximated as:

$$\pi(x) = (ax + b)\ mod\ p \tag{4}$$

where p is prime and $0 < a < p, 0 \leq b < p$. Both a and b are chosen randomly. We apply t different random permutation for every signature (by

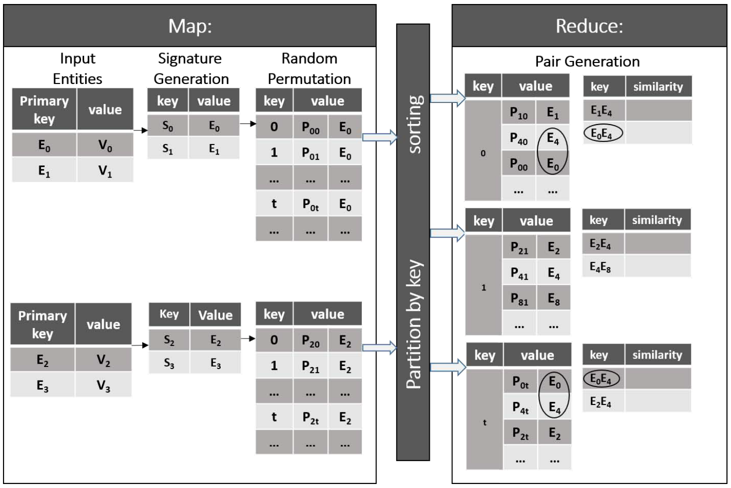


Fig. 2. Example of random-based matching algorithm on MapReduce

choosing random values for a and b, t number of times). Thus for each signature S_u, we have t different permutation results: $\{P_{u1}, P_{u2}, ..., P_{ut}\}$. We regard this result as our map output. As a consequence, we have t different map output for each entity represented as (i, P_{ui}, E_u), with i, P_{ui} and E_u referring permutation number, the corresponding permutation result and the entity ID.

In reduce phase, we expect to achieve entity pair similarity. After an automatic sorting procedure during the shuffle between map and reduce, the reduce phase faces t number of groups represented as (i, L_i) , in which L_i is the sorted list on all signatures in the i_{th} round of permutation. Then we generate matching pairs between every entity u and its closest m neighbors in the sorted list. Finally, we calculate the hamming distance of every paired entities and output those with distance below a predefined threshold. The output is formatted as $(E_uE_v, similarity)(u < v)$, which are the ID concatenation of paired entities' with its similarity value.

Overall, The map tasks change each k-dimension vector into t number of d-bit signatures. d and t are always far less than k. Generally d and t are between tens to hundreds while k are normally more than tens of thousands determined by the characteristic and size of input data. That gives a significant reduction on data volume, and also a huge cut on the network transmission cost between map and reduce. Unlike most of the blocking-based entity matching methods comprised by multiple MapReduce tasks, our matching algorithm is finished in one MapReduce job. Since each MapReduce task spends extra cost on task scheduling and network communication, the cutting on the number of MapReduce jobs can lead to performance promotion. Furthermore, since all permutations of a signature are sent to reducers evenly, which are partitioned by their permutation number i.

There are the same number of pairs for each reduce task. Therefore, our model easily solve the load balancing problem.

4 Redundancy Reduction

During the pair generation step in reduce phase, all pairs are generated in parallel from different groups. As the same entity pair may be generated from multiple groups, there can be many duplicated pairs, such as the circled pairs E_0E_4 and E_2E_4 in Figure 2. It may cause redundant computation cost, which is a pervasive problem in many MapReduce-based matching algorithms. The reason for the occurrence of these redundancy is that the features of one entity are separated into multiple parts during the map phase, and each part may possibly match any part of the other entity in the reduce phase. It may happen more frequently among these highly similar entity pairs, because of the higher possibility for each part to be matched.

In redundancy-free similarity computation model [13], it adds additional annotate on each map output to tell the reducer which reducers the rest parts of this entity will be sent to. Though it is efficient, it bases on a strong precondition that all entities sent to the same reducer will definitely be paired among each others. In our algorithm, the permuted signatures of the same entity is sent to all reducers, and for each reducer a signature is only paired with its neighborhoods. So this redundancy-free solution is inapplicable for our random-based method.

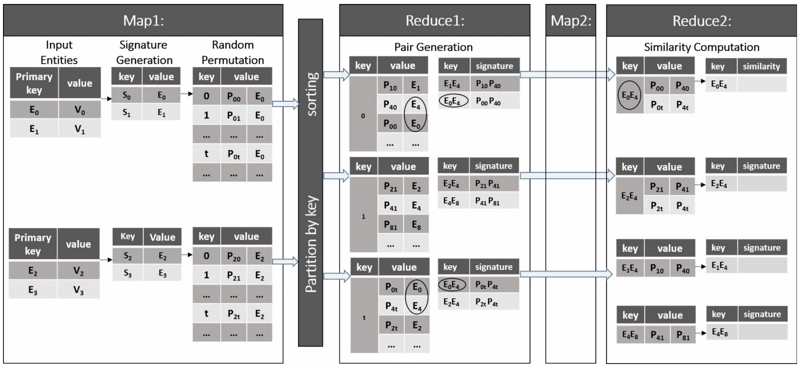


Fig. 3. Example of the deduplication method

We introduce an extra MapReduce job to reduce duplication. Figure 3 shows an example of our method. We modify the original MapReduce job in the reduce phase by cutting off the similarity computation. After generating all pairs, the reduce task terminates and outputs those pairwise information with entity IDs E_u, E_v $(u < v)$ concatenated as *key* and their ith permuted signatures P_{ui}, P_{vi} concatenated as *value*, that is $(E_uE_v, P_{ui}P_{vi})$.

The map phase of the second MapReduce job is an identity mapper which does nothing. In the following shuffle phase, all pairs with the same entity ID are grouped together. It means that all those duplicated pairs come together.

Then reducers will process the data like $(E_uE_v, list(P_{u1}P_{v1}, P_{u4}P_{v4}, ..., P_{ut}P_{vt}))$. Then we need to pick one pair of permuted signatures in the list and calculate its hamming distance. At last, we output the pair like $(E_uE_v, similarity)$ as final result.

5 Experiments

We run experiments on a 22-node HP blade cluster. Each node has two Intel Xeon processors E5335 2.00GHz with four cores and one thread per core, 16GB of RAM, and two 1TB hard disks. All nodes run CentOS 6.5, Hadoop 1.2.1, and Java 1.7.0. We evaluate the performance of our algorithm in two aspects: 1) We measure the effect on calculation performance and matching quality by using different parameters. 2) We compare the performance of our algorithm with two other state-of-the-art matching algorithms namely Document Similarity Self-Join (DSSJ)[1] and Dedoop[10].

5.1 Data Set Description

We use CiteSeerX data set. It contains nearly 1.32 Million citations of total size 2.89 GB in XML format. Each citation includes structured attributes such as *record ID, author, title, date, page, volume, publisher, etc.* It also has document *abstract.* We select a few records from CiteseerX and manually make validation sets for accuracy evaluation.

5.2 Parameter Description and Evaluation Metrics

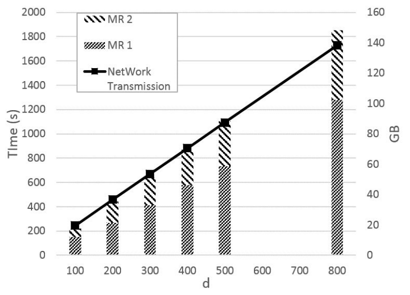
There are three parameters that may affect the performance:

- **d**: The length of the signature. It directly determines the network transmission cost and accuracy. A bigger d leads to a longer signature, and therefore increases the burden of network transmission, but it can benefit the accuracy since the signature may contain more information of the entity.
- **t**: The number of permutations. It multiplies the data transmission between map and reduce. The increase of t can also raise the pair redundancy and increase the run-time, but improve the matching accuracy.
- **m**: The window of selecting neighborhoods. It decides the amount of pairs and also causes a change on redundancy ratio. It can influence the result accuracy and execution time as well.

We introduce four metrics to evaluate system performance:

- The **network transmission cost** is measured by summing up the size of map output since all output of map phase will be sent to reducers through network.
- The **run-time** of MapReduce jobs is recorded to compare the speed of our algorithm with different parameters.

– The **redundancy rate** is calculated as total number of generated pairs / distinct candidate pairs to show the redundancy ratio.
– The **accuracy** is also measured in this part. In order to calculate the accuracy, we prepare a validation set which contains 200 entity records for accuracy measurement. We calculate the similarity between those entities manually and generate a set of top 50 similar entity pairs as the standard result set. The accuracy is measured as the fraction of pairs that appear within the standard top 50 results.



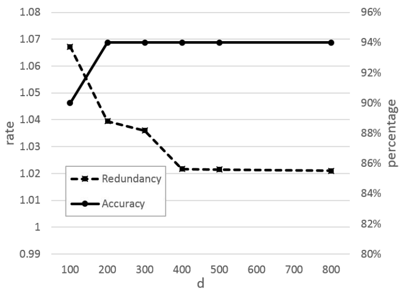
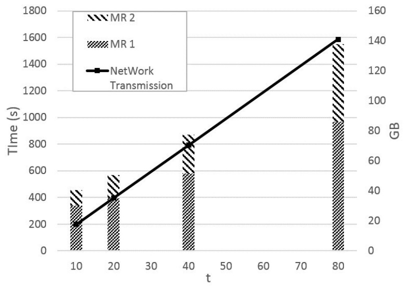


Fig. 4. Run-time and network transmission for different value of d (t=40, m=8)

Fig. 5. Redundancy rate and accuracy for different value of d (t=40, m=8)



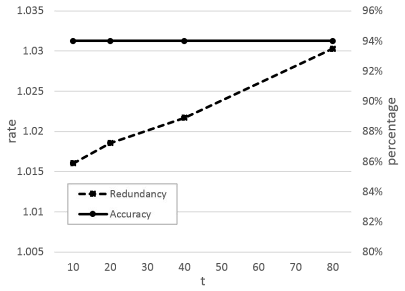


Fig. 6. Run-time and network transmission for different value of t (d=400, m=8)

Fig. 7. Redundancy rate and accuracy for different value of t (d=400, m=8)

To evaluate the performance, we first use a 200MB subset of CiteSeerX as our input to the effect of changing parameters. Figure 4 to 9 show the performance variations of our algorithm when changing one of the three parameters. Figures 4, 6, and 8 illustrate the run-time for both of two MapReduce jobs with MR1 for pair generation and MR2 for deduplication, and the transmission cost during MapReduce tasks. We can see from Figure 4 that when we increase the length of signature d, the network transmission cost together with the run-time

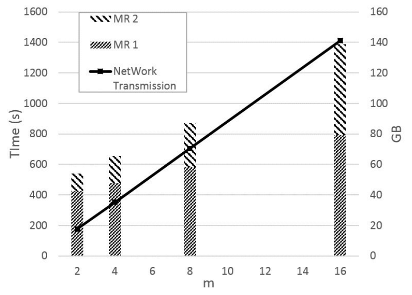


Fig. 8. Run-time and network transmission for different value of m(d=400, t=40)

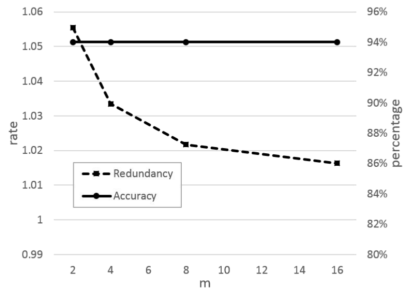


Fig. 9. Redundancy rate and accuracy for different value of m (d=400, t=40)

of MapReduce jobs has a linear growth. Meanwhile, in Figure 5, as the increasing of d,the redundancy decreases steadily and the accuracy increases smoothly. The reason is that as the signature extends, the differences between entities can be found more easily. So they may have fewer chances to be paired. Therefore, the redundancy rate decreases. The performance variations with changing t and m are listed in Figure 6 to 9. Figure 6 and 8 shows similar performance with Figure 4, but it seems that in Figure 8 we spend large part of time on MR1 when $m = 2$ or $m = 4$. It is because a smaller m cannot reduce the cost on map phase when generating signatures and doing permutations. Figure 7 clearly shows that the number of permutations t can determine the redundancy rate directly. All these three parameters can strongly influence the performance. Figure4,6,8 show that when $d \geqslant 400, t \leqslant 50$ and $m \geqslant 8$, we get a better performance on both redundancy rate and accuracy. So we choose d=500, t=50 and m=10 to do the rest of evaluations. Furthermore, the linear growth of run-time shows a good scalability of our matching method, which has a huge advantage in processing big data set.

5.3 Different System Comparison

The baseline methods for our comparison are Document Similarity Self-Join (DSSJ) and Dedoop.

Table 1. Comparison of accuracy with DSSJ (d=500,t=50,m=10)

Name	Top 10	Top 20	Top 50
DSSJ	90%	95%	94%
Random-base Matching	90%	100%	94%
Dedoop	100%	100%	100%

In order to measure the accuracy, we use the validation set mentioned previously, and compare our matching result with DSSJ and Dedoop results. The standard result tests contain top 10, 20 and 50. Table 1 shows the accuracy of these top N tests. Since Dedoop compares all possible pairs of entities and calculates cosine similarity directly, the similarity result of Dedoop are always correct. However, the transmission cost is problematic that will be analyzed in the following content. When using the parameters (d=500,t=50,m=10), we can get almost the same accuracy as DSSJ method does. At the same time, we evaluate the processing speed of our method comparing with Dedoop and DSSJ using these parameters. Figure 10 shows run-time between our method and the baseline methods. We just run a small size of data since the run-time of both Dedoop and DSSJ would exceed hours if the size of data is larger than 100MB. The reason is that both of these two methods generate enormous size of pair candidates. We get nearly 100GB map output data when running Dedoop on 200MB data set. It can burden the system drastically on network transmission and is hard to be processed in memory. On the contrary, our random-based matching method shows a good scalability on data set size. The speed of our algorithm is significantly faster than Dedoop, and far more stable even when dealing with gigabytes of input data.

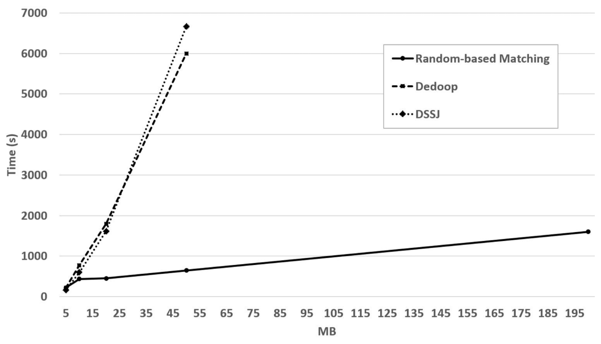


Fig. 10. Comparison of run-time with Dedoop and DSSJ (d=500,t=50,m=10)

6 Conclusion

In this paper, we study the problem of entity matching on unstructured data, which will be formatted as high-dimensional feature vectors. We take the MapReduce framework as our programming model and point out the two major challenges met on this model, which are load balancing problem and network transmission cost. We propose a random-based matching method to solve the matching problem which will help to greatly reduce the transmission cost. We use a special LSH function to generate signatures for entities, which helps to reduce entity dimensions. We take PLEB fast search algorithm to generate the candidate pairs efficiently. In addition, we propose our approach to reduce the redundancy during reduce tasks. Given the proposed algorithm, we implement it in Hadoop and analyze its performance on real data sets.

Acknowledgements. This work is partially supported by National Basic Research Program of Chi- na (Grant No. 2012CB316200), National Science Foundation of China (Grant No.61232002, 61402177 and 61332006).

References

1. Baraglia, R., De Francisci Morales, G., Lucchese, C.: Document similarity self-join with mapreduce. In: 2010 IEEE 10th International Conference on Data Mining (ICDM), pp. 731–736. IEEE (2010)
2. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-Fourth Annual ACM symposium on Theory of Computing, pp. 380–388. ACM (2002)
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
4. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. Journal of the American Statistical Association 64(328), 1183–1210 (1969)
5. Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM (JACM) 42(6), 1115–1145 (1995)
6. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613. ACM (1998)
7. Kiefer, T., Volk, P.B., Lehner, W.: Pairwise element computation with mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 826–833. ACM (2010)
8. Kim, Y., Shim, K.: Parallel top-k similarity join algorithms using mapreduce. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 510–521. IEEE (2012)
9. Kolb, L., Thor, A., Rahm, E.: Parallel sorted neighborhood blocking with mapreduce. arXiv preprint arXiv:1010.3053 (2010)
10. Kolb, L., Thor, A., Rahm, E.: Dedoop: efficient deduplication with hadoop. Proceedings of the VLDB Endowment 5(12), 1878–1881 (2012)
11. Kolb, L., Thor, A., Rahm, E.: Load balancing for mapreduce-based entity resolution. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 618–629. IEEE (2012)
12. Kolb, L., Thor, A., Rahm, E.: Multi-pass sorted neighborhood blocking with mapreduce. Computer Science-Research and Development 27(1), 45–63 (2012)
13. Kolb, L., Thor, A., Rahm, E.: Don't match twice: redundancy-free similarity computation with mapreduce. In: Proceedings of the Second Workshop on Data Analytics in the Cloud, pp. 1–5. ACM (2013)
14. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using mapreduce. Proceedings of the VLDB Endowment 5(10), 1016–1027 (2012)
15. Newcombe, H., Kennedy, J., Axford, S., James, A.: Automatic linkage of vital records (1959)
16. Ravichandran, D., Pantel, P., Hovy, E.: Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics, pp. 622–629. Association for Computational Linguistics (2005)

17. Toutanova, K., Klein, D., Manning, C.D., Singer, Y.: Feature-rich part-of-speech tagging with a cyclic dependency network. In: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, vol. 1, pp. 173–180. Association for Computational Linguistics (2003)
18. Toutanova, K., Manning, C.D.: Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In: Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics, vol. 13, pp. 63–70. Association for Computational Linguistics (2000)
19. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 495–506. ACM (2010)
20. Zobel, J., Moffat, A.: Exploring the similarity space. SIGIR Forum 32(1), 18–34 (1998)