A Load-Balanced MapReduce Algorithm for Blocking-based Entity-resolution with Multiple Keys

Sue-Chen Hsueh\textsuperscript{1}, Ming-Yen Lin\textsuperscript{2*}, and Yi-Chun Chiu\textsuperscript{2}

\textsuperscript{1}Dept. of Information Management, Chaoyang University of Technology, Taichung, Taiwan
\textsuperscript{2}Dept. of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan

schueh@cyut.edu.tw; linmy@mail.fcu.edu.tw; ojean913@gmail.com

Abstract

Entity resolution (ER), which detects records referring to the same entity across data sources, is a long-lasting challenge in database management research. The sheer volume of data collections today calls for the need of a blocking-based ER algorithm using the MapReduce framework for cloud computing. Most studies on blocking-based ER assume that only one blocking key is associated with an entity. An entity in reality may have multiple blocking keys in some applications. When the entities have a number of blocking keys, ER can be more efficient since two entities can form a similar pair only if they share several common keys. Therefore, we propose a MapReduce algorithm to solve the ER problem for a huge collection of entities with multiple keys. The algorithm is characterized in the combination-based blocking and the load-balanced matching. The combination-based blocking utilizes the multiple keys to sort out necessary entity pairs for future matching. The load-balanced matching evenly distributes the required similarity computations to all the reducers in the matching step so as to remove the bottleneck of skewed matching computations for a single node in a MapReduce framework. Our experiments using the well-known CiteSeerX digital library show that the proposed algorithm is both efficient and scalable.

Keywords: Entity resolution, cloud computing, MapReduce, load-balance.

1 Introduction

Entity resolution (abbreviated as ER), also known as data matching, de-duplication, identity resolution, or record linkage, is to identify all the manifestations referring to the same entity (Hernández and Stolfo 1995; Brizan and Tansel 2006; Chaudhuri, Ganti, and Kaushik 2006; Elsayed et al. 2008; Vernica et al. 2010; Zhang et al. 2010). ER is crucial for data integration and has been applied in many applications including price comparisons, citation matching (Pasula et al. 2012), etc. Take a price-comparison website for example, to provide all the well-known CiteSeerX digital library show that the total number of similarity computations can be massive, considering the scale of web-pages nowadays.

In general, a blocking-based ER is used to accelerate the similarity computations. Each entity in the blocking-based ER has an attribute or a specific value computed by hashing or other means called blocking key. The blocking-based ER comprises a blocking step and a matching step. The blocking step partitions all the entities in the dataset by blocking keys into several groups called blocks, where the entities in a block have the same blocking key. After that, the matching step performs similarity computations for all the entity pairs in each block and determines the similarities between entities within each block. The result is a set of all the entity pairs of high similarity. The blocking key in fact reduces the total number of similarity computations required in the blocking-based ER.

Most of the studies on blocking-based ER assume that there is only one blocking key associated with an entity. In reality, an entity may have several blocking keys. An entity can be a record in a database, a web-page in a web-site, an article in a research repository, etc. A product page may be classified to multiple categories so as to have several attribute keys; an article in DBLP or CiteSeerX generally contains many keywords or index terms. In addition, the titles and some attributes like authors of the articles can serve as blocking keys, too. When the entities have a number of blocking keys, blocking-based ER can be performed more efficient since two entities can be a similar pair only if they share several common keys.

Nevertheless, the blocking step of ER will become more time-consuming if multiple blocking keys are treated as many individual blocking keys, which consideration is adopted in most ER algorithms today. Traditional solution generates a pair for further matching if two entities share a common blocking key. Because there are more than one key associated with an entity now, the possibility of sharing common keys with other entities, which also contain many keys, is greatly increased. Consequently, the number of entity pairs generated for matching will be greatly increased. We consider that blocking keys represent specific features of entities, multiple blocking keys can be used in the blocking step to sort out the entities need to be matched so that the matching step can be accelerated.

A typical ER problem is the pairwise document similarity (Baraglia et al. 2010; Lin 2009; Elsayed et al. 2008), which detects similar documents in a collection like DBLP, CiteSeerX, or Google Scholar. The collection of articles is continuously growing as research articles keep...
appending in a surprising speed. The ER process performed by a single machine suffers from the vast amount of the increasing data. The total number of similarity computations for the potential pairs becomes very huge. For example, 1.4 million publication records in the CiteSeerX collection will require 979 billion similarity computations. The problem is exacerbated if the data size accumulates up to terabytes or petabytes. Thus, it is necessary to devise a blocking-based ER algorithm using the MapReduce framework (Dean and Ghemawat 2004) in an open architecture like Hadoop (Hadoop 2012).

Although the blocking-based ER using the MapReduce framework may solve the problem of huge dataset size, the process may still suffer from the imbalance of key distributions. Commonly, the map phase and the reduce phase are the two core processes of a MapReduce program. An input record is a form of (key, value) pairs and the key may come from a different domain. The process responsible for the map function is referred to a mapper and that for the reduce function is referred to a reducer for convenience. A mapper reads one block of (key, value) pairs, and produces a list of intermediate (key’, value’) pairs after processing. A reducer accepts the intermediate key’ with corresponding list of value’s, processes and generates the output. A default hash partitioning is used in the MapReduce framework to partition the intermediate (key’, value’) pairs to the reducers. Each reducer might be responsible for its respective 100 (key’, value’) pairs when there are 10000 keys and 100 reducers. If certain keys are more popular than the others, a large number of (key’, value’) pairs will be sent to the same reducer. A reducer accepts the intermediate key’ with corresponding list of value’s, processes and generates the output.

Therefore, in this paper, we propose a MapReduce algorithm to solve the ER problem for a huge collection of entities with multiple keys. The algorithm features in the combination-based blocking and the load-balanced matching. The combination-based blocking utilizes the multiple keys to filter out unnecessary entity pairs. The load-balanced matching evenly distributes the required similarity computations to all the reducers in the matching step. Our experiments using CiteSeerX show that the proposed algorithm is efficient and scales up linearly with respect to the dataset size.

The rest of the paper is organized as follows. Section 2 defines our problem. Related works are briefly reviewed in Section 3. Section 4 presents the proposed algorithm. The experimental results are given in Section 5. Section 6 concludes this study.

2 Problem Definition

A dataset \( D = \{E_1, E_2, \ldots, E_m\} \) contains \( m \) entities, where an entity \( E_i \) (\( 1 \leq i \leq m \)) has \( |E_i| \) blocking keys. The blocking keys of \( E_i \) is represented by \( B_{E_i} = \{k_1, k_2, \ldots, k_{|E_i|}\} \) and \( |E_i| > 1 \). Let the minimum number of blocking keys of the entities in \( D \) be \( kc, kc > 1 \). Given a similarity measure and a similarity threshold \( \theta \), the ER problem is to find out all the similar entity pairs in \( D \). An entity pair \( (E_i, E_j) \) is considered similar if \( \text{sim}(E_i, E_j) \geq \theta \), where \( \text{sim}(E_i, E_j) \) is the similarity value between \( E_i \) and \( E_j \) computed using the similarity measure. Particularly, the characteristics of blocking keys implies, in our problem, that \( \text{sim}(E_i, E_j) < 0 \) if \( |B_{E_i} \cap B_{E_j}| < kc \). That is, \( E_i \) and \( E_j \) cannot be similar if the number of common blocking keys between them is less than \( kc \). Accordingly, the similarity computation \( \text{sim}(E_i, E_j) \) can be eliminated in the ER process if \( |B_{E_i} \cap B_{E_j}| < kc \).

A typical similarity measure between \( E_i \) and \( E_j \) is the Jaccard similarity coefficient, which is defined as the size of their intersection divided by the size of their union. Assume the entities are text research articles, the Jaccard similarity coefficient can be calculated by the number of common terms divided by the total number of terms in the two articles. The blocking keys of the entities can be title words or keywords of the articles.

For example, given \( D = \{E_1, E_2, \ldots, E_{100}\} \) and \( kc = 2 \). We have 100 entities and the minimum number of blocking keys of the entities in \( D \) is 2. Assume \( E_1 \) has 4 blocking keys \( B_{E_1} = \{a, c, d, e\} \), \( E_2 \) has 3 blocking keys \( B_{E_2} = \{a, c, e\} \), and \( E_3 \) has 2 blocking keys \( B_{E_3} = \{d, e\} \), etc. Then \( |E_1| = 4, |E_2| = 3, \text{and } |E_3| = 2 \), \( B_{E_1} \cap B_{E_2} = \{a, c, e\} \), \( B_{E_1} \cap B_{E_3} = \{d, e\} \), and \( B_{E_2} \cap B_{E_3} = \{e\} \). The similarity computations need to be performed on both \( (E_1, E_2) \) and \( (E_1, E_3) \) but not \( (E_2, E_3) \) since \( |B_{E_2} \cap B_{E_3}| = 1 < kc \). If the similarity threshold \( \theta = 0.8 \), \( \text{sim}(E_1, E_2) = 0.85 \), and \( \text{sim}(E_1, E_3) = 0.7 \), entity pair \( (E_1, E_2) \) is a similar pair and is inserted into the answer set.

Note that the ER problem definition is the same as others. Nevertheless, our problem emphasizes on the variable number of blocking keys in entities. Only entities having certain number of common keys are potentially similar.

3 Related Works

Although the ER problem is a long-existing problem, most solutions are not designed for extremely large collection of entities. Some algorithms have been presented for document-similarity computations (Baraglia et al. 2010) and blocking-based ER solution under the MapReduce framework (Kiefer et al. 2010; Kim and Shim 2012; Lu et al. 2012; Metwally and Faloutsos 2012; Vernica et al. 2010; Zhang et al. 2010). These works assume that there is a unique similarity threshold \( \theta \) for the blocking step and then a reduce function for the matching step. Some of the notable works includes sorted neighborhood (Kolb et al. 2011a) and load-balanced ER (Kolb et al. 2012a; Kolb et al. 2012b).

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. Some entities with similar but not same blocking keys might be compared for similarity evaluations. In (Kolb et al. 2011a; Kolb et al. 2011b), the sorted neighborhood is implemented under the MapReduce framework while
entities crossing boundaries have to be duplicated to avoid missing potential pairs due to reducers’ limitations (Kolb et al. 2011a). Both JobSN using two MapReduce phases and RepSN using one MapReduce phase are presented. However, the problem of imbalanced loads among reducers in MapReduce is not mentioned.

BlockSplit and PairRange (Kolb et al. 2012b) consider the load-balancing problem using the MapReduce framework for single key ER. A block distribution matrix needs to be distributed using the distributedCache technique in MapReduce so that the entities can be evenly distributed to all the reducers. PairRange outperforms BlockSplit because PairRange guarantees that all reducers may receive the same number of entities (Kolb et al. 2012b).

The study in (Kolb et al. 2011a) indicates that multiple keys may exist in an entity. However, the multiple keys are treated as independent like several single keys so that duplicate distributions appear in the blocking step. The MultiRepSN (Kolb et al. 2011a) is proposed to overcome an entity with multiple blocking keys. Although reducer-loads are balanced, the algorithm might suffer from the problem of largely duplicated entity pairs.

Thus, the study (Kolb et al. 2013) presents an algorithm for ER with redundancy-free matching. The idea is to enumerate all candidate sets of entities with the index of the smallest common candidate sets. The mapper will emit (blocking keys, entity values) for the reducers. The entity value includes the entity and the smallest common blocking keys. Thus, Redundancy-Free Matching (Kolb et al. 2013; Kolb and Rahm 2013) may decrease the number of duplicate entity pairs. However, when a reducer receives an overwhelming entities, the execution time can be long. Therefore, the issue of load-balancing among reducers must be considered for an ER solution under the MapReduce framework with multiple keys.

4 The Proposed Algorithm

4.1 Overview of the Algorithm

The proposed algorithm for blocking-based ER utilizes the multiple blocking keys in each entity for an improved entity distribution in the blocking step. The distribution is more precise because an entity pair may exist in a block only when the number of common blocking keys between the pair exceeds certain threshold (i.e. \( k_c \)). Because an entity may have more than \( k_c \) keys, it needs to generate all the combinations of \( k_c \) keys for potential key comparisons. The entity distribution procedure is called combination-based blocking in the proposed algorithm.

After the blocking step, we aim to balance the working load among all the reducers for similarity computations in the matching step. The idea is to obtain a statistics of the total number of computations required for all the blocks first. We then evenly partition the computations among all the reducers to avoid potential overload of a reducer due to skewed key distributions. The procedure is called load-balanced matching in the proposed algorithm. Both combination-based blocking and load-balanced matching are designed using the MapReduce framework. Therefore, the algorithm comprises two map/reduce phases: a map/reduce phase for combination-based blocking, followed by a map/reduce phase for load-balanced matching. An overview of the proposed algorithm is shown in Fig. 1.

Fig. 1: An Overview of the Proposed Algorithm

The proposed algorithm for blocking-based ER utilizes the multiple blocking keys in each entity for an improved entity distribution in the blocking step. The distribution is more precise because an entity pair may exist in a block only when the number of common blocking keys between the pair exceeds certain threshold (i.e. \( k_c \)). Because an entity may have more than \( k_c \) keys, it needs to generate all the combinations of \( k_c \) keys for potential key comparisons. The entity distribution procedure is called combination-based blocking in the proposed algorithm.

After the blocking step, we aim to balance the working load among all the reducers for similarity computations in the matching step. The idea is to obtain a statistics of the total number of computations required for all the blocks first. We then evenly partition the computations among all the reducers to avoid potential overload of a reducer due to skewed key distributions. The procedure is called load-balanced matching in the proposed algorithm. Both combination-based blocking and load-balanced matching are designed using the MapReduce framework. Therefore, the algorithm comprises two map/reduce phases: a map/reduce phase for combination-based blocking, followed by a map/reduce phase for load-balanced matching. An overview of the proposed algorithm is shown in Fig. 1.
blocking keys. Each combination is outputted with the id of the entity. For example, \( <(a,c), 2> \), \(<(a,e), 2> \), and \(<(c,e), 2> \) are outputted for entity \( E_2 \) with \( B_{E_2} = \{ a, c, e \} \) when \( kc = 2 \); \(<(a, c, e), 2> \) is outputted for \( E_2 \) and \(<(a, c, d), 1> \), \(<(a, c, e), 1> \), \(<(a, d, e), 1> \), and \(<(c, d, e), 1> \) are outputted for \( E_1 \) with \( B_{E_1} = \{ a, c, d, e \} \) when \( kc = 3 \). In fact, \( E_1 \) and \( E_2 \) need to be compared since they have three common blocking keys when \( kc = 3 \). Thus, all the \( kc \)-combinations from \( B_{E_i} \) for entity \( E_i \) need to be generated.

The reduce function of the procedure further determines those entities having at least \( kc \) common keys to form a potential list for the matching step. The reducer would receive a \( kc \)-combination as the key and the list of entity ids having that combination. If the list contains only one entity for a certain key combination, the entity is eliminated from the matching step obviously. Therefore, we count the number of entities in the \( entity\_list \) \( (entity\_list\_count) \) while adding entities of the same \( K_0 \) to the associated \( entity\_list \). Only \( entity\_list \) having two or more entities will be sent to the load-balanced matching.

An example showing the map/reduce functions with nine entities, three mappers, two reducers, and \( kc = 2 \) is displayed in Fig 3. Mapper1 generates \(<(a, c), 1> \), \(<(a, d), 1> \), \<(c, d), 1> \), and so on; Mapper2 generates \(<(a, c), 4> \), \(<(c, e), 4> \) and so forth. A simple hash partitioning is adopted for the reducers. Assume Reducer1 accepts keys \( (a, c), (a, e) \), and so on. Hence, \( entity\_list \_{(1, 2, 4, 8)} \) of the same key \( (a, c) \) will be sent to the next procedure, load-balanced matching.

### 4.3 Load-balanced Matching

The matching step has to compute similarities for the pairs of all the 2-combinations of each entity list generated by the blocking step. A common implementation of the matching procedure uses the default hash partitioning to pass the intermediate keys to reducers. Consequently, the reducer receiving long entity-lists suffer from the vast amount of similarity computations. A skewed key distribution also leads to imbalanced workloads for reducers. Hence, the matching step cannot finish until the reducer of the heaviest workload completes. Take the entity_lists in Fig. 3 for example, reducer1 produces 7 entity pairs (including \( (1,2), (1,4), (1,8), (2,4), (2,8), (4,8), \) and \( (7,8) \)) but reducer2 produces only 2 pairs for similarity computations. The blocking-based ER can be completed earlier without unnecessary waiting if the workloads are balanced among reducers in the matching step.

Figure 4 shows the map/reduce functions in the proposed load-balanced matching procedure. Note that the \( partition \) function, which replaces the default hash partitioning, is particularly designed to evenly distribute the total number of comparisons to the total number of reducers (i.e. \( numReduceTasks \)). In order to average the workloads, we use a partition number \( partno \) to count and mark each input record (entity pair). The total number of entity pairs for matching is then obtained by the \( partno \). Note that we prune the duplicate pair \( (2, 4) \) from keys \( (a, e) \), and \( (c, e) \) here. The mapper generates \((partno, entity\_pair)\) as the (key, value) for the reducers but applying the designated partitioning for load balancing. Entity pairs are sent to all the receivers by round robin, as shown in the \( partition \) function in Fig. 4. The reducers then outputs the similar pairs if their similarity value is no less than the similarity threshold.

Figure 5 shows an example of load-balanced matching with two mappers and two reducers. The matching receives entity lists in Fig. 3. A mapper generates the \( <partno, entity\_pair> \) by enumerating all the 2-combinations of an entity list with associated \( partno \), which value is increased by 1 for each combination. The entity list \( (1, 2, 4, 8) \) will produce \( (1, 2), (1, 4), (1, 8), (2, 4), (2, 8), \) and \( (4, 8) \) entity pairs. Thus, \( <1, (1, 2)>, <2, (1, 4)>, <3, (1, 8)>, <4, (2, 4)>, <5, (2, 8)>, \) and \( <6, (4, 8) > \) are generated by mapper1; \( <7, (7, 8)>, <8, (5, 9)>, \) and \(<9, (6, 7) > \) are generated by mapper2. The partitioning determines which reducer to receive the \( <partno, entity\_pair> \) using the designed partition function. Because the total number of reducers is two, \( <partno, entity\_pair> \) having odd \( partno \) is processed by reducer1, and that having even \( partno \) is processed by reducer2. Assume that the similarity value of \( \text{sim}(E_1, E_2), \text{sim}(E_2, E_3), \text{sim}(E_3, E_4), \) and \( \text{sim}(E_4, E_5) \) is less than the similarity threshold. The matched result contains similar pairs \( (1, 2), (7, 8), (1, 4), (2, 4), \) and \( (5, 9) \). If we configure the MapReduce job with 3 reducers, the 9 computations will be evenly distributed to the 3 reducers.

### Functions for Load-balanced Matching

**Fig. 4: Functions for Load-balanced Matching**

**Fig. 5: An Example for Load-balanced Matching**
5 Experimental Results

Extensive experiments were conducted to assess the performance of the proposed algorithm. Our experiments were performed in a 30-nodes cluster which contains three types of nodes, as shown in Table 1. All nodes run in Ubuntu 10.10, Hadoop 0.20.205.0, and Java 1.6.0. Real dataset CiteSeerX (CiteSeerX 2012) was used in the experiments. CiteSeerX contains about 1.4 million publication records of total size 1.8 GB. Each publication record includes a record id, title, keywords, abstract, and URL information. The record id and its title were extracted as keywords, which serve as the blocking keys. A record has 4.9 keys in average; the maximum number of keys is 20. Very few records contain only one key and they were skipped for the ER process since the minimum number of common keys \( kc \) is greater than one in our problem definition.

### Table 1: Experimental Environments

<table>
<thead>
<tr>
<th>Config.</th>
<th>9 nodes</th>
<th>9 nodes</th>
<th>12 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i5</td>
<td>Intel Core i5</td>
<td>Intel E7600</td>
</tr>
<tr>
<td></td>
<td>3.24 GHz * 4</td>
<td>3.24 GHz * 2</td>
<td>3.06 GHz * 2</td>
</tr>
<tr>
<td>Memory</td>
<td>3.4 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hard disk</td>
<td>SATA 500G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 shows that 94.3k combinations are generated and 53.8k resulting pairs need to be matched for \( kc = 2 \). There are 216.2k combinations are generated and only 36.5k resulting pairs need to be matched when \( kc \) is increased to 3. When \( kc \) is increased to 4, the number of combinations is the largest of 411.9k but only 19.5k pairs need further matching. When \( kc \) is increased to 5, the number of pairs required matching decreased to 8.4k.

### Table 2: Combinations and Results vs. Blocking Keys

<table>
<thead>
<tr>
<th>Blocking keys</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>#combinations</td>
<td>94,365</td>
<td>216,230</td>
<td>411,962</td>
<td>209,366</td>
</tr>
<tr>
<td>#pairs</td>
<td>53,840</td>
<td>36,549</td>
<td>19,593</td>
<td>8,433</td>
</tr>
</tbody>
</table>

Figure 6 shows the total execution time with respect to the number of common blocking keys \( kc \). The number of mappers \( |M| \) was 20, and that of reducers \( |R| \) was 13. The total execution time required for both combination-based blocking (legend \( blocking \)) and load-balanced matching (legend \( load-balanced \)) are displayed. Both times increases as \( kc \) increases. As mentioned above, the number of resulting pairs decreases rapidly as \( kc \) increases so that the real matching time decreases sharply in fact.

Table 3 indicates that the maximum load of the reducers decreases from 134MB to 56MB for \( kc = 5 \) after applying load-balanced matching. The data size to be handled before applying the proposed load-balanced matching is 2.3 times of that after applying the matching. In fact, the output data size after the combination-based blocking is 74MB for \( kc = 2 \), and up to 728MB for \( kc = 5 \). Without proper balancing of the reducers, many reducers would have to wait for the maximum-loaded reducer to complete its job. Take \( kc = 5 \) for example, before applying the balanced-matching, some reducers worked with nearly zero-sized data combinations and some were heavily loaded with 134MB data. The overall process cannot finish until this heavy-loaded reducer completes. Figure 7 depicts that the workloads can be effectively balanced so that the maximum load of the reducers can be decreased for all the settings of \( kc \).
Table 3: Changes of Maximum Load of the Reducers

Next, the number of reducers was varied to evaluate the effects on total execution time. Figure 8 shows that the total execution time decreases as the number of reducers increases. As expected, more reducers may speed up the process. Note that the experiment uses an extreme large dataset by replicating CiteSeerX five times so that the differences of execution times can be exploited. When the size of input data is not very big, a MapReduce configuration with many reducers actually may cause extra-communications among the reducers. Thus, the dataset was enlarged to highlight the effects of the number of reducers.

![Max. Reducer Load](image)

**Fig. 7: Effects of Load-balanced Matching**

![Varying Number of Reducers](image)

**Fig. 8: Performance on Varying the Number of Reducers**

Fig. 9: Scalability of the Proposed Algorithm

The experimental result of evaluating the scalability of the proposed algorithm is shown in Fig. 9. The dataset CiteSeerX is replicated for 5 and 10 times, respectively. In Fig. 9, the total execution time increases linearly as the dataset size increases.

6 Conclusion

In this paper, we propose an algorithm to solve the entity resolution problem for big data analytics, using the MapReduce framework. The characteristic of multiple keys in entities is presented and utilized for effective key blocking (entity distribution) to improve the entity resolution process. The proposed algorithm features in the combination-based blocking and load-balanced matching. The combination-based blocking produces potential key combinations and distributes entity lists having common keys. The load-balanced matching generates entity-pairs combinations and balances the workloads of similarity computations among all the reducers in the MapReduce configuration. The experimental results show that the proposed algorithm may efficiently solve the entity resolution problem. Future works may extend this research to the entity resolution problem for the RS-join, such as the join between DBLP and the CiteSeerX.

7 Acknowledgement

The authors are grateful for the comments of the reviewers for improving the quality of the paper. This study is supported partly by the National Science Council, Republic of China, under grant no. NSC102-2410-H-324-006.

8 References


