Front. Comput. Sci., 2017, 11(5): 895–911
DOI 10.1007/s11704-016-5346-4

RESEARCH ARTICLE

MapReduce-based entity matching with multiple blocking functions

Cheqing JIN (✉), Jie CHEN, Huiping LIU

Institute for Data Science and Engineering, School of Computer Science and Software Engineering,
East China Normal University, Shanghai 200062, China

Abstract Entity matching that aims at finding some records belonging to the same real-world objects has been studied for decades. In order to avoid verifying every pair of records in a massive data set, a common method, known as the blocking-based method, tends to select a small proportion of record pairs for verification with a far lower cost than $O(n^2)$, where n is the size of the data set. Furthermore, executing multiple blocking functions independently is critical since much more matching records can be found in this way, so that the quality of the query result can be improved significantly.

It is popular to use the MapReduce (MR) framework to improve the performance and the scalability of some complicated queries by running a lot of map (/reduce) tasks in parallel. However, entity matching upon the MapReduce framework is non-trivial due to two inevitable challenges: load balancing and pair deduplication. In this paper, we propose a novel solution, called MrEm, to handle these challenges with the support of multiple blocking functions. Although the existing work can deal with load balancing and pair deduplication respectively, it still cannot deal with both challenges at the same time. Theoretical analysis and experimental results upon real and synthetic data sets illustrate the high effectiveness and efficiency of our proposed solutions.

Keywords entity matching, MapReduce, load balancing, pair deduplication

Received August 14, 2015; accepted December 15, 2015

E-mail: cqjin@sei.ecnu.edu.cn

1 Introduction

Entity Matching is critical in both data cleaning and data integration, which is to find matching records belonging to the same real-world object [1–5]. For example, papers or books referring to the same citations may exist in several different bibliographic sources; fingerprints belonging to the same persons may be collected from different places. Entity matching should be applied when integrating such data sources into one clean source. In the literature on the subject, it has also been referred to as object matching, record linkage, reference reconciliation and so on.

It is challenging to execute entity matching upon a massive data set efficiently since the number of record pairs is huge, i.e., $O(n^2)$ complexity where n is the number of records in the data set. The *blocking-based method* is a typical solution [6–9]. With the help of a specific blocking function, it selects a small proportion of record pairs for further comparison. The rest record pairs are ignored directly, so it greatly saves the computational cost.

Typical blocking methods include standard blocking [10], sorted neighborhood method [11], locality sensitive hashing (LSH) [12,13] and so on. Each specific blocking function can generate a key for each record. Records that share the same key are dispatched to the same block and any pair belonging to the block is treated as a candidate pair, which is to be verified later. In general, the keys for each record in the data set can be generated by scanning the data set only once. It is worth noting that the cost of key generation is cheap in comparison with the exact verification cost.

Example 1 Table 1 illustrates a small data set containing nine names, where the first six names refer to Martin Luther King, a leader in the African-American Civil Rights Movement, and the last three names refer to Mark Twain, an American writer and humorist. Hence, there are in total 18 $\left(=\binom{6}{2}+\binom{3}{2}\right)$ matching record pairs. The First/Last K Letters (FirstKL/LastKL) are two specific strategies of the standard blocking techniques, where FirstKL (/LastKL) extracts the first (/last) K letters of a record as the key. Using the FirstKL ($K = 2$) method will divide all records into two blocks according to the distinct keys: "Ma" and "M.". Similarly, using the LastKL ($K = 2$) method will result in three distinct keys: "ng", "r.", and "in".

Table 1 A small data set

	rID	Content	eID	Key ($K = 2$)	
				FirstKL	LastKL
t_1	1	Martin Luther King	α	Ma	ng
t_2	2	Martin Luther King, Jr.	α	Ma	r.
t_3	3	Marting Luther King	α	Ma	ng
t_4	4	Martin King	α	Ma	ng
t_5	5	Martin L. King	α	Ma	ng
t_6	6	M. L. King	α	M.	ng
t_7	7	M. Twaing	β	M.	ng
t_8	8	Mark Twain	β	Ma	in
t_9	9	M. Twain	β	M.	in

Note: α = Martin Luther King, β = Mark Twain

Figures 1(a) and 1(b) illustrate the candidate pairs generated by each blocking function. The candidate pairs achieved by each function are represented by the solid lines (for the matching pairs) and dash lines (for the non-matching pairs). For example, in Fig. 1(a), (t_1, t_2) is a matching candidate pair, (t_1, t_8) is a non-matching candidate pair, while (t_1, t_7) is not a candidate pair. We can observe that each blocking function achieves 11 $\left(=\binom{5}{2}+\binom{2}{2}\right)$ matching pairs.

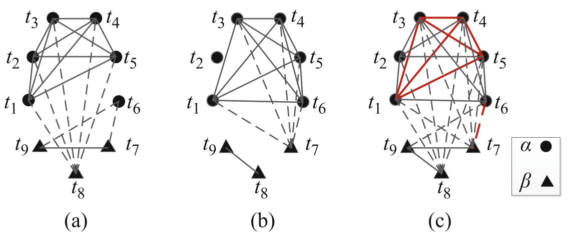


Fig. 1 The candidate pairs. (a) FirstKL; (b) LastKL; (c) FirstKL+LastKL

1.1 Using one or multiple blocking functions?

As mentioned above, the candidate pairs generated by a special blocking function cannot cover all matching pairs. The blocking function may miss some matching pairs, while also contain some non-matching pairs. Moreover, each blocking function also has different effects on a data set.

An interesting extensive issue for qualified result is to select a suitable blocking function that is capable of finding more matching pairs, executing more efficiently, or both. In general, such a blocking function can be selected by the computation upon a small training set. However, the quality may still be restrained even using the best blocking function. The reports upon real data sets (to be introduced later in Section 5) show that more than 10% matching pairs will be missed no matter which of the five blocking functions is used. Similar phenomenon is also reported in a survey paper [7].

An alternative method for qualified result is to use multiple blocking functions simultaneously instead of a single blocking function. At first, it generates the candidate pairs by using each blocking function, denoted as S_i. Subsequently, it generates $\bigcup_i S_i$ as the final candidate pairs. Finally, each pair in $\bigcup_i S_i$ is verified. The effects of using multiple blocking functions are two folds. First, the execution cost increases since more candidate pairs need to be verified. Second, the quality of the query result is further improved since more matching pairs are detected. In general, we prefer to achieve better qualities by paying a little more overhead [7,14,15].

For example, Fig. 1(c) illustrates all candidate pairs by using two blocking functions at the same time. The red lines refer to the candidate pairs generated by both blocking functions. Now, 15 matching pairs are found in the candidate pairs, significantly better than using single blocking function.

1.2 Cloud computing based entity matching and the challenges

Using the cloud computing paradigm to implement some data-intensive tasks efficiently is quite popular in recent years, such as entity matching. In such a situation, a number of tasks are running in parallel over a cluster that contains many computers. As one of the most widely used programming models in the cloud computing paradigm, the MapReduce (MR) model enables multiple map (/reduce) subtasks running in parallel to boost query processing. The motivation of this paper is to devise novel MR-based solutions to process entity matching.

Most recently, Kolb et al. first applied MapReduce framework to entity matching with single blocking function, with a special interest in load balancing because of the skewed block distribution [16]. They proposed two MR-based solutions. Their first solution, called *BlockSplit*, splits large blocks into small ones (e.g., sub-blocks) according to the input data partitions, and each sub-block will be handled by one task. However, the workloads may still be unbalanced due to varying sizes of sub-blocks. Their second solution, called *PairRange*, evenly dispatches record pairs to each reducer by enumerating all pairs based on statistical information called *block distribution matrix*, BDM. The BDM in their work in fact is a 2-D matrix that specifies the number of records of each block across each input partition.

Besides load balancing, when employing multiple blocking functions to achieve high quality, another important observation is that many same pairs will exist in multiple blocks. In Fig. 1(c), six overlapped candidate pairs are generated by two blocking functions (described by red lines). We call such pairs as duplicated pairs. In a centralized system, each pair needs to be verified once after generating the union of all candidate sets. However, it is challenging to verify distinct pairs only once in a distributed system since the duplicates spread across all running nodes. As we know, verifying the pairs is expensive, which may take several hours or even days for large data sets in real world [17]. Thus if we conduct repeated verification for these duplicates in different nodes simultaneously, it may lead to a huge waste. Kolb et al. also proposed a straightforward technique to achieve pair deduplication [18]. Given a list of blocking functions, each mapper not only emits the blocking key under current blocking function but also appends all blocking keys generated by previous blocking functions. Thus each reducer can avoid evaluating current record pair if their previous blocking keys are overlapped.

Dedoop [19] is a prototype and it provides load balancing techniques mentioned in Ref. [16] and pair deduplication mechanism mentioned in Ref. [18]. However, in fact the problem of load balancing is still unsolved when we try to combine above solutions to achieve pair deduplication and load balancing at the same time. Although *PairRange* assumes the number of comparisons in each reducer is nearly equal, the solution of pair deduplication will damage this hypothesis. Some reducers will avoid comparisons because of the overlapped previous blocking keys, leading to the result that workloads among reducers are still unbalanced. Hence, it is critical to devise novel solutions to address both challenges.

1.3 Our framework

In this paper, we propose and evaluate five MR-based solutions to handle entity matching, with special considerations on pair deduplication and load balancing. In order to achieve pair deduplication, we attempt to push the duplicated pairs (generated by different blocking functions) to one working node so that each pair can only be verified once. As for unbalanced workloads caused by skewed data distribution, we split *heavy* tasks into small subtasks. In summary, we make the following contributions:

- We restudy the issue of entity matching in a distributed environment by addressing two big challenges: load balancing and pair deduplication. One significant limit of the existing work is that these challenges cannot be handled at the same time.
- We present a novel generalized framework (MrEm) to address both challenges. Our framework consists of four stages, called *block building*, *interface implementation*, *pair verification*, and *pair cleaning*. Stages 1, 3, and 4 construct the mainstream workflow to process data, and Stage 2 is responsible for dealing with two challenges: load balancing and pair deduplication. We propose two concrete implementations to support the framework: MrEm-SI and MrEm-NSI.
- We present three basic solutions to deal with this issue, including Naive, PairUnit, and Dedoop+. Although much simpler, their performance is lower than MrEm, which emphasizes that this issue cannot be handled by some straightforward attempts due to the big challenges. Moreover, listing such methods also helps to illustrate the MrEm method.
- Finally, we conduct a series of experiments on real and synthetic data sets to illustrate high effectiveness, efficiency and scalability of the proposed solutions. We also discuss several critical parameters.

The rest of our paper is organized as follows. We review the preliminary knowledge in Section 2. In the next section, we present three basic solutions to highlight the difficulty in dealing with the two challenges, including Naive, PairUnit, and Dedoop+. In Section 4, we propose a novel general framework (MrEm) to overcome aforementioned challenges. Section 5 reports experimental results upon real and synthetic data sets. We review related work in Section 6. Finally, we conclude this paper briefly in the last section.

2 Preliminaries

In this section, we introduce the MapReduce framework [20] and some concepts. The MapReduce framework has been widely adopted for data-intensive parallel computation on shared-nothing clusters. Hadoop is a popular implementation of this paradigm [21]. The MapReduce framework can be simplified as the following two functions:

map: $(\mathsf{key}_{\mathsf{in}}, \mathsf{value}_{\mathsf{in}}) \rightarrow \mathsf{list}(\mathsf{key}_{\mathsf{tmp}}, \mathsf{value}_{\mathsf{tmp}})$;

reduce: $(\mathsf{key}_{\mathsf{tmp}}, \mathsf{list}(\mathsf{value}_{\mathsf{tmp}})) \rightarrow \mathsf{list}(\mathsf{key}_{\mathsf{out}}, \mathsf{value}_{\mathsf{out}})$;

We describe the data flow in MapReduce framework below. Data is stored on partitions of a distributed file system (DFS). Each mapper reads a lot of key-value pairs in the form of $(\mathsf{key}_{\mathsf{in}}, \mathsf{value}_{\mathsf{in}})$ from DFS and processes them separately. Subsequently, they will emit $\mathsf{list}(\mathsf{key}_{\mathsf{tmp}}, \mathsf{value}_{\mathsf{tmp}})$ as output. Then the jobs automatically sort the output according to $\mathsf{key}_{\mathsf{tmp}}$ and copy it to different reducers. Once reducers get the intermediate results, they will merge the records with the same $\mathsf{key}_{\mathsf{tmp}}$ and take $(\mathsf{key}_{\mathsf{tmp}}, \mathsf{list}(\mathsf{value}_{\mathsf{tmp}}))$ as input. Finally, they emit $\mathsf{list}(\mathsf{key}_{\mathsf{out}}, \mathsf{value}_{\mathsf{out}})$ to DFS as the final results.

Concept clarification For the ease of reading, We clarify some concepts at first. The *matching pair* describes a pair of records that refer to the same entity, i.e., (t_1, t_2) in Table 1 is a matching pair since both of them refer to the entity α. The *duplicated pair* describes a record pair that exists in multiple blocks generated by different blocking functions. For example, (t_1, t_3) is a duplicated pair when using FirstKL and LastKL. Here, (t_1, t_3) is also a matching pair, while (t_6, t_7) is a non-matching pair.

We use *pair deduplication* operator to avoid repeated verification for the duplicated pairs. In other words, it dispatches all of the duplicated pairs to one working node, so that a duplicated pair only needs to be verified once. Let's take (t_1, t_3) as an example. If it is dispatched to one working node, we can label this pair at the first verification, so that no further verification for this pair will be conducted again. Otherwise, such pair will be verified at different working nodes more than once.

One of the main goals of this paper is to implement *pair deduplication* as much as possible for efficiency.

3 Three basic solutions

In this section, we introduce three straightforward MR-based solutions for entity matching, including Naive, PairUnit, and Dedoop+. Although their performance is lower than MrEm, the novel method that will be introduced in the next section, listing such methods helps to illustrate the MrEm method.

3.1 Naive solution

The body of our first solution, Naive, contains two stages, namely *entity matching* and *pair cleaning*. The first stage generates and verifies the candidate pairs, and then emits the matching ones. Since the same pairs may be outputted several times after Stage 1, we also devise the second stage to eliminate duplicates to meet some specific requirements.

At Stage 1, it first generates t blocking keys for an incoming record r, denoted as $f_i \cdot f_i(r)$ in the *map* phase $(1 \leqslant i \leqslant t)$, where f_i denotes the i-th blocking function. The symbol "$\cdot$" denotes the concatenation operator between two strings. In the *reduce* phase, each reducer will receive the merged input data like $(key, \{v_1, v_2, \ldots, v_m\})$, where key comes from the blocking key mentioned above (like $f_i \cdot f_i(r)$), and $\{v_1, v_2, \ldots, v_m\}$ represents a number of input records sharing that blocking key. It is worth noting that a block with m records will result in $\binom{m}{2}$ candidate record pairs. Subsequently, each reducer will call specific similarity function to verify each candidate pair, and then emit the pair if matched.

At Stage 2, each mapper gets the output of Stage 1 as input. For an arbitrary record pair (t_i, t_j), it uses $i \cdot j$ as the key, and emits $(i \cdot j,\ (t_i, t_j))$[1]. In this way, any identical record pairs outputted by different mapper will be assigned with identical keys. Subsequently, all identical record pairs will be dispatched to the same reducer, where only one copy is emitted. This stage is a lightweight job.

Example 2 Figure 2 demonstrates a running example upon the data set in Table 1. We assume the original data set is split into two data partitions, and two mappers are used to handle it. Each mapper assigns two keys for every record by using FirstKL and LastKL. For example, the keys of record t_1 are "F · Ma" and "L · ng", where the symbols "F" and "L" denote the blocking functions: **F**irstKL and **L**astKL respectively. After shuffling, the records sharing the same key will go to one reducer. For example, six records $\{t_1, t_2, t_3, t_4, t_5, t_8\}$ are merged together since they share the same key "F · Ma". Subsequently, 15 $(= \binom{6}{2})$ candidate pairs are generated from this block for verification. Finally, the reducer only outputs ten pairs since there are only 10 matching pairs among 15 pairs after verification. After Stage 1, there exist some duplicated pairs in two reducers, such as (t_1, t_3). Hence, we can invoke Stage 2 to clean the results.

Although easy to implement, Naive approach still contains two obvious drawbacks: duplicates verification and unbal-

[1] If $i > j$, we use $j \cdot i$ as the key instead

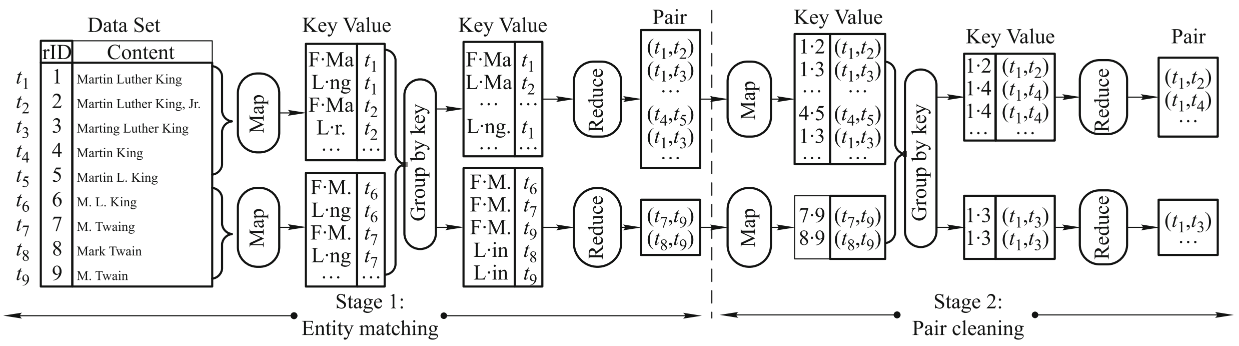


Fig. 2 A running example of Naive

anced workloads. First, some pairs will be verified more than once since they may be in different blocks, such as (t_1, t_3) and (t_1, t_4) in Fig. 2. Second, the size of each block may vary significantly, and it takes a long time to process a large block. Hence, it will be encountered with unbalanced workloads. For example, reducer 1 (the upper reducer at Stage 1, Fig. 2) is much heavier than reducer 2 (the lower reducer at Stage 1) because two large blocks ($\{t_1, t_2, t_3, t_4, t_5, t_8\}$ and $\{t_1, t_3, t_4, t_5, t_6, t_7\}$) are dispatched to reducer 1 in this case.

3.2 PairUnit solution

There exists a simple 2-stage solution to deal with the above two challenges. Our intuitive idea is that by enumerating all pairs within the same blocks in the *map* phase, duplicated pairs can be detected in the *reduce* phase. Meantime, the problem of load balancing is also resolved because all pairs are evenly dispatched to the reducers. At Stage 1, it mainly generates the blocks according to the blocking functions without generating or verifying the candidate pairs. At Stage 2, it begins to generate the candidate pairs one by one within the same block in the map phase. Then, each mapper emits the key-value pairs, where the key is the composition of two record IDs and the value refers to the pair of records. It also means that all the duplicated pairs will be merged into a single key-value pair due to the same key. Hence, any duplicated pair only needs to be verified once, and this method can balance the workloads because each basic unit only contains one pair. We call this solution PairUnit.

Example 3 Figure 3 illustrates a running example of PairUnit. After Stage 1, four blocks (b_1, b_2, b_3 and b_4) have been built by using two blocking functions. At Stage 2, we generate all candidate pairs in the map phase, and use the composition of two record IDs as the key. For example, we assign the key of the pair (t_1, t_3) as "1·3", where the symbol "·" still denotes the concatenation operator. Even though the records like t_1 and t_3 belong to more than one blocks, their pairs only require to be verified once in the reducer side through this way.

It seems that this solution can easily solve the challenges we have outlined, but it has its own shortcomings. First of all, the amount of data to be shuffled from mappers to reducers is huge. If a block contains m records, each record should be replicated $(m - 1)$ times since we can generate $(m - 1)$ pairs by integrating this record with the others. For a large m, the cost on shuffling is intolerable. Although the I/O cost can be lowered by using the *combine* function that can merge identical pairs in same mappers, the amount of data is still too big to transfer. Second, the *map* function at Stage 2 will expand to generate all candidate pairs. If a large block contains quite

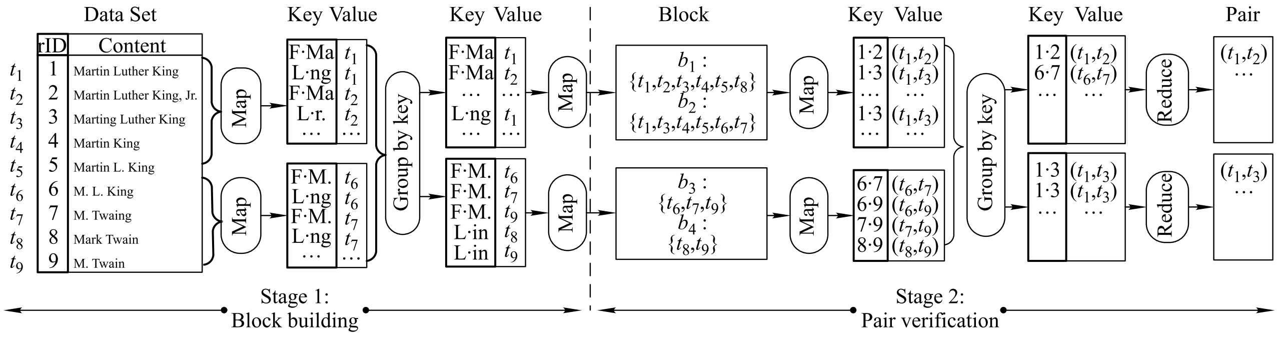


Fig. 3 A running example of PairUnit

a lot of records, the cost of pair generation still remains as a big burden, which leads to unbalanced workloads among mappers. Eventually, it will make all reducers wait until all mappers complete. Hence, PairUnit can only be applied to the applications with small data amount. When the data amount becomes large, this solution is inefficient.

3.3 A variant of Dedoop solution: Dedoop+

The work in Ref. [16] by Kolb et al. only uses one blocking function for entity matching, with a special design to deal with load balancing. Subsequently, the same authors study how to deal with entity matching by using multiple blocking functions, with a special design on pair deduplication [18]. Although these two techniques have contributed a lot in the Dedoop system, how to combine them together have not been reported yet, not to mention the performance [19]. Hence, we try to propose a simple method called Dedoop+ by integrating these two techniques together.

Figure 4 illustrates this solution by using a running example, which contains two stages. At Stage 1, four blocks are generated: b_1, b_2, b_3, and b_4[2)] by using two afore-mentioned blocking functions. The output results of mappers also contain additional information about the keys generated by previous blocking functions. For example, "F · M." and "L · ng" are generated successively for record t_6. The additional information for "F · Ma" is empty since it is the first blocking key. Contrarily, the additional information for "L · ng" is "F · Ma", which has already been generated. Meantime, it also outputs a BDM file to collect statistical information about all blocks.

At Stage 2, the BDM file is used as the distributed cache file to deal with load balancing, which will be loaded by *map* (*/reduce*) tasks initially. In this example, there are 34 candidate pairs in total, so that each reducer needs to compare 17 record pairs. In the map phase, each mapper will transfer record pairs in current block to the corresponding reducers according to the BDM file. In the reduce phase, each reducer will combine the records in the same block. Before executing verifications, the additional information is used to determine whether the current record pair needs to be compared or not. For example, since the record pair (t_1,t_3) in the block b_3 has the same previous blocking key "F · Ma", this pair can be ignored directly. This pair is verified by the upper reducer at Stage 2 in Fig. 4.

Although the work in Ref. [16] deals with load balancing well, its overall performance decreases if we combine it with the work in Ref. [18]. The reason behind is that some reduces will skip the current verification due to the overlapped additional information, which makes the load unbalanced. When more blocking functions are employed, more duplicated pairs in blocks will not be verified, leaving the workloads more unbalanced. For example, the load in Fig. 4 is very biased, i.e., Reducer 1 compares 17 pairs while Reducer 2 only compares ten pairs.

4 A novel solution: MrEm

As mentioned above, load balancing and pair deduplication are two main challenges when using multiple blocking functions, while three basic solutions cannot handle them well. Naive simply sends all records in each block to one reducer for pair verification, so that the problems of load balancing and pair deduplication still remain. It seems that PairUnit can deal with both challenges perfectly. Unfortunately, the actual performance is poor in most occasions since much additional overhead will be created, such as generating and shuffling all record pairs. Dedoop+ also leaves the problem of load

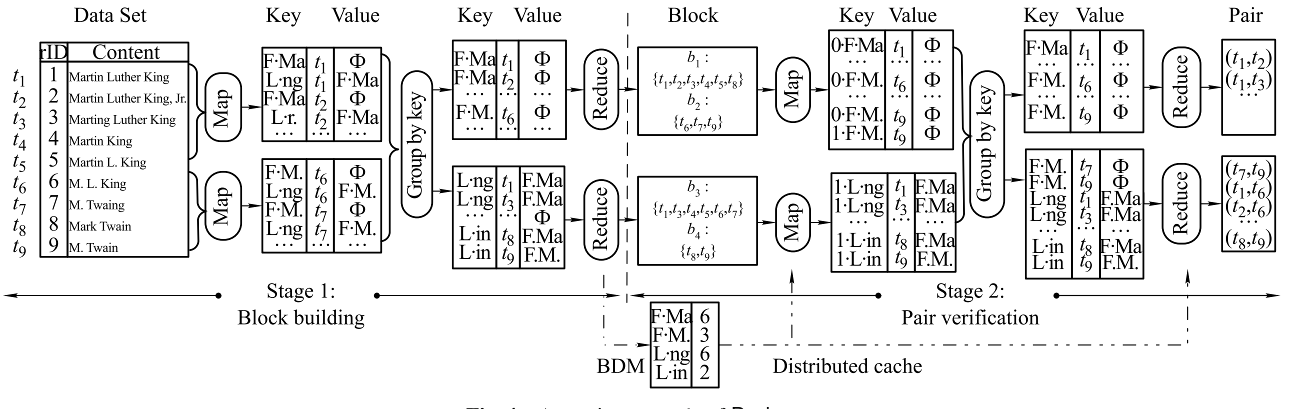


Fig. 4 A running example of Dedoop+

2) For the ease of simplicity, we exchange b_2 with b_3

Table 2 Comparisons among all solutions

Solution	Load balancing	Pair deduplication	Block size	Description
Naive	No	No	Any	Simple, but cannot handle any challenge
PairUnit	Yes	Yes	Tiny	Only suitable for small data amount
Dedoop+	No	Yes	Normal	The problem of load balancing is still unsolved
MrEm	Yes	Yes	Normal	Detect duplicates and split large blocks together

balancing unsolved when combining two techniques [16,18] together. Table 2 summarizes the differences of all solutions. The 2nd and 3rd columns describe the support of load balancing and pair deduplication. The fourth column, *Block Size*, refers to the size of blocks generated during execution.

Consequently, it is necessary to devise a novel solution to deal with both challenges together. Next, we first introduce the general framework in Section 4.1, and then describe the critical stage in Section 4.2. Finally, we discuss the connection between basic solutions and our novel solution.

4.1 General MR-based framework

We propose a general framework for entity matching, called MR-based entity matching(MrEm), with two primary goals: 1) dividing a large block into some small sub-blocks, and 2) dispatching similar blocks to one reducer to detect duplicates. Here, similar blocks means such blocks are sharing a large proportion of common records. In summary, we need to implement the interface em.

```
interface em {
    int MAXBS;
    splitBlock(largeblock);
    assignKey(block);
}
```

In order to achieve load balancing, large blocks must be split into some sub-blocks in advance. The parameter MAXBS is used to control the maximal size of each block. We will discuss how to set the value in Section 4.2.1. The splitBlock function is responsible for splitting large blocks. The assignKey function assigns a proper key to each block, with the goal of assigning the identical key to similar blocks so that they can be merged into one key-value pair. Consequently, in single working node, we can detect the duplicated pairs by maintaining a hash set or Bloom Filter [22] in memory to record the already compared pairs.

The distributed cache mechanism provided by Hadoop [21] is critical to implement such an interface. It is a facility to cache files in all working nodes. However, since it will bring in data communication, the file size must not be large. Figure 5 presents our new framework with four stages.

- Stage 1 (Block building) At this stage, the input data is divided into a number of blocks by using *t* blocking functions. All records in one block have the same blocking key. Moreover, we sort the records within the same block according to their IDs, which is essential for the following stages. Note that this stage is same as Stage 1 in PairUnit.
- Stage 2 (Interface implementation) At this stage, some statistics are gathered to implement the interface em, which will be used in the next stage. We illustrate two concrete ways to implement this interface in this paper. Please refer to Sections 4.2.2 and 4.2.3 for details.
- Stage 3 (Pair verification) This stage contains the following steps. First, each large block will be divided into some sub-blocks by calling splitBlock function when its size exceeds MAXBS. Second, a proper key is assigned to each block (or sub-block) by calling assignKey function. Finally, we use a hash set to detect duplicates and verify distinct pairs by employing exact similarity functions in the *reduce* phase.
- Stage 4 (Pair cleaning) Since there still exist a few duplicated record pairs in different reducers after Stage 3, we use this stage to clean the results. This stage is same as that in Naive.

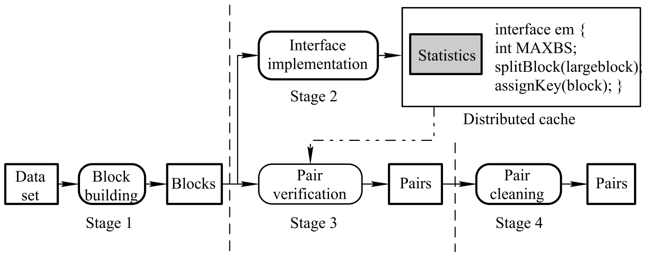


Fig. 5 Our MR-based framework: MrEm

Stages 1, 3, and 4 form a mainstream workflow, and Stage 2 is the basis of Stage 3.

4.2 Stage 2: Interface implementation

Stage 2 is undoubtedly the most critical in our framework. Here, we first introduce the principles to implement the interface em, following which we propose two concrete implementation plans.

4.2.1 Designing principles

The intrinsic principles to implement em are the answers of the following two questions.

Question 1: How to split a large block?

The unbalanced workloads are mainly caused by large blocks which need to be split into smaller ones. The first issue arisen is how to define a *large* block. Commonly, this concept is related to the settings of a cluster. Assuming there are R reducers and P record pairs for verifying, a large block may contain at least P/R pairs. In other words, a block containing n records is treated as *large* if $\binom{n}{2} \geqslant P/R$.

The second question is how to divide a large block. Given a block with size n ($n > \mathsf{MAXBS}$), we first generate c subgroups where $c = \lceil n/\mathsf{MAXBS} \rceil$. Then $c + \binom{c}{2}$ sub-blocks belonging to two categories are created. The first category, called *self-join* sub-block, contains all records in each subgroup. While the second one, called *cross-join* sub-block, contains all records from two sub-groups. The maximal number of pair verifications for any *self-join* or *cross-join* subblock is no greater than $\binom{\mathsf{MAXBS}}{2}$ or MAXBS^2 respectively, thus MAXBS should be less than $\sqrt{P/R}$. In real evaluations, the value of MAXBS can be set quite smaller. However, setting MAXBS too small will deteriorate the overall performance since the burst of too many small-sized sub-blocks causes additional overhead, such as generating and shuffling the record pairs.

Example 4 According to the FirstKL blocking function in Table 1, six tuples $\{t_1, t_2, t_3, t_4, t_5, t_8\}$ have the same key "Ma". Assuming $\mathsf{MAXBS} = 3$, this block is split into two self-join sub-blocks ($\{t_1, t_2, t_3\}$ and $\{t_4, t_5, t_8\}$) and one cross-join subblock: $\{t_1, t_2, t_3; t_4, t_5, t_8\}$ where ";" is a delimiter to distinguish two sub-groups.

Question 2: How to assign a same key to similar blocks?

Assigning a same key to similar blocks (containing many common records) so that they can be processed in one reducer is critical for pair deduplication. An equally important requirement is that such work should be completed in parallel.

At first, let's consider two arbitrary blocks, b' and b''. If they are totally different, $\frac{|b' \bigcup b''|}{|b'|+|b''|} = 1$, where $|b'|$ denotes the number of records in b'. Otherwise if b' is the same as b'', $\frac{|b' \bigcup b''|}{|b'|+|b''|} = \frac{1}{2}$. We also observe that for two similar blocks: b_1 and b_2 in Fig. 3, the value of $\frac{|b_1 \bigcup b_2|}{|b_1|+|b_2|}$ is $\frac{8}{12}$, close to $\frac{1}{2}$. Hence, a small value of such metric is desirable. We define this metric, *tightness*, formally below.

Definition 1 (Tightness) The tightness ξ for a set of blocks $(b_1, b_2, \ldots)$ is computed as: $\xi = |\bigcup_i b_i| / \sum |b_i|$.

Inspired by the above example, we describe our solution below. We use h hash functions, and each of them is capable of generating a random permutation for all records within the same blocks. The concatenation of top-k record IDs in each permutation is treated as a signature for that block. Note that each block b has h signatures, denoted as $sig_1(b), sig_2(b), \ldots, sig_h(b)$. Such work can be conducted through a MapReduce framework.

- Map phase For each arriving block b, emit h key-value pairs: $(sig_1(b), b), (sig_2(b), b), \ldots, (sig_h(b), b)$.
- Reduce phase For each arriving key-value pair $(sig, (b_1, b_2, \ldots))$, compute *tightness* ξ, and then emit (sig, ξ).

Similarly, in order to distinguish the signatures generated by different hash functions, we use $h_i \cdot sig_i(b)$ instead of $sig_i(b)$, where $1 \leqslant i \leqslant h$.

The output of the *reduce* phase will be used by all tasks in the next stage. Hence, we merge all the outputs of reducers into a single file[3)], named as TiTable, and use it as a distributed cache file. It is efficient to use TiTable as a distributed cache file since its size is quite small.

The final step is assigning a key for each block according to TiTable. First, a group of h hash functions are assigned to each mapper. Notice that these hash functions must be the same with the previous step in order to generate the same signatures. Subsequently, it looks up TiTable, and treats the signature with smallest *tightness* value as its final key.

4.2.2 Plan 1: Splitting-based implementation (SI)

The first plan is a splitting-based implementation. At Stage 2, it divides large blocks into small ones in the map phase. The way to split a large block has been described in Section 4.2.1. Then, it generates h signatures for each block by using h random hash functions. In the reduce phase, it creates TiTable.

The implementation of interface em to be used by Stage 3 is described below. The splitBlock function is also the same as that in Section 4.2.1, and the assignKey function looks up

3) We use a Hadoop instrument to execute this task: hadoop fs -getmerge <src> <localdst> [addnl]

TiTable to pick up a proper key for a given block.

Figure 6 shows the 2nd and 3rd stages by using SI. Assuming MAXBS = 3, at Stage 2, the *map* function firstly splits b_1 and b_2 into three sub-blocks respectively. Let H_1, H_2 denote two hash functions to generate two signatures ($k = 1$) for each block. For example, block b_3 has two signatures "$H_1 \cdot 6$" and "$H_2 \cdot 9$". Subsequently, we generate the TiTable in the reduce phase. The first column refers to the signature, and the second column refers to the *tightness* value. Note that at this stage, we only need to transfer the IDs of records from the mappers to the reducers since other attributes will not be used.

At Stage 3, each mapper first loads TiTable into memory when it starts. Subsequently, it reads blocks from Stage 1, splits large blocks into small sub-blocks, and then assigns a proper key to each block (or sub-block) according to TiTable. In the reduce phase, similar blocks sharing the same key are merged into one key-value pair, such as these four sub-blocks with key "$H_1 \cdot 1$". Finally, matching pairs are emitted by reducers. We can observe that five duplicated pairs are detected by reducers, including (t_1, t_3), (t_1, t_4), (t_1, t_5), (t_3, t_4), and (t_3, t_5).

We call the solution using SI implementation as MrEm-SI.

4.2.3 Plan 2: No-splitting-based implementation (NSI)

Different from SI, our second implementation (No-splitting-based implementation, NSI) does not split any large blocks into small sub-blocks in the *map* phase at Stage 2. Instead, it tries to construct the TiTable based on the original blocks in the *reduce* phase. Moreover, the TiTable in NSI is an improved version since a new column is added to record the information about splitting points. Hence, we call the new data structure TiTable$^+$. The splitting points are a series of points used to divide the whole block, which are constructed by visiting each signature one by one. For each signature, if the size of any block is no greater than MAXBS, the corresponding splitting information is empty. Otherwise, we attempt to find the largest block affiliated with this signature, and divide it into a series of sub-groups with a size no greater than MAXBS. The margin points of each sub-group are recorded as the splitting points. Similar to TiTable, TiTable$^+$ is also suitable to be used as a distributed cache file, since we only store record IDs of the splitting points.

Figure 7 shows the running example of Stage 2 using NSI. In the map phase, each mapper generates two keys for each original block using two hash functions: H_1 and H_2. Let $k = 1$. In the reduce phase, each reducer merges signatures and calculates the corresponding *tightness* value. Moreover, it also needs to find the splitting points for large blocks under current signature. For example, signature "$H_1 \cdot 1$" brings two large blocks b_1 and b_2 together. Since their length is the same, it randomly regards one block (here, b_1 is chosen) as the largest one and finds its splitting points: {3}. Thus the splitting points {3} are appended to the third column at the first line in TiTable$^+$.

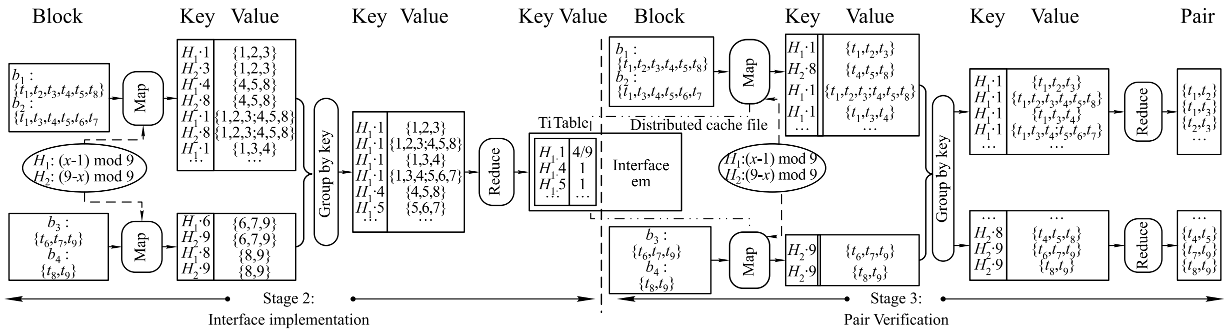


Fig. 6 A running example of MrEm-SI

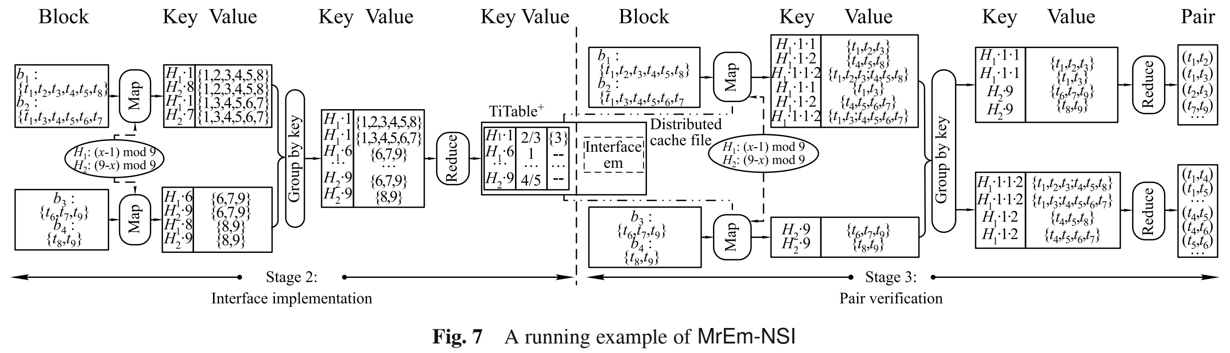


Fig. 7 A running example of MrEm-NSI

We next describe how to implement the interface em to be used at Stage 3. The splitBlock function will do nothing if the size of current block is no greater than MAXBS. Otherwise, it will look up the TiTable$^+$ to find an entry with the smallest *tightness* value, so that it is capable of splitting this block into some sub-blocks according to the splitting points.

The assignKey function here is different from that in SI implementation due to the requirement of assigning keys to some newly generated sub-blocks. We divide the large block into some sub-groups and mark each sub-group sequentially, i.e, $1, 2, \ldots$. The key for each sub-block is the concatenation of two parts: the signature in TiTable$^+$ and the tag of the corresponding sub-group(s).

Figure 7 also shows the execution of Stage 3 using NSI implementation. In the map phase, each mapper caches the TiTable$^+$ once setting up. Then it looks up TiTable$^+$ to find the proper key for each block. Meantime, it splits two large blocks b_1 and b_2 into three sub-blocks respectively according to the splitting points $\{3\}$. Finally, it assigns a new key to each sub-block. For example, b_2 is split into two sub-groups: $\{t_1, t_3\}$ and $\{t_4, t_5, t_6, t_7\}$. Thus the key of the *self-join* sub-block $\{t_1, t_3\}$ is "$H_1 \cdot 1 \cdot 1$", where the prefix "$H_1 \cdot 1$" denotes the signature in TiTable$^+$ and the last symbol "1" denotes it is in the first sub-group. The key of the *cross-join* sub-block $\{t_1, t_3; t_4, t_5, t_6, t_7\}$ is "$H_1 \cdot 1 \cdot 1 \cdot 2$", where "$H_1 \cdot 1$" still denotes the signature and the suffix "$1 \cdot 2$" denotes it is from sub-groups 1 and 2. In the reduce phase, reducers run the similar way without further explanations. In this manner, we can detect six duplicates, including (t_1, t_3), (t_1, t_4), (t_1, t_5), (t_3, t_4), (t_3, t_5), and (t_4, t_5).

We call the solution using NSI implementation as MrEm-NSI. Alignment is the main advantage of NSI over SI, since all large blocks with the same signature are split by a same series of splitting points. We take b_1 and b_2 in Fig. 7 as an example. When using SI and NSI implementation, we can detect five and six duplicated pairs respectively. However in NSI, the splitting results for b_1 and b_2 are $(\{t_1, t_2, t_3\}, \{t_4, t_5, t_8\})$ and $(\{t_1, t_3\}, \{t_4, t_5, t_6, t_7\})$ respectively. We can see that the splitting results are aligned better than SI.

4.3 Discussion

Our new solution has five characteristics: lightweight interface implementation, excellent scalability, load balancing, pair deduplication, and compatibility.

• Lightweight interface implementation   As mentioned above, Stages 1, 3, and 4 construct the mainstream workflow, and Stage 2 serves to implement the interface em. In general, we expect a lightweight implementation. We claim ours are lightweight implementations because of two reasons. First, it is efficient to generate signatures and calculate TiTable (or TiTable$^+$). Second, we only shuffle record IDs within blocks, instead of records, to reducers to minimize the data communication.

• Excellent scalability   Although MapReduce supports parallel mechanism by nature, some MR-based solutions cannot support this functionality well, since part of the work must be conducted by a single map (/reduce) task, such as global sorting in Ref. [23]. Contrarily, in our framework, multiple map (/reduce) tasks can run in parallel at any stage. Besides, the size of the distributed cache file (TiTable) is small, which can be easily loaded into memory.

• Load balancing and pair deduplication   Our solution can achieve good balancing since we evenly dispatch the split sub-blocks into each node by randomly routing the keys in MapReduce framework. Meanwhile, we attempt to dispatch similar blocks to the same working node to support pair deduplication. It is worth noting that the above two techniques can work together and make the number of compared candidate pairs in each node roughly same. This is the major difference between Dedoop(+) and ours.

• Compatibility   The framework of our solution contains three stages in the mainstream, and one stage to implement em. Three basic solutions, including Naive, PairUnit, and Dedoop+, are compatible to this framework. For example, PairUnit implements em below: splitBlock splits any block into smallest blocks and one block contains only single pair, and assignKey assigns the composite ID of records as the key.

5 Experimental results

In this section, we evaluate the effectiveness, efficiency, and scalability of our novel solutions, in comparison with three basic solutions. We run experiments on an 8-node HP ProLiant DL360 cluster. Each node has eight Intel Xeon E5606 2.13GHz quad-core processors, 32GB physical memory and 1TB disk. We install the RedHat Enterprise 6.0 operating system, Java 1.6 with a 64-bit server JVM and Hadoop 1.0.3 on each node. One node is assigned to run the Hadoop jobtracker (the MapReduce master node responsible for assigning tasks to different working nodes) and the namenode (the master node for HDFS). The rest seven nodes (datanodes) are used to store data and to run MapReduce jobs, resulting in 28 ($= 7 \times 4$) slots for mapper tasks and 56 ($= 7 \times 8$) slots for reducer tasks in total. Since we implement pair verification in the reduce phase (the most costly step), we allocate 40 slots for the reducers[4]. The allocation strategy will change at the scalability testing part (Section 3).

For all experiments, we adopt edit distance as the similarity function at *pair verification* stage and regard pair as matching if the similarity value is above 0.8. We use real and synthetic data sets, as introduced below.

• Real data set (CiteSeer) The CiteSeer data set[5] is a citation collection and has attributes such as *author, title, date, page, volume, publisher*, etc. We use *title* for verification since its length is longer than the rest attributes and has higher quality.

• Synthetic data set (Twitter) We generate a labelled synthetic data set based on a Twitter data set. For each tweet, we generate a number of matching records (the number is uniformly selected in [2,200]) by randomly changing some letters, and limit the error rate to no greater than 2%.

We use *recall*, *running time*, *speedup* and *scaleup* to measure the effectiveness, efficiency and scalability of entity matching in the distributed environment. *Recall* measures the proportion of the matching pairs detected by the blocking function(s). $Recall = \frac{bm}{am}$, where *bm* denotes the number of matching pairs sharing at least one block and *am* represents the number of all matching pairs. A greater *recall* value means higher effectiveness. *Running time* measures the efficiency of our solutions. *Speedup and scaleup* test the scalability of a parallel approach. *Speedup* describes how much a parallel approach is faster than a corresponding sequential approach, and *scaleup* refers to the ability of the parallelism to increase with both the system and the amount of required work.

5.1 Effectiveness testings

In general, using multiple blocking functions can achieve higher effectiveness. Here, we adopt five specific methods: First/Last K Letters (FirstKL/LastKL), First/Last M Tokens (FirstMT/LastMT) and min-Hash. We set $K = 5$ for FirstKL/LastKL, and $M = 1$ for FirstMT/LastMT. Min-Hash uses two parameters: r to control the number of hash functions in each key and q to shingle an attribute into q-grams. A min-Hash key is in fact the concatenation of r minimal hash values upon all q-grams obtained by using such r hash functions. By default, we set $r = 2$, and $q = 5$.

At first, we evaluate *recall* upon a randomly selected subset of CiteSeer (containing 100 000 records) and Twitter (containing 500 000 records) by using different blocking functions. Figure 8(a) reports the *recall* values under different blocking functions. We can observe that the *recall* value is below 0.9 for all situations. Moreover, no blocking function behaves well for both data sets, which highlights the importance of using multiple blocking functions for higher effectiveness.

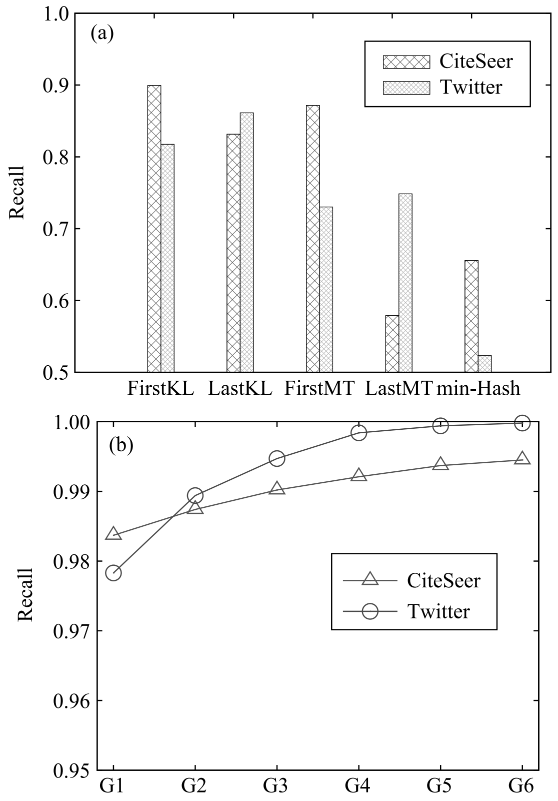


Fig. 8 Effectiveness of the blocking functions. (a) Single blocking function; (b) blocking function groups

Figure 8(b) shows the *recall* values under different groups, each containing more than one blocking function. Table 3 shows the detailed combination of each group. The number of blocking functions in each group rises from 2 to 10. For example, the fourth group, G4, contains six blocking functions: FirstKL, LastKL, and four independent min-Hash functions. The *recall* value continues to grow with the increment of the number of blocking functions.

5.2 Efficiency testings

Figure 9 shows the running time of Dedoop+, MrEm-SI, and MrEm-NSI under different blocking function groups. Remember that the number of blocking functions continues to

[4] It is fairly too expensive to build a large-scale cluster for an academic institute. In order to evaluate load balancing within the current cluster, we set the number of reduce slots bigger than that of map slots

[5] http://pike.psu.edu/download/edbt10/harra/

increase from G1 to G6. MrEm-NSI performs better than Dedoop+ upon both data sets due to the better support of both load balancing and pair deduplication. Moreover, the processing cost of Dedoop+ grows faster than MrEm-NSI with the increasing number of blocking functions.

Table 3 Six blocking function (BF) groups

Group name	Combinations of BF	Number of BF
G1	FirstKL, LastKL	2
G2	FirstKL, LastKL, min-Hash	3
G3	FirstKL, LastKL, min-Hash * 2	4
G4	FirstKL, LastKL, min-Hash * 4	6
G5	FirstKL, LastKL, min-Hash * 6	8
G6	FirstKL, LastKL, min-Hash * 8	10

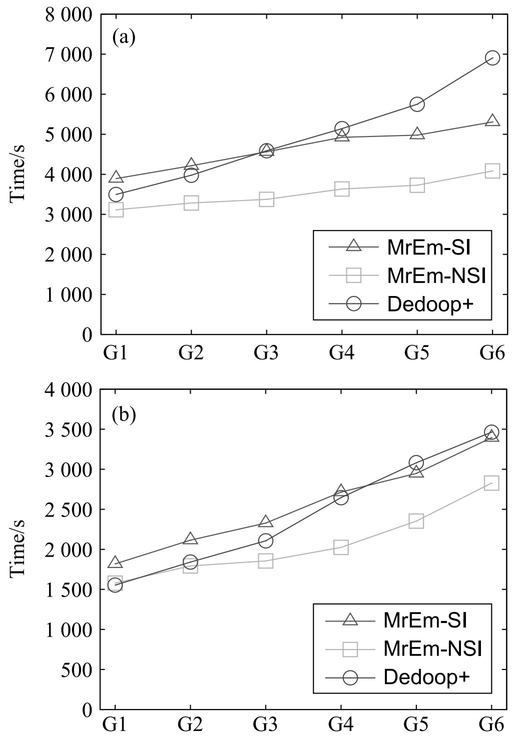


Fig. 9 Executing cost under diff. blocking function groups. (a) CiteSeer (500 000 records); (b) Twitter (1 000 000 records)

Figure 10 demonstrates the quartiles of 40 reducers at the *pair verification* stage with G1, G3, G5. The lowest line and the highest line in the quartile plot are the time when the first reducer and the last reducer finish the task. Besides, the lower bound and the upper bound in the box are the time when the 10th (= 1/4×40) and 30th (= 1/4×40) reducer finish the task while the median line shows the 20th (= 1/2 × 40) reducer's finishing time.

From this figure, we can observe that Dedoop+ is over a longer span than MrEm-SI and MrEm-NSI upon both data sets. *PairRange* can support load balancing with the assumption that the number of candidate pairs within all the reducers is nearly equal. Taking pair deduplication into consideration will lead to the fact that some reducers are free of comparing the candidate pairs. Thus it causes the workloads unbalanced and makes the performance worse. However, MrEm-NSI and MrEm-SI can both support load balancing and pair deduplication at the same time. Furthermore, MrEm-NSI behaves significantly better than MrEm-SI upon both data sets since MrEm-NSI can align sub-blocks better. In the following experiments, we choose G5 as the default blocking function group at the *block building* stage.

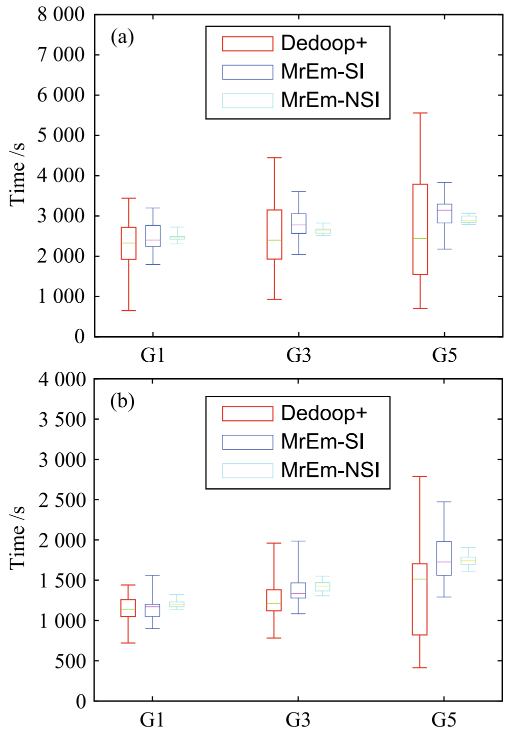


Fig. 10 The quartiles of reducers at the pair verification stage. (a) CiteSeer (500 000 records); (b) Twitter (1 000 000 records)

Figure 11(a) shows the performance of all solutions upon the CiteSeer data set with different data sizes. Among five solutions, MrEm-NSI behaves the best and MrEm-SI takes the second place. Since Dedoop+ still has the problem of load balancing, its execution time is nearly two times as much as that in MrEm-NSI. Naive supports neither pair deduplication nor load balancing, so its performance is lagging far behind the above three methods. PairUnit also performs bad because too many tiny sub-blocks (only contain one record pair) are generated and shuffled. With the increasing of data set, PairUnit will generate huge data to transfer from mappers to reducers.

Figure 11(b) shows the execution time of all solutions upon the Twitter data set by varying the data sizes. MrEm-NSI and MrEm-SI perform best because both of them consider load balancing and pair deduplication. An interesting observation is that PairUnit runs the fastest when data set size is small (500 000 records). In such a case, the number of candidate pairs is relative small, so that PairUnit can process it relatively quickly. However, the performance of PairUnit dramatically deteriorates when the data size increases, because much more data needs to be transferred from the mappers to the reducers.

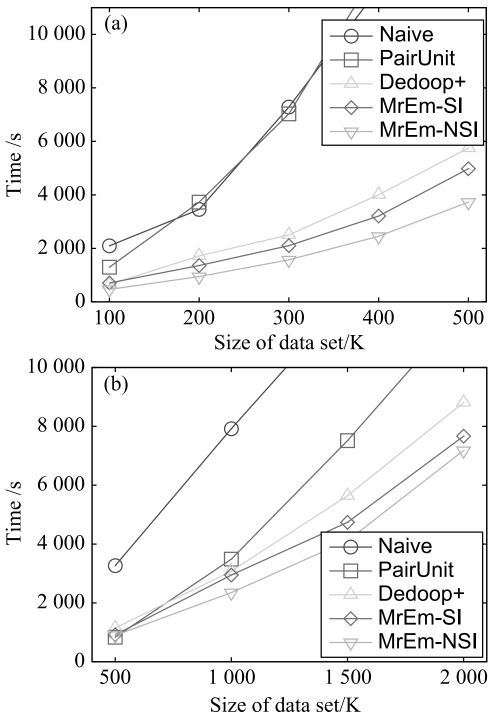


Fig. 11 Execution time. (a) CiteSeer; (b) Twitter

To illustrate the execution time more clearly, Table 4 shows the time spent in each stage for Dedoop+, MrEm-SI, and MrEm-NSI. In all situations, the pair verification stage dominates the overall time. Moreover, the pair verification cost of Dedoop+ is nearly two times as much as that in MrEm-NSI. Regarding to the interface implementation stage, MrEm-SI consumes more time than MrEm-NSI since it needs to split the large blocks at first while MrEm-NSI need not. Finally, regarding to the pair cleaning stage, the amount of input data determines the overall time.

5.3 Scalability testings

We continue to test the scalability of the proposed methods, including *speedup* and *scaleup*. In order to evaluate the speedup of our two solutions: MrEm-SI and MrEm-NSI, we fixed the data set size and varied the cluster size. Figure 12 shows the running time upon the CiteSeer data set (containing 100 000 records) when the number of reduce slots increases from 1 to 40. As aforementioned, we test the performance under different *reduce* slots because pair verification, the most costly step, is implemented at the reduce phase. Table 4 also shows that the cost on pair verification is over 80% of the overall cost for large data sets.

Table 4 Execution time in each stage/s

Solution	Stage	The size of data set		
		10^5	3×10^5	5×10^5
Dedoop+	BDM Building	62	91	151
	Pair Verification	587	2 414	5 600
MrEm-SI	Block Building	61	66	130
	Interface Implementation	67	108	530
	Pair Verification	484	1 678	3 914
	Pair Cleaning	96	251	404
MrEm-NSI	Block Building	61	66	130
	Interface Implementation	51	53	69
	Pair Verification	285	1 230	3 132
	Pair Cleaning	69	224	397

Figure 12(a) shows the execution time of above two solutions decreases with the increasing number of reduce slots. Figure 12(b) illustrates the "relative scale", which describes the ratio between the running time for the smallest cluster size (1-reduce slot) and that of the current setting. For example, for the 40-*reduce* slot case, the "relative scale" of MrEm-NSI is 22 (= 10 267/465). Figure 13 shows the running time

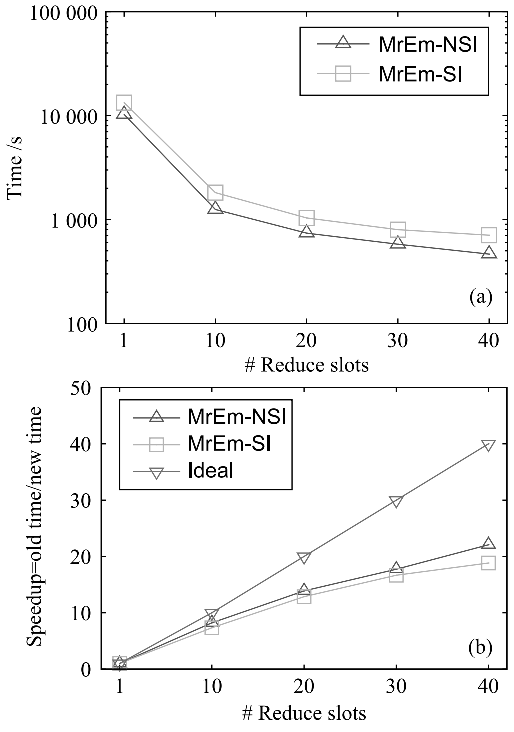


Fig. 12 Speedup on CiteSeer (100 000 records). (a) Running time; (b) relative running time

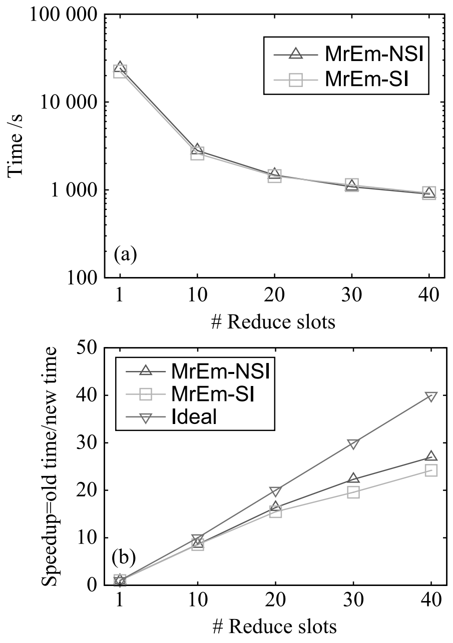


Fig. 13 Speedup on Twitter (500 000 records). (a) Running time; (b) relative running time

of our solutions upon the Twitter data set (containing 500 000 records) on clusters of 1 to 40 reduce slots. The speedup values of two solutions are quite good.

We then test the scaleup. A system with good scaleup performance tends to have similar running time when the workload increment rate is identical to that of the working nodes. Since the running time is strongly dependent on the number of candidate pairs, we set the increment of the working nodes proportional to that of the candidate pairs. For example, when the number of reduce slots is doubled, the number of candidate pairs is also doubled, even though the data set size is not doubled. Figure 14 shows the scaleup results on the CiteSeer and Twitter data sets. We observe that our solutions have good scaleup performance.

5.4 Evaluations of parameters

Finally, we evaluate some important parameters, including the number of hash functions h, signature size k, and MAXBS in the interface em (see Section 4). The first two parameters are for pair deduplication, and the last is for load balancing.

5.4.1 Pair deduplication

We use the number of candidate pairs to measure different parameters. It is worth noting that if there is no splitting large block, MrEm-SI performs exactly the same as MrEm-NSI. For the comparison purpose, we also calculate *Distinct* and *All*, where the former assumes all candidate pairs have been de-duplicated before verification, and the latter assumes no de-duplication will be performed for any candidate pair before verification.

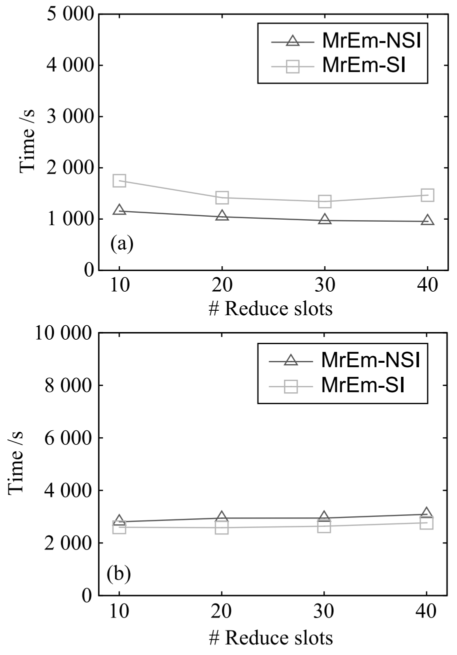


Fig. 14 Scaleup on two data sets. (a) CiteSeer; (b) Twitter

Figure 15 illustrates the change of the number of candidate pairs when varying h and k upon the CiteSeer (containing 10 000 records) and Twitter (containing 100 000 records) data sets respectively. In general, a smaller k will lead to better

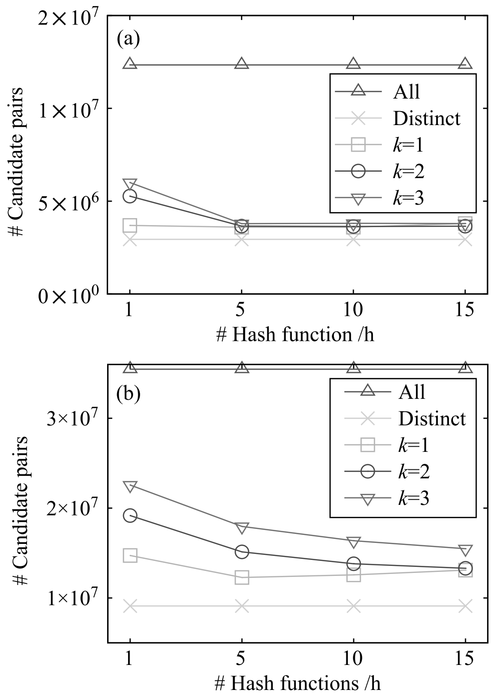


Fig. 15 The evaluation of h and k parameters. (a) CiteSeer (100 000 records); (b) Twitter (500 000 records)

performance. When h becomes bigger, the values remain stable with different k values. Since more signatures are generated with the increment of h and then, more duplicates can be detected so that the number of pairs to be verified is reduced. Another aspect is when we set h bigger than 5, the number of candidate pairs remains stable. Consequently, we set $h = 10$ and $k = 2$ as default.

5.4.2 Load balancing

MAXBS decides the size of sub-block after splitting. All experiments were conducted on the CiteSeer data set (containing 100 000 records). Figure 16(a) shows the number of candidate pairs under different MAXBS values. For the comparison purpose, we also draw a curve for MrEm-NoSplit which implements no block splitting strategy (for example, we set: MAXBS$=\infty$). We can observe that the MrEm-NSI curve is quite close to the MrEm-NoSplit curve, while the MrEm-SI curve is above the other two curves. Moreover, when MAXBS goes down, the number of candidate pairs of MrEm-SI will rise, while MrEm-NSI still remains stable. The reason is that MrEm-NSI can align the sub-blocks but MrEm-SI cannot.

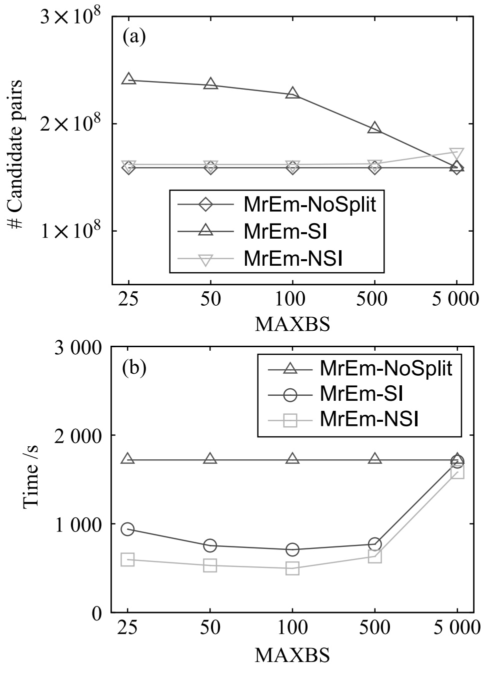


Fig. 16 The evaluation of MAXBS. (a) # Candidate pairs; (b) running time

Figure 16(b) shows the execution time under different MAXBS values. We execute all four stages of MrEm in the experiments. We observe that MrEm-NoSplit takes the longest time to execute, due to the influence of unbalanced workloads. Both MrEm-SI and MrEm-NSI support load balancing and pair deduplication, so they run much faster than MrEm-NoSplit. Moreover, MrEm-NSI performs better than MrEm-SI because more duplicates can be detected, making the overall time shorter. We can observe that if MAXBS is set too great, unbalanced workloads will lower the overall performance. Another observation is that the execution time rises when MAXBS is set too small, since a huge number of small sized sub-blocks are generated and shuffled. Hence, we recommend to set MAXBS = 100 as default under current circumstances. We should note that this value is dependent on the data amount and cluster environment.

6 Related work

The blocking-based method for entity matching generates several *blocking keys* independently for each original record, so that matching records can be pushed to the same block. Typical work includes standard blocking [10], sorted neighborhood method [11], q-gram based indexing [24], canopy clustering [25], locality sensitive hashing [12,13], StringMap [26] and so on.

The MapReduce framework is commonly used to analyze large-scale data nowadays. He et al. designed a MapReduce-based DBSCAN algorithm for heavily skewed data [27]. When applying blocking-based entity matching into the MapReduce framework, some new challenges should be overcome. Kolb et al. have proposed solutions to solve the inherent problem in the MapReduce framework: load balancing. Meantime, they also apply multi-pass sorted neighborhood method to MR framework [14]. Kolb et al. also proposed a simple solution to achieve pair deduplication [18]. Based on the previous work, Dedoop [19] is proposed for blocking-based entity matching upon MapReduce framework. However, Dedoop still has its own limitation when solving the problem of load balancing and pair deduplication at the same time.

Other applications like set-similarity joins to find similar strings, documents clustering to combine similar document together also face above two challenges when they use distributed framework. Since pair-wise computation is time-consuming, a set of signatures will be generated at first to avoid unnecessary comparisons. Vernica et al. [23] applied set-similarity joins to MapReduce framework and proposed three stages to provide end-to-end solutions. They partitioned data across nodes according to the token frequency in order to achieve load balancing and minimize the replication from mappers to reducers side. However, since two strings will share several identical prefixes, they still face the problem of verifying them in different nodes more than once. Refs. [28–30] proposed efficient methods for similarity joins by using MapReduce framework, they mainly focused on how to prune the dissimilar pairs, while load balancing was not considered deeply.

Moreover, there exist other popular parallel frameworks for data analysis, such as Spark[6]. Spark can work well upon the machine-learning situations when a number of iterations are inevitable, because it keeps a large number of tuples in memory to avoid I/O cost. For example, the k-means algorithm needs to re-compute the central point of each cluster after generating new clusters, until the central point does not change. As such step may repeat for many times, running upon the Spark framework is a better choice. However, the processing task will not repeat for so many times, which means the performance gap between Spark and Hadoop is insignificant.

7 Conclusions

In this paper, we study the problem of how to handle entity matching based on the MapReduce framework. Recently, some researchers try to consider MR-based entity matching using blocking function(s). Load balancing and pair deduplication are its intrinsic challenges. Our proposed solution can solve above two challenges at the same time. Analysis and experimental reports show that our novel solutions are effective, efficient and scalable.

Acknowledgements Our research is supported by the National Basic Research Program of China (2012CB316203), the National Natural Science Foundation of China (Grant Nos. 61370101 and U1501252), Shanghai Knowledge Service Platform Project (ZF1213), and Innovation Program of Shanghai Municipal Education Commission (14ZZ045).

References

1. Benjelloun O, Garcia-Molina H, Menestrina D, Su Q, Whang S E, Widom J. Swoosh: a generic approach to entity resolution. The VLDB Journal—The International Journal on Very Large Data Bases, 2009, 18(1): 255–276
2. Bilenko M, Mooney R J. Adadptive duplicate detection using learnable string similarity measures. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2003, 39–48
3. Guo S T, Dong X L, Srivastava D, Zajac R. Record linkage with uniqueness constraints and erroneous values. Proceedings of the VLDB Endowment, 2010, 3(1–2): 417–428
4. Li P, Dong X L, Maurino A, Srivastava D. Linkingtemporal records. Proceedings of the VLDB Endowment, 2011, 4(11): 956–967
5. Rastogi V, Dalvi N, Garofalakis M. Large-scale collective entity matching. Proceedings of the VLDB Endowment, 2011, 4(4): 208–218
6. Bilenko M, Kamath B, Mooney R J. Adaptive blocking: learning to scale up record linkage. In: Proceedings of the 6th IEEE International Conference on Data Mining. 2006, 87–96
7. Christen P. A survey of indexing techniques for scalable record linkage and deduplication. IEEE Transactions on Knowledge and Data Engineering, 2012, 24(9): 1537–1555
8. De Vries T, Ke H, Chawla S, Christen P. Robust record linkage blocking using suffix arrays and bloom filters. ACM Transactions on Knowledge Discovery from Data, 2011, 5(2): 9
9. Michelson M, Knoblock C A. Learning blocking schemes for record linkage. In: Proceedings of the National Conference on Artificial Intelligence. 2006, 440–445
10. Fellegi I P, Sunter A B. A theory for record linkage. Journal of the American Statistical Association, 1969, 64(328): 1183–1210
11. Hernández M A, Stolfo S J. The merge/purge problem for large databases. ACM SIGMOD Record, 1995, 24(2): 127–138
12. Gionis A, Indyk P, Motwani R. Similarity search in high dimensions via hashing. The VLDB Journal — The International Journal on Very Large Data Bases, 1999, 99(6): 518–529
13. Indyk P, Motwani R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing. 1998, 604–613
14. Kolb L, Thor A, Rahm E. Multi-pass sorted neighborhood blocking with MapReduce. Computer Science-Research and Development, 2012, 27(1): 45–63
15. Whang S E, Menestrina D, Koutrika G, Theobald M, Garcia-Molina H. Entity resolution with iterative blocking. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. 2009, 219–232
16. Kolb L, Thor A, Rahm E. Load balancing for MapReduce-based entity resolution. In: Proceedings of the 28th IEEE International Conference on Data Engineering. 2012, 618–629
17. Köpcke H, Thor A, Rahm E. Evaluation of entity resolution approaches on real-world match problems. Proceedings of the VLDB Endowment, 2010, 3(1–2): 484–493
18. Kolb L, Thor A, Rahm E. Don't match twice:redundancy-free similarity computation with MapReduce. In: Proceedings of the 2nd Workshop on Data Analytics in the Cloud. 2013, 1–5
19. Kolb L, Rahm E. Parallel entity resolution with dedoop. Datenbank-Spektrum, 2013, 13(1): 23–32
20. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113
21. White T. Hadoop: The Definitive Guide. 3rd ed. O'Reilly Media, Inc., 2012
22. Mitzenmacher M. Compressed bloom filters. IEEE/ACM Transactions on Networking, 2002, 10(5): 604–612
23. 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. 2010, 495–506

[6] http://spark.apache.org

24. Baxter R, Christen P, Churches T. A comparison of fast blocking methods for record linkage. ACM SIGKDD, 2003, 3: 25–27

25. Cohen W W, Richman J. Learning to match and cluster large high-dimensional data sets for data integration. In: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2002, 475–480

26. Jin L, Li C, Mehrotra S. Efficient record linkage in large data sets. In: Proceedings of the 8th International Conference on Database Systems for Advanced Applications. 2003, 137–146

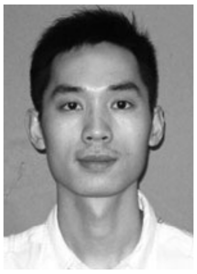
27. He Y B, Tan H Y, Luo W M, Feng S Z, Fan J P. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Frontiers of Computer Science, 2014, 8(1): 83–99

28. Das Sarma A, He Y Y, Chaudhuri S. Clusterjoin: a similarity joins framework using map-reduce. Proceedings of the VLDB Endowment, 2014, 7(12): 1059–1070

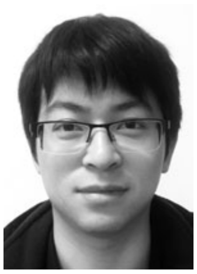
29. Deng D, Li G L, Hao S, Wang J N, Feng J H. Massjoin: a MapReduce-based method for scalable string similarity joins. In: proceedings of the 30th IEEE International Conference on Data Engineering. 2014, 340–351

30. Kim Y, Shim K. Parallel top-k similarity join algorithms using MapReduce. In: Proceedings of the 28th IEEE International Conference on Data Engineering. 2012, 510–521

Cheqing Jin is a professor at East China Normal University, China. He received his master and bachelor degrees from Zhejiang University (ECNU), China in 1999 and 2002 respectively, and his PhD degree from Fudan University, China in 2005, all in computer science. He worked as an assistant professor in East China University of Science and Technology, China from 2005 to 2008, afterwards he joined ECNU on October 2008. In 2003 and 2007, he visited the University of Hong Kong, China and the Chinese University of Hong Kong, China respectively. He has acted as the PC members for more than ten conferences. His main research interests include streaming data management, location-based services, uncertain data management, data quality, and database benchmarking.



Jie Chen received his undergraduate and master degree from East China Normal University, China in 2011 and 2014, respectively. As of now, he is working in PayPal, Risk Management team to be a risk analyst. His research area is data quality and data mining, especially for handling big data.



Huiping Liu received the BS degree in software engineering from East China Normal University, China in 2013. Currently, he is a PhD student supervised by Prof. Cheqing Jin. His research mainly focuses on data quality, massive data mining and processing, and location-based services.