J Supercomput (2019) 75:623–645
https://doi.org/10.1007/s11227-017-2008-8

Unsupervised blocking and probabilistic parallelisation for record matching of distributed big data

Chenxiao Dou[1] · **Yi Cui**[2] · **Daniel Sun**[1,3] · **Raymond Wong**[1] · **Muhammad Atif**[4] · **Guoqiang Li**[2] · **Rajiv Ranjan**[5]

Published online: 16 March 2017

Abstract *Record Matching* refers to identifying pairs of records that relate to the same entities across different data sources. In many applications of data mining, record matching is usually associated to quadratic complexity. In practice, the number of non-matching record pairs always far exceeds the number of matching pairs, and this is called *imbalance problem*. *Blocking* is a technique of data reduction, which can filter unlikely matching pairs before record matching. However, for big data there is no fast and effective blocking algorithm yet. In this paper, we report on big data infrastructure to improve efficiency of blocking. Our approach runs blocking process independently and distributedly on the partitions of whole data. To improve efficiency, we adopt a probabilistic technique to balance the speed and the effect of the algorithm that we proposed for distributed blocking. Our experimental analysis endorses the superiority of our technique and shows its novel scalability.

Keywords Big data · Record matching · Blocking · Density · Parallelisation

✉ Daniel Sun
daniel.sun@data61.csiro.au

1 University of New South Wales, Sydney, NSW, Australia

2 School of Software, Shanghai Jiao Tong University, Shanghai, China

3 Data61, CSIRO, Canberra, ACT, Australia

4 National Computational Infrastructure, Canberra, ACT, Australia

5 Newcastle University, Newcastle upon Tyne, UK

1 Introduction

Automatically linking records from different data sources becomes more and more important in many big data applications. A crucial step of integrating data from multiple sources is detecting and eliminating duplicate records. This process is called Entity Matching, Record Linkage, or Record Matching (RM) [1], which is a well-known problem in many applications such as address matching and citation de-duplication. The goal of Record Matching is to identify records that represent the same real-world entities from a variety of data sources.

Most of state-of-art methods adopt machine learning techniques to train a decision classifier for matching records. The learned matcher classifies each record pair as Match or Non-match based on the similarity between them. However, in the learning process, RM suffers a lot from high skewness of data distribution due to its inherently quadratic complexity. For instance, in a data set including M records, the number of all pairwise-created record pairs reaches $\theta(M^2)$, while the maximum number of truly matching pairs is M. Thus, it is difficult to sample a training set in which the number of matching pairs is relatively equivalent to the number of non-matching pairs. With continuously increasing volume of data sets, this imbalance problem becomes more and more severe.

Many blocking methods have been proposed to alleviate the imbalance problem [2–4]. The idea of blocking is to avoid evaluation of dissimilar pairs. In traditional methods, blocking criteria are designed manually and determined by human experience. However, such methods are unfriendly to users, especially if data sets are significantly big and dirty. Hence, unsupervised blocking algorithms become attractive in recent years. But the efficiency of unsupervised blocking algorithms is challenged by big data. First, the "Volume" of big data deteriorates efficiency of an algorithm because the amount of data to inspect can be expanded quadratically in record matching—that is—even "bigger" data. Second, the "Variety" of big data requests to inspect more details of data, and therefore more dimensions, i.e. data features, have to be considered for record matching.

To efficiently process huge and various data for record matching, a countermeasure is to run parallel blocking on those distributed big data platforms like Spark and MapReduce. However, transferring a sequential blocking algorithm to a distributed version is non-trivial, and not every unsupervised blocking approach is suitable for parallelisation. On distributed big data platforms, the distributed versions may even run slower than its sequential version: First, frequent read/write operations in memory and disk are time-consuming. For example, in MapReduce a new map task has to reload the data that are already sorted in the previous rounds; Second, frequent communication and data transferring between nodes take a considerable proportion of time overhead. In large-scale and geographically distributed data processing systems, users have to apply for remote computing resources, which locate in different availability zones. Hence, data exchange time between nodes in those clusters becomes a non-trivial factor influencing the running time of parallelised algorithms. Moreover, the latency of networks sometimes may exceed the actual computing time of algorithms.

In this paper, to efficiently scale up record matching for big data applications, we aim to improve an unsupervised and density-based blocking method *DUB* [5]

into a distributed version. Our proposed method first randomly separates input data into multiple partitions or use the data sets that have been already partitioned and distributed in large-scale systems. Then, our method runs multiple instances of the blocking algorithm *DUB* locally on each data set, from which each algorithm instance returns a local blocking boundary. The process of attaining local boundary requires no communication and data transferring inter-nodes. At the last step, we introduce a probabilistic method that concludes a global blocking boundary by merging all the returned local boundaries together, and this global boundary is guaranteed to be close to or be the same as the real boundary derived from running *DUB* in a single machine.

The contributions of this paper are summarised as follows:

1. We parallelise the density-based blocking algorithm into a distributed probabilistic blocking algorithm that has almost the same blocking performance with *DUB* but more scalable for big data.
2. The new algorithm significantly reduces system latency by running the blocking algorithm on each node independently.
3. The new algorithm is adaptable for both Spark and MapReduce, no matter how the computing cluster is organised.

This paper is a significant extension of our previous work [6], in which we pointed out the problem of big data record matching and discussed an early version of the technique in this paper.

This paper is structured as follows. We discuss related work in Sect. 2. In Sect. 3, we review the *DUB* algorithm and give the problem definition. Our distributed algorithm is elaborated and analysed in Sect. 4, including the method of parameter tuning. We evaluate our algorithm on the real data sets in Sect. 5. Finally, the paper is concluded in Sect. 6.

2 Related work

Record matching is a well-studied problem of determining whether two records refer to the same entities or not [1]. One of the most challenging difficulties is the imbalance problem that the non-matching record pairs always vastly outnumber the matching pairs [7–9]. This imbalance problem greatly affects the process of matching records. Most of matching algorithms [10–12] based on machine learning techniques hardly play well on a highly imbalanced input data.

To alleviate the imbalance issue, various blocking methods have been proposed to filter record pairs that are unlikely to be matched. The traditional blocking criteria are manually designed according to the attributes of data sets [13]. For saving human effort, learning techniques [2,3] are adopted to produce blocking criteria automatically. Whang et al. [4] proposed an iterative blocking framework, which enables record matching across different blocks. The above methods have a good performance on blocking accuracy, but their performance on efficiency is challenged by big data.

Using big data infrastructures to process the data-intensive tasks in parallel is a popular approach in both industry and academia community, and the analytic missions can be well managed and organised [14,15]. Distributed systems can support various query processing on large-scale data [16–19]. Recently with increasing size of data

sets, the number of record pairs that need to be blocked grows dramatically. To improve scalability, some parallel algorithms [20–22] exploit the power of multiple cores to reduce the time of matching process. In order to achieve a good balance on the number of comparisons in multiple nodes, Christen [23] introduces a technique that groups similar profiles by sets of keys, such that the comparisons are then executed inside the groups. Kolb et al. [24] proposed a parallel algorithm of Standard blocking methods [13]. This algorithm is an automatic data partitioning approach for the multi-pass Sorted Neighborhood (SN) method. They also adapted Sorted Neighborhood blocking method [25] into a distributed version in [26]. Efthymiou et al. [27] parallelised Meta-Blocking method [28] using MapReduce. The proposed algorithm could distribute workload evenly among cluster nodes. All the above have not taken data exchange and communication into account. In the real world, geographically large-scale distributed computation is possible, e.g. cloud computing in different availability zones. Hence, the efficiency of above approaches may be heavily influenced by the latency in geographically distributed clusters. In large-scale distributed systems, latency time can be accumulated along transmission paths and result in significant communication delay.

To reduce latency, some work [29–33] have been proposed to study the highly distributed clusters. G-Hadoop [29] supports data processing on heterogenous clusters using a two-layer management. The top layer governs all the inter-cluster master nodes to balance the resource scheduling; the bottom layer manages the intra-cluster slave nodes for the specific computing tasks. However, those authors have not specifically optimised the communication cost across the clusters. In the work [30], the authors facilitated Map/Reduce tasks executed on geo-distributed datasets. They optimised the data transferring by exploiting Data Transformation Graph (DTG). Using the shortest weighted path in DTG, the approach can bring a big cost-saving on the data communication. But the computational complexity of finding the shortest path is non-trivial. Luo et al. [31] proposed a hierarchical framework adopting the Map-Reduce-Global Reduce model that comprises three elementary functions: Map, Reduce, and Global Reduce. The hierarchical framework gathers computation resources from different clusters and runs Map/Reduce jobs simultaneously across them. In the work [32,33], the authors studied the execution of Map/Reduce jobs on clusters of virtual machines. Through optimising the allocation of virtual machines and the distribution of data partitions, the data locality can be improved, leading to the cross network traffic reducing.

Most of the existing approaches enhance the distributed system scalability by optimising the data distribution and task scheduling. But such improvement focuses more on the system level than on the algorithm level. Its effect to specific blocking algorithms may not be observable as well. In this paper, we study a density-based blocking method *DUB* [5] for large-scale record matching problem. The authors [5] discovered a widely existing property, Density Monotonicity. In a similarity space, the area of lower similarities is supposed to always contain more points than the area of higher similarities does. As this monotonicity can be obviously observed in many real data sets, *DUB* shows a strong capability on automatically and accurately blocking non-matching record pairs. However, when *DUB* is applied to distributed systems, *DUB* needs to access the data in all nodes to compute the global density, which brings

much communication overhead. This process of computing density hampers *DUB* from being efficiently parallelised.

3 Preliminaries

3.1 DUB algorithm review

In this section, we review one density-based unsupervised blocking algorithm, *DUB*, which was proposed for record matching problem in [5].

The traditional blocking methods using similarity as measure of matching have one common problem: Users have to manually set thresholds through many experimental trails, and the determination of the thresholds is usually difficult. Unlike the traditional methods, *DUB* is a blocking approach exploiting density to decide whether two records are matching or not. It was found that in similarity space, area of higher similarity often has lower density and the boundary of high-density area almost coincides with the class boundary. With this discovery of density, *DUB* algorithm can determine a complex blocking criterion by searching for the density boundary. As the density property can be observed in many real data sets, the blocking performance of *DUB* is practically high.

We give a brief introduction to *DUB* as follows. The first step of *DUB* is to map all the record pairs into a d-dimensional similarity space $[0, 1]^d$, each dimension of which represents a similarity measure used to estimate how likely a record pair is truly matching with a score between 0 and 1. For instance in Table 1, the upper table contains three citation records from DBLP data set, and the other has two records from ACM data set. From the tables, 6 record pairs can be created pairwise and only the pair created by the last records in both tables is the truely matching pair. Using *EditDistance* similarity on "Authors" and *Jaccard* similarity on "Title" as two dimensions of a similarity space, we can map the 6 pairs into 6 points in 2D similarity space. Take the only matching pair as example. Its *EditDistance* similarity score on "Authors" is 0.595, and its *Jaccard* similarity score on "Title" is 1.0. Then, the pair is mapped into the point (0.595, 1.0) in 2-D similarity space.

In practice, most of points from non-matching pairs crowd in low-similarity area of similarity space, and the boundary of high-density area almost coincides with the class boundary. Based on this observation, *DUB* searches for the boundary of region, resided by high-density points, in low-similarity area. The record pairs, that are mapped into the points inside such a region, are regarded as unlikely matching pairs and are blocked out then. The density here is defined as follows:

Definition 1 (*Density*) In a d-dimensional similarity space $[0, 1]^d$, given a point set $\mathbb{D}$ and a distance threshold r, *Density* of one point p, denoted as $\rho(p)$, is defined as the number of points, each of which has an Euclidean distance to p less than r. (In this paper, $\mathbb{D}$ is the set of points mapped from all input record pairs with d different similarity measures, and r is a user-given parameter.)

A boundary can be a set of arbitrary candidate points in $\mathbb{D}$, and hence, there are too many possible boundaries. In our work, *DUB* sets a *granularity parameter* k and only

Table 1 Citation dataset

Title	Authors	Venue	Year
Safe query languages for constraint databases	Peter Z. Revesz	TODS	1998
Efficient Filtering of XML Documents for Selective Dissemination of Information	Mehmet Altinel, Michael J. Franklin	Very Large Data Bases	2000
Standards for databases on the grid	Susan Malaika, Andrew Eisenberg, Jim Melton	ACM SIGMOD Record	2003
Title	**Authors**	**Conf**	**Year**
Database techniques for the World-Wide Web: a survey	D. Florescu, A. Levy, A. Mendelzon	SIGMOD	1998
Standards for databases on the grid	S. Malaika, A. Eisenberg, J.	SIGMOD	2003

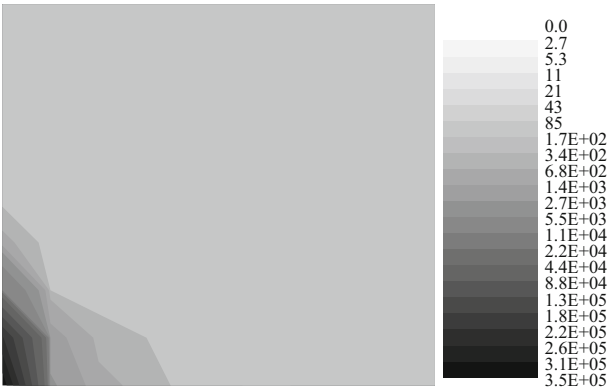

Fig. 1 Contour of citation pairs. The data are the publication records from DBLP and Google Scholar. The *panel on the left* is the similarity space of two similarities. The *right* is the contour levels, which represent the numbers of points per unit area

considers the virtual points $p = (p_1, \ldots, p_d)$ where $p_i = j/k$, $j \in \{0, 1, \ldots, k\}$ as the boundary points. Then, the number of candidate points for a boundary is limited to k^d. For example, in Fig. 2, with a *granularity* $k = 10$, only the points like $(0.1, 0.1)$ are considered as the boundary points, even if $(0.1, 0.1)$ may not be a true point in $\mathbb{D}$. There, the number of candidate boundary points is 10^2.

To accelerate the process of searching boundary, *DUB* adopts a widely existing property about the monotonicity of density, that is, in similarity space the area of lower similarities is supposed to often contain more points than the area of higher similarities does, vice versa. This property exists in many real data sets. Figure 1 shows this monotonicity explicitly on a citation data set. *Monotonicity of Density* is formally defined in the following:

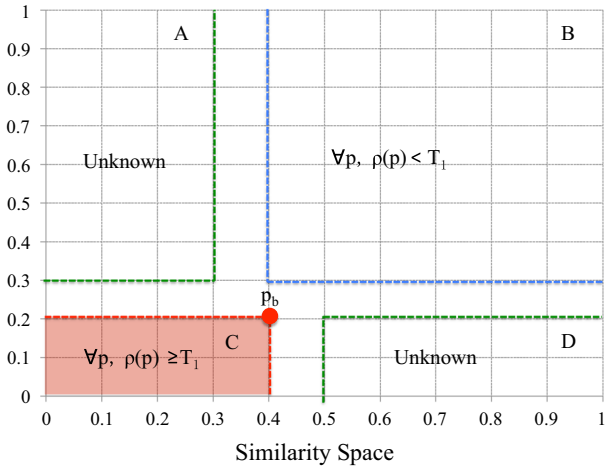

Fig. 2 Three regions in the similarity space

Definition 2 (*Domination*) In a d-dimensional similarity space $[0, 1]^d$, given two points $p = (p_1, \ldots, p_d)$ and $p' = (p'_1, \ldots, p'_d)$, if $p_i \leq p'_i$ for all $1 \leq i \leq d$, p' *dominates* p, denoted as $p \preceq p'$; if $p \preceq p'$ and $p_i \neq p'_i$ for some $1 \leq i \leq d$, it is denoted as $p \prec p'$.

Definition 3 (*Monotonicity*) For any two points p and p' such that $p \preceq p'$, if $\rho(p) \geq \rho(p')$, the density is monotonic with respect to Domination.

With the monotonicity, *DUB* runs in two main phases:

1. In the first phase, the algorithm runs *Binary Search* on the d-dimensional similarity space to find the boundary points that $\rho(p) \geq T_1$, where T_1 is a predefined density threshold. When one boundary point is located, it divides the similarity space into three regions: unknown density, density at least T_1, and density less than T_1. In the region of 'unknown' density, the algorithm repeats the process of finding the boundary point on the remaining 'unknown' region until no more unknown region remains. After the boundary is found, the points inside the boundary will be blocked.
   For example, when the algorithm finds the first boundary point P_b such that $\rho(P_b) \geq T_1$, according to *Monotonicity of Density*, the similarity space is divided into three parts as shown in Fig. 2. For a point p in C, it has $\rho(p) \geq T_1$; for a point p in B, it has $\rho(p) < T_1$; for a point p in both $A$and D, $\rho(p)$ is uncertain. To be aware of the whole space, the algorithm repeatedly searches for the boundary points in A and D.
2. The monotonicity property in some areas may not be available. Hence, in the second phase, without *Monotonicity of Density*, the algorithm uses *Brute-Force Search* to enumerate the points that $\rho(p) \geq T_2$ where T_2 is the other density threshold less than T_1. The algorithm starts from one boundary point found in the first phase, and then exhaustively checks the neighbour points. If a neighbour point

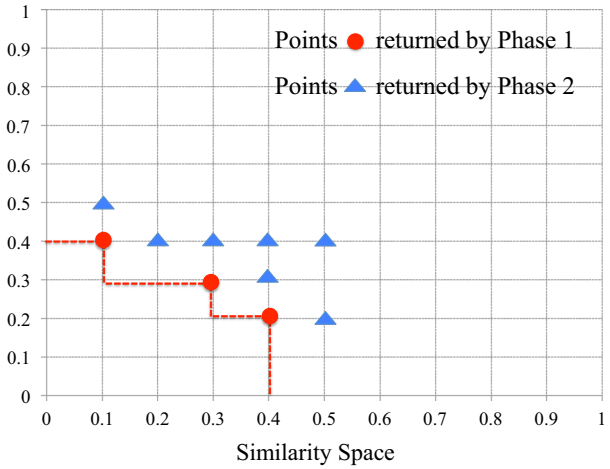

Fig. 3 An example of found boundary

p, $T_1 > \rho(p) \geq T_2$, is found, the algorithm will repeatedly check the neighbours of p until no new point is found. After all enumerated points are checked, the points within a distance r to any enumerated point will be blocked.
The result of each phase above is a set of points. Figure 3 shows an example of the result after the phases.

3.2 Problem and metrics

Given two input record tables R and S, we need to join them to generate all possible pairs of records. Each record pair will be mapped into a point in the similarity space described in the last section. As each record in R has to be compared with every record in S, the set of all possible candidate pairs is huge. For example, if $|R| = 10^3$ and $|S| = 10^6$, the number of candidate pairs will reach 10^9. It is hard to deal with such a huge amount of data in a single computer. The time cost and space cost are intolerable in practice.

DUB is designed for accurate blocking in centralised computing. However, for big data, its efficiency may be rather pessimistic. Therefore, adapting *DUB* to popular big data software infrastructure becomes a necessary task towards efficient blocking for big data. In Hadoop eco-systems, when *MapReduce* or *Spark* framework is adopted, a data set needs to be partitioned into small parts, which are processed in distributed clusters. Since *DUB* is a heuristic algorithm driven by the variation of density gradient in similarity space, naively adapted *DUB* may have to pay an expensive price for data communication and may be even slower than original *DUB*. Thus, the problem that we are studying is how to make *DUB* efficiently parallised on the large-scale system without loss of blocking accuracy.

Running Time is the metric to evaluate the blocking efficiency and is denoted as the job time of the blocking algorithm on Spark or MapReduce. *Recall*, *Reduction Ratio*

are the measures for evaluation on the blocking accuracy, and defined in the following. Denote the set of all record pairs as $\mathbb{C}$, the set of all matching pairs as $\mathbb{C}_{\text{match}}$, and the set of all non-matching pairs as $\mathbb{C}_{\text{non}}$, the set of blocked record pairs by proposed blocking algorithm as $\mathbb{B}$.

$$Recall = 1 - \frac{\sum_{x \in \mathbb{C}_{\text{match}}} 1[x \in \mathbb{B}]}{|\mathbb{C}_{\text{match}}|} \tag{1}$$

$$Reduction\ ratio = \frac{\sum_{x \in \mathbb{C}_{\text{non}}} 1[x \in \mathbb{B}]}{|\mathbb{C}_{\text{non}}|} \tag{2}$$

Our goal is to design a distributed algorithm named *DDUB* that can block the record pairs in a short *Running Time* and have high *Recall* and *Reduction Ratio*.

3.3 Naive distributed solution

A naive method to implement distributed *DUB* is to straightly make the process of computing one point's density in parallel. A naive parallelisation of *DUB* first randomly splits the data set into equally sized small parts and then assigns them to N nodes. When *DUB* needs to query the density of one point, each node is invoked to compute the density of that point locally on its local data set. After all local densities are known, the sum of them is the global density. In this way, the output set of boundary points will be exactly same to that found by *DUB*.

We name this naively distributed *DUB* to be *NDUB* in this paper. Figure 4 is an example about how *NDUB* searches for the boundary points in parallel with 3 nodes. Similar to *DUB*, with the density monotonicity, *NDUB* also plays *Binary Search* to find the boundary points. Whenever a point's density is queried, the algorithm will map the point to the nodes N_1, N_2 and N_3. Then, N_1, N_2 and N_3 compute the point's local density, respectively. The sum of returned local densities is the global density. Through many rounds, P_1, P_2, and P_3 are found as the boundary points.

The naive method has a big drawback that the nodes communicate too many times in the whole process. Whenever one point's density is queried, all nodes have to communicate to sum up the global density, leading to a poor efficiency.

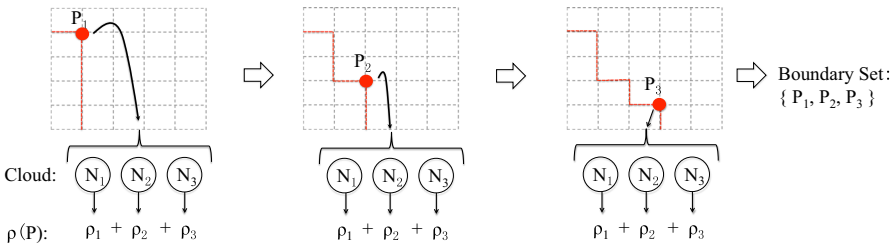

Fig. 4 Example of *NDUB*

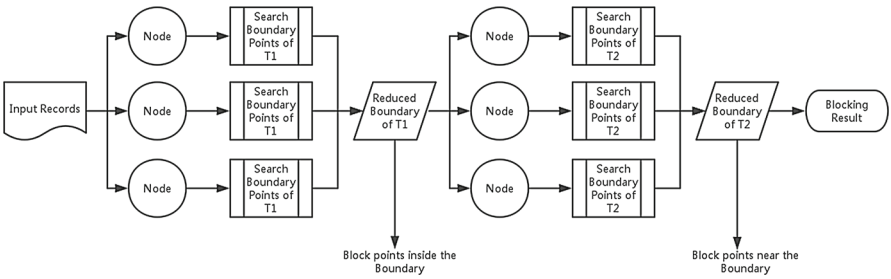

Fig. 5 Work flow of *DDUB*

4 Unsupervised blocking for big data

In Sect. 3, we have discussed about *Monotonicity of Density*, which can be observed over many data sets of record matching. This property still holds approximately in randomly sampled subsets from the whole input data set, if the input data set is sufficiently large, i.e. big data. This is the cornerstone of our proposed *DDUB* in the following. Through randomly delivering each record pair into N nodes, the entire data set is partitioned into N parts, each of which is a sample set of the original data set. Then, *DUB* is able to run independently on each node. As each record pair is randomly and independently distributed, according to the density defined in Sect. 3, it is easily seen that for a specific point, its density computed on each data partition, i.e. local density, is expected to be $1/N$ of its density in the entire input data set, i.e. global density. Consequently, given a density threshold T, if we know that in most of computing nodes a point has its local density larger than T/N, very likely the point has a global density over T.

Based on the above motivation, we now make a formal description of our algorithm, *DDUB*. Given a point p, let $\rho_i(p)$ be the point's local density computed on the i-th node and $\rho(p)$ be the point's global density computed on the entire input data set. If there exist at least m nodes returning $\rho_i(p) \geq \frac{T}{N}$, we will regard the global density of point p larger than T, $\rho(p) \geq T$. m is a user-defined threshold on the number of nodes where $\rho_i(p) \geq \frac{T}{N}$. The detail about how to determine m will be discussed in Sect. 4.2. After running the blocking algorithm on each node, we introduce a merge strategy to aggregate the blocking results. The workflow of our approach is given in Fig. 5.

4.1 Probabilistic parallelisation

DUB sets two density thresholds for blocking the mapped points of record pairs in similarity space: a higher one T_1 for blocking the points in the region where the monotonicity property applies, and a lower one T_2 for the points in the region where the monotonicity property does not strictly hold. As described in Sect. 3, the algorithm first blocks the points with a density no less than T_1 and then blocks the points having a density no less than T_2 in the rest part. Our *DDUB* tries to achieve the same accuracy performance as *DUB* can do, and for this reason, we need to adjust the thresholds for

each data partition. After the entire data set has been randomly split into N partitions, the expected size of each partition is $1/N$ of the original size and we use T_1/N instead of T_1 as the higher threshold and T_2/N instead of T_2 as the lower threshold.

With the new thresholds, we perform *DUB* independently on each node. But in runtime, the nodes do not block points directly at once. In our *DDUB*, we set a user-defined parameter m. Only if a point's local density on at least m nodes is not less than T_1/N in the first phase or T_2/N in the second phase, we will regard its global density no less than T_1 or T_2 and block the point on every node. For simplicity, in the following, we define two new terms: *coverage* and *majority*.

Definition 4 (*Coverage*) Given a point p and one point set Q, if $\exists q \in Q$ and $p \preceq q$, we say Q *covers* p, denoted as $p \sqsubset Q$.

Definition 5 (*Majority*) Given a point p and N nodes, with a threshold T, we denote the *majority* $\Gamma_T(p)$ to be how many nodes return a local density of p no less than T.

In general, *DDUB* also has two main phases. First, it searches for all the points p such that $\Gamma_{\frac{T_1}{N}}(p) \geq m$, and blocks them out. Then, on the rest points, it searches for and filters out the points with $\Gamma_{\frac{T_2}{N}}(p) \geq m$.

4.1.1 Search for the points with $\Gamma_{\frac{T_1}{N}}(p) \geq m$

In *DUB*, the first phase is to use *Monotonicity of Density* to locate the region with its density no less than T_1 and return a set of boundary points for this region. In this paper, as we run *DUB* separately in N nodes, each will return a set of boundary points with the threshold T_1/N for its own data partition. Then, we need to search for the points under $\Gamma_{\frac{T_1}{N}}(p) \geq m$ from the N boundary sets. In this phase, our goal is to confirm a new region boundary that the points inside the region are most likely to have a global density no less than T_1.

Take two nodes for example. Set $m = 2$. The returned boundary points are shown in Fig. 6. The boundary set Q_A returned by Node A is $\{(0.2, 0.8), (0.4, 0.4)\}$ and the boundary set Q_B returned by Node B is $\{(0.2, 0.6), (0.6, 0.2)\}$. Then, according to the definition of *coverage*, all the points p with $p \preceq (0.2, 0.6)$ or $p \preceq (0.4, 0.2)$ are *covered* by both Q_A and Q_B and have $\Gamma_{\frac{T_1}{N}}(p) = 2$. The area bounded by $\{(0.2, 0.6), (0.4, 0.2)\}$ is the mutual area of the two local boundaries. So the new boundary set with $m = 2$ is $\{(0.2, 0.6), (0.4, 0.2)\}$.

A similarity space may have a big dimension number d and a small granularity value k, such that the number of points that need to be evaluated may be considerably big. Consequently, enumerating the points to check their $majority$ become rather low-efficiency. In order to quickly confirm the new boundary, we discover another important property *Monotonicity of majority*.

Monotonicity of Density is an important property for locating the boundary points with the threshold T_1 in *DUB* (see Definition 1-3). Based on this property, another property, *Monotonicity of majority*, can be concluded, as described in the following:

Theorem 1 *Given* Monotonicity of Density *and* N *returned boundary sets with threshold* T*, for two points* $p \preceq p'$*, it must have* $\Gamma_T(p) \geq \Gamma_T(p')$.

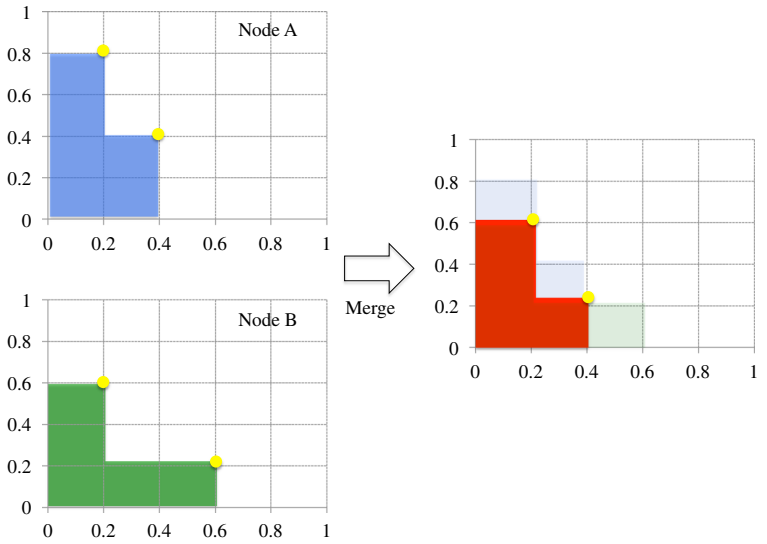

Fig. 6 Example of majority: $m = 2$

Proof Given two points $p \preceq p'$ subject to $\Gamma_T(p) < \Gamma_T(p')$, there must exist one node having $\rho(p) < T$ and $\rho(p') \geq T$, such that $\rho(p) < \rho(p')$. But according to *Monotonicity of Density*, for $p \preceq p'$, on each node we should have $\rho(p) \geq \rho(p')$. This contradicts $\rho(p) < \rho(p')$. Therefore, given *Monotonicity of Density*, with $p \preceq p'$, we can only conclude that $\Gamma_T(p) \geq \Gamma_T(p')$. □

This theorem tells a fact that if the density monotonicity holds, the majority is also monotonic w.r.t. $\preceq$. With the *Monotonicity of majority*, we can perform *Binary Search* on finding the overall boundary points that $\Gamma_{\frac{T_1}{N}}(p) \geq m$.

The following algorithm is similar to the procedure using the density monotonicity to find the density boundary points in the first phase of *DUB*. The only difference is that we check out if the *majority*, rather than *density*, of a point is less than the threshold. When a boundary point p is found, according to *Monotonicity of majority*, for all the points $p' \preceq p$, we can have $\Gamma_{\frac{T_1}{N}}(p') \geq m$ and for all the points $p'' \succ p$, $\Gamma_{\frac{T_1}{N}}(p'') < m$. Then, we split the similarity space into three regions: one consisting of the points with $\Gamma_{\frac{T_1}{N}}(p) \geq m$, one consisting of the points with $\Gamma_{\frac{T_1}{N}}(p) < m$, and another consisting of unknown points. For the region where the *majority* of points is unknown yet, we repeat the process of finding the boundary points with *majority* no less than m, until no more 'unknown' region remains. The detailed algorithm is as follows.

In Fig. 7, we give an example of how our algorithm works. At beginning, *DUB* is independently run on three nodes. Before each of them returns a boundary set based on their own data partition, there is no reducing task or node communication at all. When the three local boundary sets returned from nodes, we run Alg. 1 to find the

Input : Majority threshold m
Output: The boundary set $\mathbb{M}$

Procedure `FindBoundarySet(`m`)`
  $\mathbb{U} = \{(1/k, \ldots, 1/k)\}$ `/* U is the set of points with unknown Majority */`
  $\mathbb{M} \leftarrow \varnothing$ `/* M is the boundary set */`
  foreach *point* $p \in \mathbb{U}$ **do**
    if $\Gamma_{\frac{T}{N}}(p) \geq m$ **then**
      $p_{new} \leftarrow$ **FindBoundaryPoint(** p, m **)** `/* Binary Search is exploited to find` p_{new} `*/`
      $\mathbb{M} \leftarrow \mathbb{M} \cup p_{new}$ `/*` p_{new} `is a Boundary Point */`
      $\mathbb{U} \leftarrow$ **UpdateUnknownPointSet(** $\mathbb{U}, p_{new}$ **)** `/*` p_{new} `is used to confirm the new Parts of Unknown Majority and Parts of Known Majority. Similar to Fig. 1 */`
    end
  end
  Return $\mathbb{M}$

Algorithm 1: Search for the overall *Boundary* set

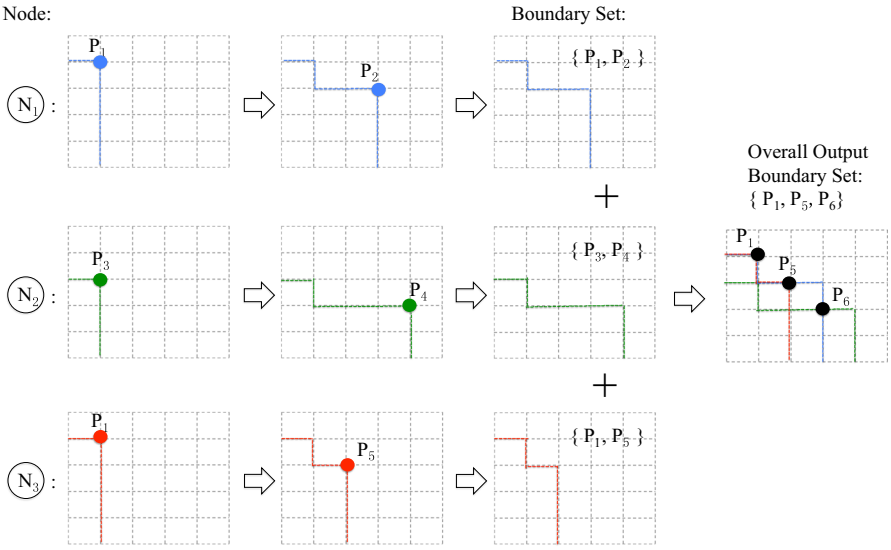

Fig. 7 Exampe of *DDUB*: $m = 2$

region of points *covered* by at least two boundary sets. The set of points determining such a region is the new boundary set and is used as the input for the next phase.

4.1.2 Search for the points with $\Gamma_{\frac{T_2}{N}}(p) \geq m$

After the overall boundary set under the threshold T_1 is concluded, our algorithm will deliver the new boundary set to each node. Then, we will begin the second phase to search for the points with $\Gamma_{\frac{T_2}{N}}(p) \geq m$. There are two steps in this phase. First, as

the region with the density no less than T_1 is confirmed, each node need to remove the points inside this region from its data partition. Second, in the place where the monotonicity does not strictly hold, each node need to enumerate the points with the density no less than $\frac{T_2}{N}$.

According to *Monotonicity of Density*, all points that are *covered* by the global boundary set should have a density at least T_1. After the first phase, the remaining set on each node should be rather small. We start the second phase of *DDUB* in each node with a threshold $\frac{T_2}{N}$. The algorithm will enumerate the points with the density no less than $\frac{T_2}{N}$ on the rest data set and return a point set with the density $\frac{T_2}{N}$. On the N returned enumeration sets, we need to find all the points p that $\Gamma_{\frac{T_2}{N}}(p) \geq m$.

As in the second phase of *DUB* the density monotonicity property cannot strictly hold, *Monotonicity of Majority* does not hold either. After all nodes return the enumeration sets, we have to search for the points appearing in at least m enumeration set. Since the collection of candidate points is rather small compared to the entire data set, and the collection must be a subset of the union set of all returned enumeration sets, we construct the union set first and then check the *majority* of each element incrementally. The process is described in the following.

Let E_i be the enumeration set returned from the i-th node. We construct the candidate enumeration set E, $E = E_1 \cup E_2 \cdots \cup E_N$. For every point p in E, our algorithm will check its *majority*. If $\Gamma_{\frac{T_2}{N}}(p) < m$, it means point p has a high possibility to satisfy $\rho(p) < T_2$. Then, we remove it out from E. After the algorithm exhausts all the elements in E, E becomes the target set of points with a global density no less than T_2.

Finally, E will be assigned to each node. Then, our algorithm will block all the points that are within r distance of any point in E. The rest points of record pairs in all nodes are our final result. By now, the blocking is finished.

4.2 Parameter tuning

Since we aggregate the results from distributed nodes into a global conclusion, it is necessary to guarantee that the aggregated result is the same as or very close to the result obtained from a sequential algorithm, *DUB*. In this section, we provide an analysis, based on which we can correctly tune our parameters.

First, let us recall our approach. According to the definition of density, $\rho(p)$ is the number of points near point p within a distance r on all the record pairs and $\rho_i(p)$ is the density of point p returned by the i-th node. For simplicity, in this section, we use ρ and ρ_i to represent $\rho(p)$ and $\rho_i(p)$. It is obvious that $\rho_1 + \cdots + \rho_N = \rho$. Since the process of data deployment is random and uniform, each point q s.t. $||q - p|| \leq r$ has the equal probability $1/N$ to appear in every node, that is, $E[\rho_i] = \frac{\rho}{N}$. For this sake, in each node, we tune the density threshold to be $\frac{T}{N}$, where T is a fixed density threshold for ρ. This procedure is same to the process of throwing balls into bins.

In our distributed algorithm, m is set as a *majority* threshold to determine whether a point should be deemed as a high-density point. However, given a value of m, a point that has been deemed distributedly as a high-density point is possibly against

the density threshold, that is, $\rho \geq T$ may not hold. By all means, we should be aware of this probability that majority threshold is not consistent with density threshold.

Lemma 1 *Let x be the random variable representing the least number of nodes, where there are more than n points. Given ρ and $nm \leq \rho$, the probability of $x \geq m$, i.e. $Pr\{x \geq m|\rho\}$ is Eq. 3.*

Proof Since at least m nodes have at least n points, the bin-ball game can be split into two phases: First, we take nm points from ρ and evenly assign the points into m nodes chosen from N nodes. In this phase, there are $\binom{\rho}{nm}$ possibilities for the assignment. Second, we freely assign the remaining points to all the N nodes. There are $N^{\rho-mn}$ possibilities. Totally, we have $\binom{\rho}{nm}N^{\rho-mn}$ ways to assign the points. Well, randomly assigning points to the nodes has N^{ρ} possibilities. Thus, we summarise the above to derive the probability of at least n points in at least m nodes

$$\begin{aligned} Pr\{x \geq m|\rho\} &= \frac{\binom{\rho}{nm}N^{\rho-mn}}{N^{\rho}} \\ &= \frac{\binom{\rho}{nm}}{N^{mn}}. \end{aligned} \tag{3}$$

□

When we run our distributed algorithm, we actually do not know ρ, but we can know the global distribution of ρ by sampling in preprocessing. Because similarity space has *monotonicity of density*, we only sample the number of points along with the density decreasing paths. Thus, we can easily have $Pr(\rho)$, given a distance to the highest density point. Then, the probability of $\rho > T$ is shown in the following theorem.

Theorem 2 *When there are at least n points in at least m nodes, the probability of $\rho > T$ is Eq. 8.*

Proof Given at least n points in at least m nodes, we can know that the minimum value $\rho_{\min}$ of ρ is mn. According to *Monotonicity of Density*, the maximum value $\rho_{\max}$ of ρ is $\rho(p_{\min})$, where $p_{\min} = (1/k, \ldots, 1/k)$ is the virtual point such that there is no other point p satisfying $p \preceq p_{\min}$. For $\rho > T$, ρ should range in $[T, \rho_{\max}]$. When we observe $x \geq m$, there are only two possibilities, either $\rho \geq T$ or $\rho < T$. According to the law of total probability, we know

$$\begin{aligned} Pr\{x \geq m\} &= \sum_{\rho=nm}^{\rho_{\max}} Pr\{x \geq m|\rho\}Pr\{\rho\} \\ &= Pr\{x \geq m|\rho \geq T\}Pr\{\rho \geq T\} \\ &\quad + Pr\{x \geq m|\rho < T\}Pr\{\rho < T\}, \end{aligned} \tag{4}$$

$$\sum_{\rho=nm}^{\rho_{\max}} Pr\{x \geq m|\rho\}Pr\{\rho\} = \sum_{\rho=nm}^{\rho_{\max}} \frac{\binom{\rho}{nm}}{N^{mn}} Pr\{\rho\},$$

$$Pr\{x \geq m|\rho \geq T\} = \sum_{\rho=T}^{\rho_{\max}} Pr(x \geq m|\rho) = \sum_{\rho=T}^{\rho_{\max}} \frac{\binom{\rho}{nm}}{N^{mn}}, \tag{5}$$

$$Pr\{x \geq m|\rho < T\} = \sum_{\rho=mn}^{T-1} Pr(x \geq m|\rho) = \sum_{\rho=mn}^{T-1} \frac{\binom{\rho}{nm}}{N^{mn}}, \tag{6}$$

and

$$Pr\{\rho < T\} = 1 - Pr\{\rho \geq T\}. \tag{7}$$

By summing up, we can resolve $Pr\{\rho \geq T\}$,

$$Pr\{\rho \geq T\} = \frac{\sum_{\rho=nm}^{\rho_{\max}} \left(\frac{\binom{\rho}{nm}}{N^{mn}} Pr\{\rho\}\right) - \sum_{\rho=mn}^{T-1} \frac{\binom{\rho}{nm}}{N^{mn}}}{\sum_{\rho=T}^{\rho_{\max}} \frac{\binom{\rho}{nm}}{N^{mn}} - \sum_{\rho=mn}^{T-1} \frac{\binom{\rho}{nm}}{N^{mn}}} = \frac{\sum_{\rho=nm}^{\rho_{\max}} \left(\binom{\rho}{nm} Pr\{\rho\}\right) - \sum_{\rho=mn}^{T-1} \binom{\rho}{nm}}{\sum_{\rho=T}^{\rho_{\max}} \binom{\rho}{nm} - \sum_{\rho=mn}^{T-1} \binom{\rho}{nm}} \tag{8}$$

□

By tuning n and m, we can make the above probability sufficiently big and guarantee that the aggregated boundary is probabilistically correct. In this paper, we have set $n = \lfloor \frac{T}{N} \rfloor$ for the above analysis, and thus we only need to tune m.

5 Experiments

We conduct a series of experiments to evaluate the effectiveness of our method *DDUB* in three aspects: *Recall, Reduction ratio*, and *Running Time*.

5.1 Settings and configurations

– *Environment*:
we run our experiments in Nectar cloud[1] using up to 16 nodes, each of which has two 2.5 GHz-cores, 8GB RAM and Ubuntu 15.10; and the underlying infrastructures are Hadoop 2.7.2 and Spark 2.0.1.

[1] http://nectar.org.au/.

– *Data*:
we use four data sets for evaluation: *Goods*, *Small Citation*, *Middle Citation* and *Large Citation*.[2] The volume of the data sets is 1.3GB, 12.2GB and 126.4GB, respectively. *Goods* has 4 attributes: *title, description, manufacturer and price*. The other three data sets have 4 attributes: *title, authors, venue and year*. One real-world entity can have two different records in the data set, e.g. Table 1
  1. *Goods* is a data set recording the product information from Google and Amazon. There are 4590 total unique records. It has 1.0×10^7 possible record pairs for matching. Among these record pairs, there exist only 1300 matching ones. As *Goods* has the same size with *Small Citation*, there is no need to test the efficiency on both sets. We do not use *Goods* for efficiency experiment.
  2. *Small Citation* is one collection of publication records. *Small Citation* has 4910 records sourced from DBLP and ACME. It has 1.2×10^7 pair entries in which 2224 pairs are truly matching.
  3. *Middle Citation* is a duplicated data set. We duplicate the data set *Small Citation* 10 times to be a new data set *Middle Citation*. It has about 1.2×10^8 pair entries. As *Middle Citation* is the duplicate of *Small Citation*, we only test the efficiency on it.
  4. *Large Citation* is the biggest data set among the three data sets. *Large Citation* has the same data type with *Small Citation*, recording the publication information. Its 66879 citation records are downloaded from DBLP and Google Scholar. The number of pairwise-created record pairs is 2.2×10^9, in which 5347 pairs refer to the same entities.

– *Algorithms*:
DUB is the base-line algorithm for comparison in this paper. As it is a sequential algorithm, we do not compare its efficiency with the other two distributed algorithms but evaluate its *Recall* and *Reduction Ratio* on the validation data sets. *NDUB* is the naive version of decentralised *DUB*, and has been described in Sect. 3. As it is implemented in a distributed way, we mainly take it as an efficiency competitor to *DDUB*, which is our proposed distributed algorithm.

5.2 Prior discussion

Spark and MapReduce are two popular big data platforms. MapRedcue is a disk-based system, while Spark is a memory-based system. Generally speaking, for processing the same amount of data that can be fully fed into the memory, Spark runs above 10 times faster than MapReduce does. With RAM price decreasing, recently, more and more users prefer to build the computing cluster on Spark. However, MapReduce still maintains its dominance in the area such like system reliability and data capacity. Furthermore, though Spark performs much better on Multi-Pass algorithms than MapReduce, for One-Pass algorithm, MapReduce and Spark have almost the same efficiency performance. Thus, we compare the algorithms on the both platforms. An algorithm that works well on the both platforms is more welcomed.

[2] http://dbs.uni-leipzig.de/en/research/projects/object_matching.

5.3 Blocking accuracy

In the context of this paper, parallelising an algorithm should not invoke too much accuracy loss. To investigate the blocking effect, two major measures are used: *Recall* and *Reduction Ratio*. They have been defined in Sect. 3.

In the following experiments, we first compare *DDUB* with *DUB*. *NDUB* is not compared because *NDUB* is a distributed implementation of *DUB* with zero-loss of accuracy. Its *Recall* and *Reduction Ratio* will always be the same as those of *DUB*.

In *DUB*, *NDUB* and *DDUB*, as their density definitions are same, we run the algorithms using the same parameters. To achieve a satisfying result, the values of r, k, T_1 and T_2 should be carefully selected. Since k directly determines the number of binary search rounds on each dimension, for the sake of efficiency, we default $k = 50$. According to the density definition, the density of one point is defined as the number of points within r neighbouring to the point. Thus, the combination of $\{r, T_1, T_2\}$ can decide the result of blocking. To find a suitable combination of $\{r, T_1, T_2\}$, we randomly sample a small data set from the input data set and run *DUB* with different combinations of $\{r, T_1, T_2\}$. The combination that can prune as many instances as possible is preserved as the output parameter combination. As the found T_1 and T_2 come from the small sample set, at the last, we need to scale up T_1 and T_2 to the size of the input data set.

Our algorithm is also affected by the number of nodes indeed. Based on the analysis in Sect. 4.2, the number of nodes has an immediate impact on the selection of threshold m. So we vary the number of nodes N in $\{2, 4, 8, 16\}$ and set m equal to the number in $\{1, 3, 5, 13\}$, accordingly.

The results on the data sets *Goods* and *Small Citation*are given in Figs. 8 and 9. For the concern of experiment time, we only test the huge data set *Large Citation* with 16 nodes and *DUB* gets *Recall* 0.864 and *Reduction Ratio* 0.981; *DDUB* get *Recall* 0.864 and *Reduction Ratio* 0.977. The results show that by setting a proper value of m, the loss of *Recall* and *Reduction Ratio* of *DDUB* can be very small. No matter what value of N is set, the performance of *DDUB* is almost unchanged. The reason is that with a big data set, the local density of a given point will almost be equal to $\frac{1}{N}$ of its global density. No matter that $N = 2$ or $N = 16$, the input data set is big enough

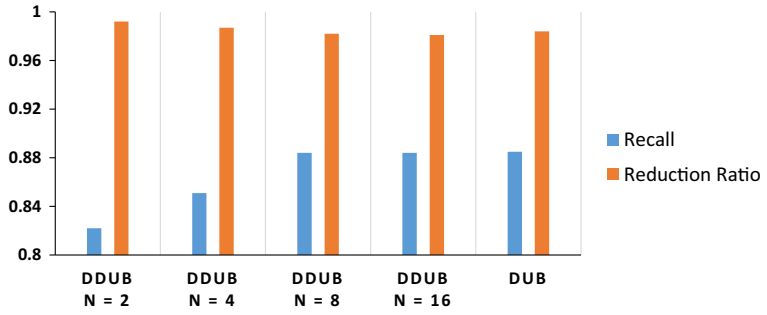

Fig. 8 Recall and reduction ratio: Goods

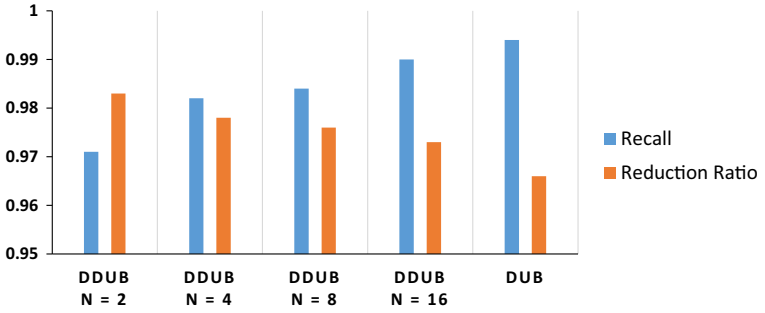

Fig. 9 Recall and reduction ratio: Small Citation

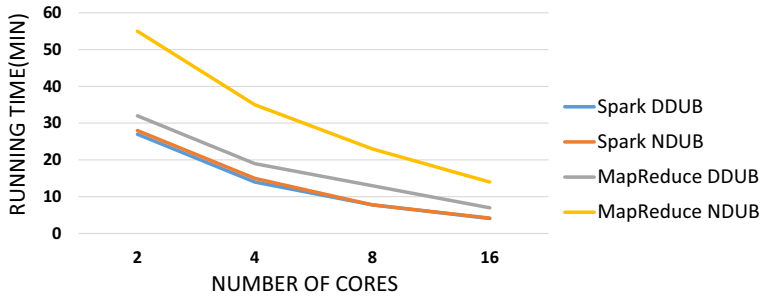

Fig. 10 Spark versus MapReduce: Small Citation

to sample a fair data partition, whose size is close to $\frac{1}{N}$ of the size of entire data set. The searching process with a threshold $\frac{T}{N}$ on a data partition is almost the same as searching the boundary of T on the entire data set.

5.4 Blocking efficiency

In the above, the experimental results show that *DDUB* has the ability to achieve a high *Recall* and *Recall Reduction*. Compared to the blocking performance of *DUB*, its loss of accuracy is rather small. Next, we will evaluate the efficiency of two parallised algorithms: *DDUB* and *NDUB*. Note that, in the following figures, we do not continue experiments that run longer than 4000 min, and the curves stretching out of the figure frame imply that the running time is too long to wait for us.

As widely known, Spark is a memory-based distributed framework, which stores the processed data in memory; MapReduce is a disk-based distributed framework, which stores the data in disk. In our experiment, we implement both *DDUB* and *NDUB* on Spark and MapReduce.

In MapReduce framework, when computing the global density of one point, *NDUB* first maps the data into N nodes and then waits and collects all the local densities from each node. This process includes one map task and one reduce task. In the entire workflow of *NDUB*, it has many rounds of computing density incurring many

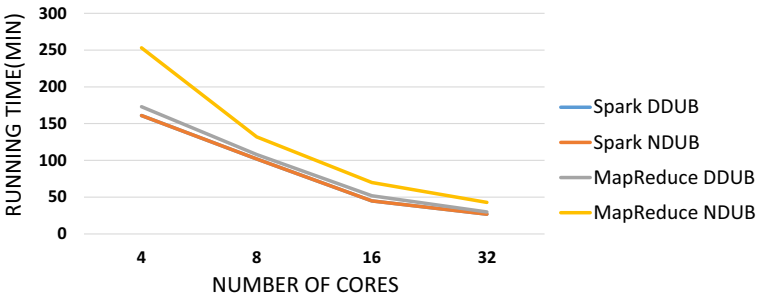

Fig. 11 Spark and MapReduce: Middle Citation

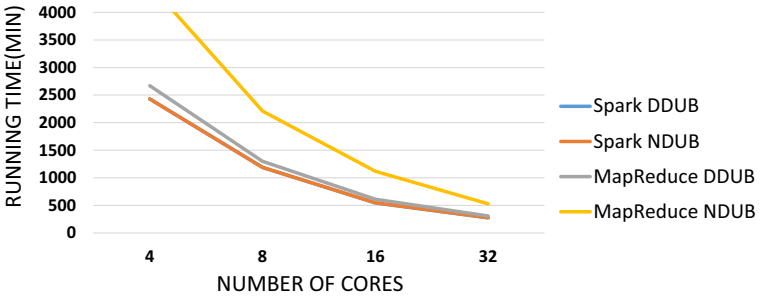

Fig. 12 Spark versus MapReduce: Large Citation

tasks of map and reduce. As each map or reduce task reads and writes in the disk, *NDUB* will spend a much long time on I/O operations, which is the reason that *NDUB* on MapReduce performs worst in Figs. 10, 11 and 12. However, *DDUB* is a one-pass distributed algorithm that runs the process of density computation locally and respectively. It has only one map and reduce task, which saves a bulk of I/O time compared to *NDUB*. From Figs. 10, 11 and 12, with the I/O time significantly reduced, we can see that *DDUB* on MapReduce almost runs the same time with *DDUB* on Spark. The result shows the superiority of our one-pass algorithm: *DDUB* on MapReduce performs as well as *DDUB* on Spark, while *NDUB* can only work on Spark.

In the above experiments, we can observe that the difference between *NDUB* and *DDUB* in Spark is not very clear. It is because that all the above experiments are conducted on a local network. The real RTT time between inter-cluster nodes is rather trivial and about 0.4 ms. While the network communication occurs every round in *NDUB* and only one round in *DDUB*, 1000 rounds can only produce a 4s difference which is very small compared to the whole running time. But, in real-world systems, cloud clusters may be geographically distributed and the RTT time is far more than 0.4 ms. To further study the performance of *NDUB* and *DDUB* in Spark, we will test the algorithms with different network conditions in the following experiments.

In order to evaluate the performance of *DDUB* and *NDUB* on geographically distributed large-scale system, we increase the latency of our network to simulate the latency in the real-world network. From our investigation, regularly, the RTT time between nodes in the same local network is ranged from 0 to 10 ms; the RTT time

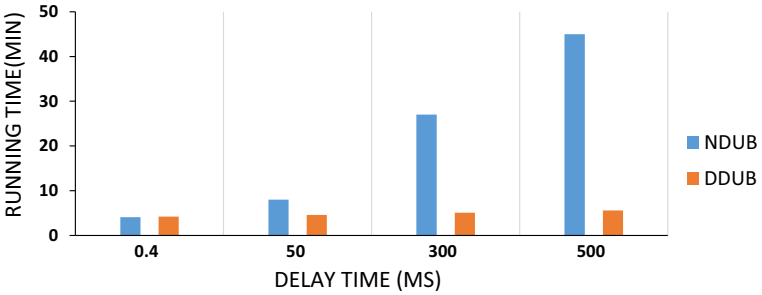

Fig. 13 Network delay: Small Citation

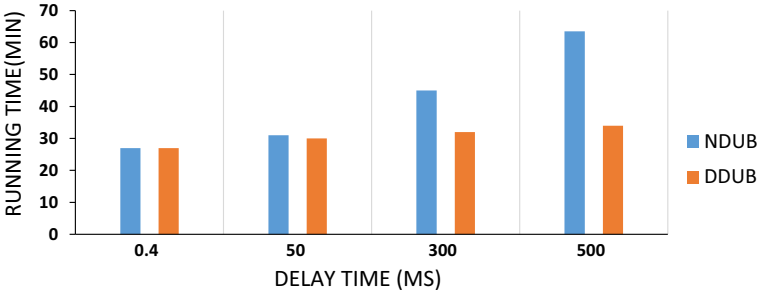

Fig. 14 Network delay: Middle Citation

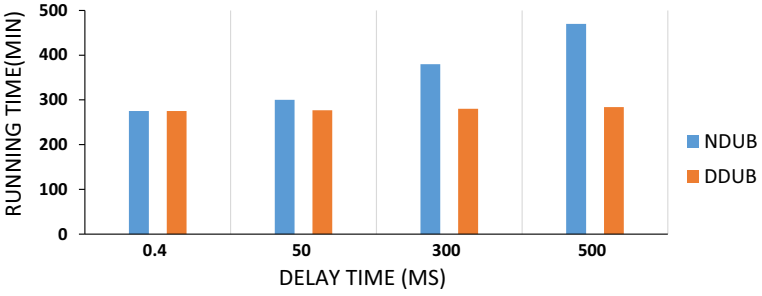

Fig. 15 Network delay: Large Citation

between nodes located across nation is ranged from 10 to 100 ms; and the RTT time between nodes located across continent is ranged from 100 to 500 ms. Hence, in our experiments, we add the RTT delay time in the range from 0 to 500 ms. We test the algorithms with the delay time: 0.4, 50, 300 and 500 ms. 0.4 ms is the minimal RTT time that we can set. For *Small Citation* set, 16 cores are used to run the algorithms and for the other two sets, 32 cores are used. The results are shown in Figs. 13, 14 and 15. When the delay time is set to a very small number, it implies that both *DDUB* and *NDUB* run in a local network, where the communication between nodes is almost free. *DDUB* and *NDUB* both work well on Spark. However, when the delay time increases to 50 ms, *DDUB* shows a higher efficiency than *NDUB*. We can observe that the running time of *DDUB* does not change much but the running time of *NDUB* greatly rises

up. The reason is that in *DDUB*, each node only runs on its own local data partition and rarely communicates with other nodes, leading to a low communication time; in *NDUB*, at each round of computing density, all nodes communicate for summing up the global density, leading to an accumulated big communication time finally. When the delay time reaches 500 ms, the running time of *DDUB* only increases a small amount, but the running time of *NDUB* reaches an unacceptable level.

Overall, for *Recall* and *Reduction Ratio*, our proposed algorithm *DDUB* has comparable performance with the other algorithms. For adaptability, *DDUB* can be well applied on both Spark and MapReduce platforms. For efficiency, *DDUB* undoubtedly wins in two aspects, read/write operation savings and communication time reduced. For scalability, our algorithm is a clear winner, as its running time grows linearly with the size of data.

6 Conclusion

Record matching is important in many applications of data mining, and blocking is a key for data reduction in recording matching. To adapt record matching to big data, it is inevitable to enhance record blocking. In this paper, we studied the problem of blocking records and parallelised a density-based blocking algorithm. Compared to the traditional blocking techniques, our proposed method has a small loss of recall and reduction ratio but good efficiency gain. We compared our density-based approach with other methods using real data sets and demonstrated the superiority of our approach in terms of *Reduction Ratio*, *Recall* and *Running Time*. The advantage of our approach is outstanding in the case of high cost of communication and data transferring. The results of the experiments have shown that the performance improvement of our approach is promising.

References

1. Elmagarmid AK, Ipeirotis PG, Verykios VS (2007) Duplicate record detection: a survey. IEEE Trans Knowl Data Eng 19(1):1
2. Bilenko M, Kamath B, Mooney RJ (2006) Adaptive blocking: learning to scale up record linkage. In: ICDM'06 Sixth International Conference on Data Mining (IEEE, 2006), pp 87–96
3. Michelson M, Knoblock CA (2006) Learning blocking schemes for record linkage. In: Proceedings of the National Conference on Artificial Intelligence, vol 21. (Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006), p 440
4. Whang SE, Menestrina D, Koutrika G, Theobald M, Garcia-Molina H (2009) Entity resolution with iterative blocking. In: SIGMOD Conference, pp 219–232
5. Dou C, Sun D, Wong R (2016) Unsupervised blocking of imbalanced datasets for record matching. In: International Conference on Web Information Systems Engineering. Springer, Berlin
6. Dou C, Sun D, Chen YC, Li G, Liu J (2016) Probabilistic parallelisation of blocking non-matched records for big data. In: 2016 IEEE International Conference on Big Data (Big Data), pp 3465–3473. doi:10.1109/BigData.2016.7841009
7. Chaudhuri S, Chen BC, Ganti V, Kaushik R (2007) Example-driven design of efficient record matching queries. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB Endowment), pp 327–338
8. Bilenko M, Mooney RJ (2003) On evaluation and training-set construction for duplicate detection. In: Proceedings of the KDD-2003 Workshop on Data Cleaning, Record Linkage, and Object Consolidation, pp 7–12

9. Arasu A, Götz M, Kaushik R (2010) On active learning of record matching packages. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (ACM), pp 783–794
10. Sarawagi S, Bhamidipaty A (2002) Interactive deduplication using active learning. In: KDD. ACM, pp 269–278
11. Tong S, Koller D (2001) Support vector machine active learning with applications to text classification. J Mach Learn Res 2:45
12. Tejada S, Knoblock CA, Minton S (2002) Learning domain-independent string transformation weights for high accuracy object identification. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp 350–359
13. Newcombe HB (1988) Handbook of record linkage: methods for health and statistical studies, administration, and business. Oxford University Press, Oxford
14. Wang R, Sun D, Li G, Atif M, Nepal S, LogProv (2016) Logging events as provenance of big data analytics pipelines with trustworthiness. In: 2016 IEEE International Conference on Big Data (Big Data), pp 1402–1411. doi:10.1109/BigData.2016.7840748
15. Wu D, Zhu L, Xu X, Sakr S, Sun D, Lu Q (2016) Building pipelines for heterogeneous execution environments for big data processing. IEEE Softw 33(2):60. doi:10.1109/MS.2016.35
16. Akbudak K, Aykanat C (2017) Exploiting locality in sparse matrix–matrix multiplication on many-core architectures. IEEE Trans Parallel Distrib Syst PP(99):1–1. doi:10.1109/TPDS.2017.2656893
17. Kunfang S, Lu H (2016) Efficient querying distributed big-XML data using MapReduce. Int J Grid High Perform Comput 8(3):70
18. Zeng Q, Zhao M, Liu P, Yadav P, Calo S, Lobo J (2015) Enforcement of autonomous authorizations in collaborative distributed query evaluation. IEEE Trans Knowl Data Eng 27(4):979
19. Slagter K, Hsu CH, Chung YC (2015) An adaptive and memory efficient sampling mechanism for partitioning in MapReduce. Int J Parallel Program 43(3):489
20. Christen P, Churches T, Hegland M (2004) Febrl—a parallel open source data linkage system. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, Berlin, pp 638–647
21. Kim Hs, Lee D (2007) Parallel linkage. In: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management. ACM, pp 283–292
22. Efthymiou V, Stefanidis K, Christophides V (2015) Big data entity resolution: from highly to somehow similar entity descriptions in the Web. In: 2015 IEEE International Conference on Big Data (Big Data). IEEE, pp 401–410
23. Christen P (2012) A survey of indexing techniques for scalable record linkage and deduplication. IEEE Trans Knowl Data Eng 24(9):1537
24. Kolb L, Thor A, Rahm E (2012) Dedoop: efficient deduplication with Hadoop. Proc VLDB Endow 5(12):1878
25. Hernández MA, Stolfo SJ (1995) The merge/purge problem for large databases. In: ACM Sigmod Record, vol 24. ACM, pp 127–138
26. Kolb L, Thor A, Rahm E (2012) Multi-pass sorted neighborhood blocking with MapReduce. Comput Sci Res Dev 27(1):45
27. Efthymiou V, Papadakis G, Papastefanatos G, Stefanidis K, Palpanas T (2015) Parallel meta-blocking: realizing scalable entity resolution over large, heterogeneous data. In: 2015 IEEE International Conference on Big Data (Big Data). IEEE, pp 411–420
28. Papadakis G, Koutrika G, Palpanas T, Nejdl W (2014) Meta-blocking: taking entity resolutionto the next level. IEEE Trans Knowl Data Eng 26(8):1946
29. Wang L, Tao J, Ranjan R, Marten H, Streit A, Chen J, Chen D (2013) G-Hadoop: MapReduce across distributed data centers for data-intensive computing. Future Gener Comput Syst 29(3):739
30. Jayalath C, Stephen J, Eugster P (2014) From the cloud to the atmosphere: running mapreduce across data centers. IEEE Trans Comput 63(1):74
31. Luo Y, Plale B (2012) Hierarchical mapreduce programming model and scheduling algorithms. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012). IEEE Computer Society, pp 769–774
32. Shabeera T, Madhu Kumar S (2015) Optimising virtual machine allocation in MapReduce cloud for improved data locality. Int J Big Data Intell 2(1):2
33. Hsu CH, Slagter KD, Chung YC (2015) Locality and loading aware virtual machine mapping techniques for optimizing communications in mapreduce applications. Future Gener Comput Syst 53:43