Knowledge and Information Systems (2020) 62:359–400
https://doi.org/10.1007/s10115-019-01347-0

REGULAR PAPER

Exploiting block co-occurrence to control block sizes for entity resolution

Dimas Cassimiro Nascimento[1,2] · Carlos Eduardo Santos Pires[2] · Demetrio Gomes Mestre[2]

Received: 30 May 2018 / Revised: 23 February 2019 / Accepted: 27 February 2019 /
Published online: 26 March 2019

Abstract
The problem of identifying duplicated entities in a dataset has gained increasing importance during the last decades. Due to the large size of the datasets, this problem can be very costly to be solved due to its intrinsic quadratic complexity. Both researchers and practitioners have developed a variety of techniques aiming to speed up a solution to this problem. One of these techniques is called blocking, an indexing technique that splits the dataset into a set of blocks, such that each block contains entities that share a common property evaluated by a blocking key function. In order to improve the efficacy of the blocking technique, multiple blocking keys may be used, and thus, a set of blocking results is generated. In this paper, we investigate how to control the size of the blocks generated by the use of multiple blocking keys and maintain reasonable quality results, which is measured by the quality of the produced blocks. By controlling the size of the blocks, we can reduce the overall cost of solving an entity resolution problem and facilitate the execution of a variety of tasks (e.g., real-time and privacy-preserving entity resolution). For doing so, we propose many heuristics which exploit the co-occurrence of entities among the generated blocks for pruning, splitting and merging blocks. The experimental results we carry out using four datasets confirm the adequacy of the proposed heuristics for generating block sizes within a predefined range threshold as well as maintaining reasonable blocking quality results.

Keywords Deduplication · Entity resolution · Heuristics · Data quality

✉ Dimas Cassimiro Nascimento
dimascnf@gmail.com

Carlos Eduardo Santos Pires
cesp@dsc.ufcg.edu.br

Demetrio Gomes Mestre
demetriogm@gmail.com

1 Federal Rural University of Pernambuco, Garanhuns, Brazil

2 Federal University of Campina Grande, Campina Grande, Brazil

1 Introduction

During the last decades, the increasing size of the datasets has grown the complexity of data-related problems. One of the problems related to the quality of the data stored in datasets is the task of identifying duplicated entities, which is commonly called entity resolution (also known as record reconciliation, entity matching and deduplication [4]). Theoretically, if a dataset D contains n entities, then in order to identify duplicated entities, every entity has to be compared with the other $n-1$ entities in D. This approach leads to a quadratic complexity ($O(n^2)$), such that n is the size of the dataset.

In order to avoid the high number of comparisons generated by a naive solution for the entity resolution problem, one can employ an initial step, prior to the entity comparison step, denoted as indexing [4]. This step aims to reduce the number of comparisons by limiting the comparisons only between entities that share common properties. The most known state-of-the-art indexing techniques are sorted neighborhood [18] (and its adaptive variants [38]), canopy clustering [6] and blocking [4]. For instance, the blocking step consists in splitting the dataset into several blocks. In practice, this property is the result of applying a blocking key function (or a set of functions) that maps (or aggregates) one or more attribute[1] values of an entity into a single value. This mapping can be done by selecting a substring or the Soundex of an input attribute value. Therefore, entities that produce the same blocking value are grouped in the same block, and thus, we compare the pairs of entities that share at least one block. This approach may reduce significantly the amount of generated comparisons. For instance, if n entities are split evenly into b blocks, then $\frac{n}{2}(\frac{n}{b}-1)$ comparisons are generated. After this step, we can measure the quality of the blocking result by analyzing the amount of comparisons between entities that is decreased by the blocking approach as well as by evaluating the number of duplicated entities that share at least one block.

The blocking step also has disadvantages, since it may degrade the quality of the entity resolution result. This phenomenon happens each time the employed blocking key function(s) produces different blocking key values for a pair of duplicated entities, and thus, the duplicated entities are placed into different blocks. Clearly, the efficacy of the blocking phase is strongly related to the efficacy of the employed blocking key function(s). In practice, it is hard to choose a single blocking key function that is able to effectively reduce the number of comparisons generated for the comparison phase as well as maintain acceptable blocking results (i.e., the number of duplicated entities in the same block is maximized). This choice is especially challenging when the entities stored in the dataset present data quality problems such as inaccurate and missing data [1,2].

To overcome this difficulty, one can use multiple [4,23] blocking keys in order to index the dataset. Instead of using a single blocking key function, one can use multiple blocking key functions which are initially employed in order to tackle the blocking step, and thus, a separated blocking result (a set of blocks) is generated for each blocking key function. On the other hand, it is noteworthy that the more the blocking key functions are employed, the more the blocks are generated, which in turn degrades the blocking capability to reduce the number of comparisons to be executed in the comparison phase.

A possible approach to overcome this problem is to employ pruning algorithms in order to control the size of the generated blocks. By ensuring that the block sizes are within a predefined range threshold, we can guarantee that the pruned blocking results (i.e., output of the pruning algorithms) will produce an expected (configurable) number of comparisons between entities in the comparison phase. In addition, the ability to control the block sizes is

[1] Portions of the entity schema can also be exploited [26].

relevant in order to facilitate the execution of a variety of tasks. For instance, there are several privacy-preserving entity resolution techniques [33,35] that require minimum and maximum block sizes. In order to ensure k-anonymous privacy [34], each block must contain at least k entities, which reduces the vulnerability to frequency-based attacks [33]. Another example is related to privacy-preserving indexing approaches for multiparties, which can enhance the privacy and the quality of the privacy-preserving entity resolution results [30,31] by producing blocks within a specific lower and upper range.

In turn, the execution of real-time [29] entity resolution approaches may require that a (maximum) predefined number of comparisons between entities should be performed in order to meet time restrictions [25]. Similarly, the users may be interested in cleansing the data for a limited time, and thus, they want to make best possible use of it [28]. Other applications such as anti-terrorism systems (since the ER process needs to identify a suspect within a limited time to prevent his escaping) and the matching of company names and individuals in the context of stock market trading [37] also need to perform ER considering time constraints. In order to tackle these scenarios, the control of the number of generated comparisons can be achieved by limiting the size of the generated blocks.

Also, the parallelization [22] of the comparisons generated by an entity resolution task in a distributed environment is facilitated if the block sizes are within a predefined range threshold. This is because, by controlling the size of the blocks, we can avoid the load balancing problem [15] that can be caused by the processing of large blocks by the computing nodes in a distributed environment. Finally, by limiting the size of the blocks, the appliance of more complex classification techniques, which can produce high-quality results, but are hard to scale when datasets are large becomes feasible. For instance, the collective classification techniques [7,10,14,16,17] work by mapping the entities into a similarity graph (usually using a similarity join operation [21]) and then employing clustering techniques over the generated graph. These steps can be executed more efficiently if we can control the size of the blocks, since the similarity join and the clustering algorithms are employed independently for each generated block.

In [12], the authors propose an approach to control block sizes that iteratively applies blocking key functions in order to split large blocks. Thus, this approach is not suitable when the employed blocking key functions are all initially employed to generate the blocking result. In turn, the authors of [26] tackle the problem of pruning a dataset that is indexed by multiple blocking key functions, but they provide no control over the size of the generated blocks. To the best of our knowledge, there are no available algorithms to control block sizes (i.e., which ensure that the generated block sizes are within a predefined range threshold) in the context of multiple blocking keys which are all initially employed in order to index the dataset. To fill this gap, we offer mainly four contributions in this paper. First, we formalize the problem of pruning a set of blocking results in order to control the size of the blocks generated by multiple blocking key functions. Second, we propose a metaheuristic (*MCBS*) that specifies high-level steps to effectively solve the tackled problem. Third, we propose five heuristics that exploit the co-occurrence of entities among blocks in order to shrink, split, merge and exclude blocks, which are employed to control the block sizes and still maintain high-quality blocking results. Fourth, we discuss the efficacy and efficiency of different algorithms, generated by combining the *MCBS* metaheuristic and the proposed heuristics, as well as the influence of different parameter values (such as interval size and weighting schema) over these results, based on the results of an experimental evaluation using four different datasets.

Table 1 Sample dataset D_1. The eID attribute represents the entity id in D_1 and identical eID colors represent duplicated entities

eID	Title	Year	Venue	Author1
d1	A dataquality-aware cloud service based on machine learning	2015	Salamanca	Nascimento, D.C. et. al
d2	Cloud service using Machine Learning	2015	Spain	FILHO, D.C.
d3	Cloud service based on metaheuristic	2016	Spain	Nascimento Filho, D
d4	Cloud service based on provisioning algorithms		Salamanca	Filho, D
d5	Adaptive sorted neighborhood blocking		Salamanca	GOMES, Demetrio
d6	Sorted neighborhood blocking for entity matching with MapReduce	2015	Salamanca	G M, Demetrio
d7	Approach to detect subsumption relations between tags	2015	Salamanca	REGO, Alex
d8	A Supervised learning approach to detect subsumption relations	2015	Spain	R., Alex
d9	Building lexical-semantic resources	2016	Spain	G., Jose
d10	Lexical-semantic resources based on heterogeneous information	2015	Spain	GILDO, Jose
d11	Framework for screening	2016	Italy	Vinicius Rosa
d12	Framework for virtual screening	2016	Pisa	ROSA, Vinicius
d13	Nodule classification based on shape distributions	2016	Pisa	Fernandes, Valeria
d14	Lung nodule classification based on shape distributions		Italy	Fernandes, V.
d15	A Classification based on shape distributions	2016	Italy	Valria, P. M.
d16	A taxonomy of reliable request-response protocols	2015	Salamanca	Naghmeh Ivaki

2 Motivating example

In this section, we present a motivating example in order to illustrate three main concepts: (i) single blocking key indexing; and (ii) indexing using multiple blocking key functions; and (iii) quality of blocking results. In Table 1, we present the sample dataset D_1, which contains 16 entities that represent scientific publications. The eID attribute represents the entity id in D_1 and identical eID colors represent duplicated entities.

In order to illustrate that the blocking step may prevent the identification of duplicated entities by avoiding the comparison between true matches (duplicated entities) that are placed into different blocks, we present examples[2] of 4 different blocking key functions that are (separately) employed in order to index dataset D_1.

Example 1 (*Single blocking key indexing*) By indexing dataset $D1$ using a single blocking function $f_1 = F_1(Title)$ (first letter of the *Title* attribute value), we generate the following blocking result: $\{A\{d_1, d_5, d_7, d_8, d_{15}, d_{16}\}, C\{d_2, d_3, d_4\}, S\{d_6\}, B\{d_9\}, L\{d_{10}, d_{14}\}, F\{d_{11}, d_{12}\}, N\{d_{13}\}\}$. Note that this blocking result generates 20 comparisons between entities and decreases the quality of the entity resolution results by preventing the comparison

[2] For single blocking key indexing, we employ the following notation: $\{bk_1\{e_1, e_2, \ldots\}, bk_2\{e_3, e_4, \ldots\}, bk_3\{e_5, e_6, \ldots\}, \ldots\}$, such that each bk_i represents an unique blocking key value generated by the employed blocking key function.

between the following pairs of (duplicated) entities: $\{(d_1, d_2), (d_1, d_3), (d_1, d_4), (d_5, d_6), (d_9, d_{10}), (d_{13}, d_{14}), (d_{13}, d_{15}), (d_{14}, d_{15})\}$. Hence, eight true matches (duplicated entities) are missed.

In order to improve the efficacy of the blocking approach, we can employ multiple blocking keys to index D_1, as illustrated in Example[3] 2.

Example 2 (*Indexing using multiple blocking keys*) By indexing the D_1 using the set of blocking key functions $\mathcal{F} = \{F_1(Title)$ (first letter of the *Title* attribute value), $F_1(Author1)$ (first letter of the *Author 1* attribute value), $(Venue)$ (the *Venue* attribute value)$\}$, we generate the following blocking collection $\mathcal{B}_{D_1}$: $\{\{A\{d_1, d_5, d_7, d_8, d_{15}, d_{16}\}, C\{d_2, d_3, d_4\}\}, S\{d_6\}, B\{d_9\}, L\{d_{10}, d_{14}\}, F\{d_{11}, d_{12}\}, N\{d_{13}\}\}$, $\{N\{d_1, d_3, d_{16}\}, F\{d_2, d_4, d_{14}, d_{13}\}, G\{d_5, d_6, d_9, d_{10}\}, R\{d_7, d_8, d_{12}\}, V\{d_{11}, d_{15}\}\}$, $\{Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}, Spain\{d_2, d_3, d_8, d_9, d_{10}\}, Italy\{d_{11}, d_{14}, d_{15}\}, Pisa\{d_{12}, d_{13}\}\}\}$. This blocking result generates 69 comparisons between entities and only a single pair of duplicated entities (d_1, d_2) is missed.

Note that even though the approach based on multiple blocking key functions employed in Example 2 allows the identification of the great majority of duplicated entities in D_1, its application also increases the amount of generated comparisons when compared to the single blocking key approach illustrated in Example (1). As mentioned in Sect. 1, a promising approach that can be employed to improve even more the efficacy (i.e., reduce the number of comparisons generated by the blocking result) of the blocking technique is to execute pruning algorithms in order to control the block sizes generated by the employed blocking key function(s).

3 Notation

In this section, we present the notation adopted throughout the paper. Let $\mathcal{D} = \{e_1, e_2, \ldots, e_{|\mathcal{D}|}\}$ be a set of entities. By employing a single blocking key function f in order to index D, we generate a blocking result $B = \{b_1, b_2, \ldots, b_{|B|}\}$, s.t. $\forall e \in D(\exists b \in B(e \in b))$ and $\cup_{b \in B} b = \mathcal{D}$. The blocking result may generate disjoint blocks (i.e., $\cap_{b \in B} b = \emptyset$) or overlapping blocks (i.e., $\cap_{b \in B} b \neq \emptyset$), depending on the blocking approach that is employed. In turn, if a set $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ of blocking key functions is employed, then a blocking collection $\mathcal{B} = \{B_1, B_2, \ldots, B_{|\mathcal{B}|}\}$ is produced, such that $\forall B \in \mathcal{B}(\forall e \in D(\exists b \in B(e \in b)))$ and $\forall B \in \mathcal{B}(\cup_{b \in B} b = \mathcal{D})$.

Using the adopted notation, in Definition 1 we present the notion of aggregated cardinality.

Definition 1 (*Aggregated cardinality*) The number of comparisons (*aggregated cardinality*) generated by a single blocking result B is calculated as: $||B|| = \sum_{b \in B} \frac{|b|*(|b|-1)}{2}$. In turn, the number of comparisons generated by a blocking collection $\mathcal{B}$ is equal to $||\mathcal{B}|| = \sum_{B \in \mathcal{B}} \sum_{b \in B} \frac{|b|*(|b|-1)}{2}$.

Example 3 (*Aggregated cardinality*) The individual appliance of the blocking key functions $F_1(Title)$ (first letter of the *Title* attribute value), $F_1(Author1)$ (first letter of the *Author 1* attribute value) and $(Venue)$ (the *Venue* attribute value) produces $\frac{6*5}{2} + \frac{3*2}{2} + \frac{2*1}{2} + \frac{2*1}{2} = 20$, $\frac{7*6}{2} + \frac{6*5}{2} = 36$, $\frac{3*2}{2} + \frac{4*3}{2} + \frac{4*3}{2} + \frac{3*2}{2} + \frac{2*1}{2} = 19$ and $\frac{6*5}{2} + \frac{5*4}{2} + \frac{3*2}{2} + \frac{2*1}{2} = 29$

[3] For indexing using multiple blocking key functions, we employ the following notation: $\{\{bk_{i,1}\{e_1, e_2, \ldots\}, bk_{i,2}\{e_3, e_4, \ldots\}, bk_{i,3}\{e_5, e_6, \ldots\}, \ldots\}, \{bk_{j,1}\{e_1, e_2, \ldots\}, bk_{j,2}\{e_3, e_4, \ldots\}, bk_{j,3}\{e_5, e_6, \ldots\}, \ldots\}, \ldots\}$, s.t. each $bk_{l,m}$ is the *m-th* blocking key value generated by the *l-th* blocking key function.

comparisons between entities, respectively. In turn, the appliance of multiple blocking key functions using the set $\{F_1(Title)$ (first letter of the *Title* attribute value), $F_1(Author1)$ (first letter of the *Author 1* attribute value), $(Venue)$ (the *Venue* attribute value)$\}$ produces $\frac{6*5}{2} + \frac{3*2}{2} + \frac{2*1}{2} + \frac{2*1}{2} + \frac{3*2}{2} + \frac{4*3}{2} + \frac{4*3}{2} + \frac{3*2}{2} + \frac{2*1}{2} + \frac{6*5}{2} + \frac{5*4}{2} + \frac{3*2}{2} + \frac{2*1}{2} = 68$ comparisons between entities. In turn, the number of comparisons generated by the naive (Cartesian product) approach is equal to: $\frac{16*(16-1)}{2} = 120$.

3.1 Quality metrics

In this section, we present classical blocking quality metrics that are defined by state-of-the-art works [9,24,26].

Definition 2 (*Pair quality (PQ)*) Let $\mathcal{B}$ be a blocking collection. The pair quality metric, which is equivalent to precision in the information retrieval area, aims to calculate the percentage of correct duplicated entities among the blocks, as shown in Eq. (1).

$$PQ(\mathcal{B}) = \frac{|\mathcal{D}^{\mathcal{B}}|}{||\mathcal{B}||} \tag{1}$$

such that $\mathcal{D}^{\mathcal{B}}$ is the number of detectable matches (i.e., pairs of duplicate entities that share at least one block in $\mathcal{B}$). A PQ value ranges from [0,1] and the closer the value is to 1, the more precise are the blocking results.

Definition 3 (*Pairs completeness*) Let D be a dataset and $\mathcal{B}$ be its blocking collection. The pairs completeness metric, which is equivalent to recall in the information retrieval area, aims to calculate the ratio between the amount of correct identified duplicated entities by the blocking results and the total amount of duplicated entities, as shown in Eq. (2).

$$PC(\mathcal{B}) = \frac{|\mathcal{D}^{\mathcal{B}}|}{\#\text{duplicates in } \mathcal{D}} \tag{2}$$

such that $\mathcal{D}^{\mathcal{B}}$ is the number of detectable matches (i.e., pairs of duplicate entities that share at least one block in $\mathcal{B}$) and #duplicates is the number of duplicated pairs of entities in $\mathcal{D}$. A PC value ranges from [0,1] and the closer the value is to 1, the more complete are the results.

We can calculate the efficacy of the pruning of a blocking collection using the reduction ratio (RR) metric [24], as presented in Definition 4.

Definition 4 (*Reduction ratio*) The reduction ratio metric aims to measure the efficacy of a pruning operation between an input blocking collection $\mathcal{B}$ and a pruned blocking collection $\mathcal{B}'$. For doing so, the reduction ratio is calculated as: $RR(\mathcal{B}, \mathcal{B}') = 1 - \frac{||\mathcal{B}'||}{||\mathcal{B}||}$. A RR value ranges from [0,1] and the greater the result, the greater is the efficacy of the pruning.

3.2 Blocking collection pruning

In Definition 5, we present the goal of a pruning algorithm, which aims to reduce the aggregated cardinality of a blocking collection and still maintain acceptable blocking quality results (see Definitions 2 and 3).

Definition 5 (*Pruning of blocking collection*) Let $\mathbf{D}$ be a dataset and $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ be a set of blocking key functions which are initially applied in order to index the entities of $\mathbf{D}$

and generate a blocking collection $\mathcal{B}$. A pruning algorithm aims to create a pruned blocking collection $\mathcal{B}'$, such that $RR(\mathcal{B}', \mathcal{B})$, $PC(\mathcal{B}')$ and $PQ(\mathcal{B}')$ are maximized.

Note that Definition 5 does not specify any restriction related to the size of the blocks of the pruned blocking collection $\mathcal{B}'$. Next, we define the problem of pruning a blocking collection for controlling block sizes.

3.3 Problem definition

In Definition 6, we formalize the problem that is tackled in this paper, i.e., how to prune a blocking collection in order to control the block sizes generated by multiple blocking key functions.

Definition 6 (*Pruning of a blocking collection for controlling block sizes*) Let $\mathbf{D}$ be a dataset, S_{min} and S_{max} be two block size parameters and $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ be a set of blocking key functions which are initially applied in order to index the entities of $\mathbf{D}$ and generate a blocking collection $\mathcal{B}$. The problem consists in creating a pruned blocking collection $\mathcal{B}' = \{B_1, B_2, \ldots, B_{|\mathcal{B}'|}\}$, such that for each block $b \in \cup_{B \in \mathcal{B}'} B : S_{min} \leq |b| \leq S_{max}$ and $RR(\mathcal{B}', \mathcal{B})$, $PC(\mathcal{B}')$ and $PQ(\mathcal{B}')$ are maximized.

There are intrinsic conflicts associated with the problem presented in Definition (6). First note that the generation of large blocks in the pruned blocking collection tends to favor $PC(\mathcal{B}')$ result (i.e., recall), but may drastically decrease the generated $PQ(\mathcal{B}')$ (i.e., precision), since the aggregated cardinality $||\mathcal{B}'||$ will be significant. Conversely, the generation of small blocks in the pruned blocking collection tends to favor $PQ(\mathcal{B}')$ (since $||\mathcal{B}'||$ is diminished), but it may also decrease $PC(\mathcal{B}')$ if the number of detectable duplicates ($|D^{\mathcal{B}}|$) is significantly reduced by the block pruning. For these reasons, by manipulating the input blocking collection in order to improve $RR(\mathcal{B}', \mathcal{B})$, a pruning algorithm may decrease both $PC(\mathcal{B}')$ and $PQ(\mathcal{B}')$, depending on the performed block manipulations. This phenomenon may happen when (at least) a pair of duplicate entities which share at least one block in the input blocking collection does not share any block in the pruned blocking collection.

Therefore, the main challenge of the problem presented in Definition (6) relies on manipulating the input blocks in order to control block sizes (to facilitate the execution of a variety of tasks, as explained in Sect. 1) and still maintain reasonable quality results (i.e., the overall amount of generated comparisons is minimized, and the number of comparisons between true matches is maximized). Since the metrics that need to be maximized represent inherently conflicting goals, a reasonable solution for this problem is expected to prune the input blocking collection, taking into account the block size restrictions, in order to present a good balance between $RR(\mathcal{B}', \mathcal{B})$ and $PC(\mathcal{B}')$.

Another obvious objective that is not explicitly mentioned in Definition 6 is to minimize the execution time required to prune the input blocking collection $\mathcal{B}$. Since the indexing phase is part of a broader entity resolution process (which involves standardization, indexing, entity comparison, classification and clerical review [4]), then by minimizing the time spent in the indexing phase we are also reducing the overall time required to solve an entity resolution problem.

4 Proposed approach

In this section, we present our approach to tackle the investigated problem (Definition 6). For doing so, we present the adopted strategies (Sects. 4.1–4.3) that are exploited in this paper. The overview of the proposed approach is presented in Sect. 4.4. In the following, in Sect. 4.5 we propose a metaheuristic, named metaheuristic for controlling block sizes (MCBS), which intends to prune an input blocking collection indexed by multiple blocking keys in order to generate blocks whose sizes are within a predefined range limit and still maintain acceptable quality results. Finally, we propose five heuristics (Sect. 4.6) in order to tackle the *MCBS* steps.

4.1 Exploiting block co-occurrence

Since the investigated problem (Definition 6) receives as an input a blocking collection that is initially indexed by multiple blocking keys, in this paper we propose heuristics that exploit the co-occurrence [26] of entities among blocks. The key idea behind this intuition is that the more the blocks are shared between two entities, the greater is the probability of these entities being classified as true matches.

For instance, if the entities e_1 and e_2 are exact matches (i.e., e_1 and e_2 have the same attribute values), then for any set of $|\mathcal{F}|$ blocking key functions that are employed to index these entities, e_1 and e_2 will co-occur in $|\mathcal{F}|$ blocks (considering a disjoint blocking approach). In turn, if e_2 and e_3 are inexact matches (i.e., e_2 and e_3 have similar attribute values), then the more the blocking key functions are employed to index these entities, the more the blocks e_2 and e_3 will tend to share. Finally, if e_3 and e_4 are quite dissimilar, then if $|\mathcal{F}|$ blocking key functions are employed to index these entities, e_3 and e_4 will tend to co-occur in a number $a \ll |\mathcal{F}|$ of blocks.

4.2 Weighting scheme

In [26], the authors propose five strategies to associate a weight for calculating co-occurrence of entities among blocks, denoted as weighting schemes (WS). These weighting schemes aim to explore the following intuition: the larger is the block size, the lower should be the weight associated with the co-occurrence of two entities in this block. In this paper, we explore two WS definitions[4]: ARCS and ECBS. We restate these definitions [26] as follows.

Definition 7 (*Aggregated reciprocal common scheme (ARCS)*) The ARCS score associated with a pair of entities (e_1, e_2) is defined as the sum of the reciprocal individual cardinalities of their common blocks. Formally, $ARCS(e_1, e_2, \mathcal{B}) = \sum_{b \in B_{e_1,e_2}} \frac{1}{||b||}$, s.t. $\forall b \in B_{e_1,e_2} : e_1 \in b \wedge e_2 \in b$.

Definition 8 (*Enhanced common blocking scheme (ECBS)*) The ECBS score associated with a pair (e_1, e_2) of entities takes into account the total number of blocks that are associated with each one of the co-occurring entities. More formally, $ECBS(e_1, e_2, \mathcal{B}) = |B_{e_1,e_2}|.log\frac{\#\mathcal{B}}{|B_{e_1}|}.log\frac{\#\mathcal{B}}{|B_{e_2}|}$, s.t. $\forall b \in B_{e_1,e_2} : e_1 \in b \wedge e_2 \in b$, $\forall b \in B_{e'} : e' \in b$ and $\#\mathcal{B} = \sum_{B \in \mathcal{B}} |B|$.

[4] These schemes have produced encouraging experimental results in [26].

4.3 Aggregated co-occurrence scores

In this section, we propose two metrics that aim to aggregate co-occurrence weights (co-occurrence score) of an entity in a block (Definition 9) and the average co-occurrence score of a block (Definition 10).

Definition 9 (*Entity co-occurrence score mean*) We calculate the entity co-occurrence score mean of an entity e_1 for a single block b as the average co-occurrence weight of all pairs (e_1, e_2), s.t. $e_2 \in b \backslash e_1$, as shown in Eq. (3).

$$EntityCooScore(e_1, b, \mathcal{B}, WS) = \sum_{e_2 \in b \backslash e_1} \frac{WS(e_1, e_2)}{|b| - 1} \tag{3}$$

s.t. WS is the weighting scheme employed to calculate the co-occurrence weight between e_1 and e_2.

We can employ weighting schemes in order to aggregate the co-occurrence scores between entities aiming to calculate the co-occurrence score mean associated with a block. We formalize this idea in Definition 10.

Definition 10 (*Block co-occurrence score mean*) We calculate the block co-occurrence score mean for a single block b as the average co-occurrence score of all entities in b, as shown in Eq. (4).

$$BlockCooScore(b, \mathcal{B}, WS) = \frac{\sum_{e \in b} EntityCooScore(e, b, \mathcal{B}, WS)}{|b|} \tag{4}$$

s.t. WS is the weighting scheme employed to calculate the co-occurrence weight between the entities in b.

Therefore, using Definition 10, we tackle the investigated problem by employing heuristics that aim to accomplish two objectives: control the pruned block sizes and maximize the block co-occurrence score mean of the pruned blocks.

4.4 Approach overview

The overview[5] of the proposed approach is presented in Fig. 1. The approach receives as input parameter a blocking collection, which is generated by an initial indexing step that employs multiple blocking key functions to index the dataset. Then, the blocks of the input blocking collection are shrunk, i.e., we remove entities from blocks that would probably generate superfluous comparisons (i.e., comparisons between non-matching entities). Then, we split large blocks ($|b| > S_{max}$) in order to ensure a maximum block size (S_{max}). After that, we merge small blocks ($|b| < S_{min}$) to guarantee a minimum block size (S_{min}). Finally, we exclude blocks that would probably produce a large number of superfluous comparisons between entities.

The *shrink*, *split*, *merge* and *exclude* operations use the parameters WS and λ, which represent the employed weighting scheme and the reach of pruning heuristics, respectively. In practice, λ may be used to calculate the number of entities to be excluded from a block, the number of blocks from which an entity will be removed or the number of blocks to be

[5] The *merge* operations may be performed using blocks from different blocking results (although this possibility is not shown in Fig. 1).

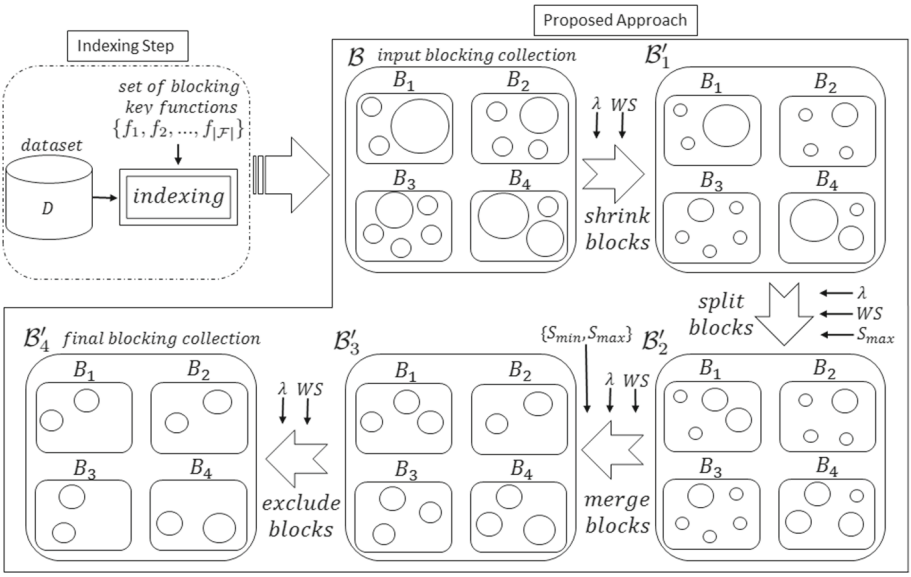


Fig. 1 Overview of the proposed approach

excluded from the blocking collection. Therefore, the semantics of the λ parameter depends on the employed heuristic.

The specific sequence of the steps (*shrink*, *merge*, *split* and *exclude*) presented in the proposed approach has been chosen for three main reasons. First, by shrinking the blocks as an initial step of the approach, the *merge*, *split* and *exclude* steps are performed more efficiently, since the algorithms will process smaller blocks. Second, the approach splits the blocks before performing the *merge* step for two main reasons: (i) it is easier to find candidate pairs of blocks to be merged when the blocks are small, since there are scenarios in which the algorithm may decide to merge a specific pair of blocks, but is prevented due to the maximum block size restriction (S_{max}) and (ii) the merge step will be executed more efficiently if the blocks are small ($\leq S_{max}$). Third, if the proposed approach employs co-occurrence scores in order to guide the processing performed by its first three steps, then after the pruning of the input blocking collection that is performed by the *shrink*, *merge* and *split* steps, the produced blocks will likely group entities that present high co-occurrence scores in the same blocks. This result entails that the evaluation of the mean co-occurrence scores associated with the produced blocks will be more accurate, and thus, it becomes more evident to choose which blocks should be excluded.

Another line of reasoning that can be employed to justify the specific order of the steps presented in the proposed approach is to analyze the undesired effects that are generated if we change the order of the approach steps. If we change the order of the *shrink* step with any other approach step, then the later step will have to process more entities among the blocks, and thus, it will become more complex and inefficient. In turn, if we change the order of the steps *split* and *merge*, then (i) it will be more difficult to find promising pairs of blocks to be merged; (ii) large blocks ($\geq S_{max}$) will not be eligible to be selected for merge operations; and (iii) if a merge algorithm decides to merge the blocks disregarding the maximum size restriction (S_{max}), then the *split* step will have to process more blocks to be split. Finally, if the *exclude* step is executed earlier in the approach workflow, then an algorithm for excluding

blocks will have to analyze low-quality blocks (i.e., blocks that have not been pruned yet) and, consequently, it will be more difficult to choose the blocks to be excluded without sacrificing quality results.

4.5 MCBS metaheuristic

The *MCBS* metaheuristic is used to structure the high-level steps of the proposed approach (Fig. 1) and is presented in Algorithm 1. The algorithm is composed by four steps that can be replaced by specific implementations of heuristics in order to perform the following tasks: (i) *ShrinkBlocks* (line 3), which intends to remove entities from blocks; (ii) *SplitBlocks* (line 5), which aims to split blocks in order to ensure that all block sizes are less than or equal to S_{max}; (iii) *MergeBlocks* (line 7), which intends to merge blocks in order to ensure that all block sizes are greater than or equal to S_{min}; and (iv) *ExcludeBlocks* (line 9), which aims to exclude blocks from the input blocking collection.

Algorithm 1: $MCBS$ Metaheuristic

Input : $\mathcal{B}$: input blocking collection
S_{max}: maximum block size
S_{min}: minimum block size
WS: a weighting scheme
λ: a percentage (0,1) that determines the reach of pruning heuristics
Output: $\mathcal{B}'$: a pruned blocking containing block sizes between S_{min} and S_{max}

```
1  begin
2      // remove entities from blocks
3      B' ← ShrinkBlocks(B, WS, λ)
4      // return blocks with a maximum block size
5      B' ← SplitBlocks(B', WS, S_max)
6      // return blocks with a minimum block size
7      B' ← MergeBlocks(B', WS, S_min, S_max)
8      // exclude blocks from the blocking collection
9      B' ← ExcludeBlocks(B', WS, λ)
10     return B'
```

4.6 Heuristics for shrinking blocks

The aim of the heuristics that are proposed in this section is twofold. First, the more we reduce the amount of entities in the pruned blocks (i.e., $\sum_{B\in\mathcal{B}'}\sum_{b\in B}|b|$), the more we are increasing the reduction ratio between the input and pruned blocking collection, i.e., $RR(\mathcal{B},\mathcal{B}')$. Second, by reducing $\sum_{B\in\mathcal{B}'}\sum_{b\in B}|b|$, we are also reducing the time required to execute the subsequent steps of the *MCBS* metaheuristic.

For doing so, the main challenge relies on removing entities from the blocks without compromising significantly the quality results (PQ and PC) associated with the pruned blocking collection result. In order to tackle this challenge, we propose three heuristics that aim to exploit entity co-occurrence scores (Definition 9) and block co-occurrence scores (Definition 10) in order to select entities to be removed from blocks. The three heuristics receive as input parameters a blocking collection $\mathcal{B}$, a weighting scheme WS and a percentage λ. The output of the heuristics is a pruned blocking collection $\mathcal{B}'$.

4.6.1 Low entity co-occurrence pruning (lECP)

The *low entity co-occurrence pruning* heuristic aims to prune the blocks by removing each entity e from the blocks in which it is less likely to find a true match between e and the other entities. This likelihood is analyzed based on the co-occurrence score of the entities in the input blocking collection. For each entity block b, the algorithm evaluates the co-occurrence score between each entity $e \in b$ and all the other entities in $b \backslash e$. After that, the *lECP* algorithm removes the entities from b that present the lowest mean co-occurrence score values (Definition 9).

The pseudocode of the *lECP* heuristic is presented in Algorithm 2. For each block b in the blocking collection (lines 2-3), the algorithm performs the following tasks: (i) it creates an empty priority queue ordered by entity co-occurrence score (line 5); (ii) it pushes every entity e in b to the queue (lines 7-9); and (iii) it removes the $\lfloor |b| * \lambda \rfloor$ entities from b that present the lowest co-occurrence score values (lines 11–12). Finally, the pruned blocking collection is returned[6] in line 13. We present an illustration of this pruning approach in Example 4. In order to illustrate the functioning of the proposed algorithms, we employ a weighting scheme that simply counts the amount of blocks shared by two entities (denoted by *common blocking scheme—CBS* [26]), i.e., $CBS(e_1, e_2) = |B_{e_1,e_2}|$, s.t. $\forall b \in B_{e_1,e_2} : e_1 \in b \wedge e_2 \in b$.

Algorithm 2: *lECP*

```
   Input  : B: input blocking collection
            WS: a weighting scheme
            λ: a percentage (0,1) that determines the reach of the pruning
   Output: a pruned blocking collection
1  begin
2      foreach B ∈ B do
3          foreach b ∈ B do
4              // create an empty priority queue ordered by entity
                  co-occurrence score
5              Q ← GeneratePriorityQueue()
6              // push every entity e in b into the queue
7              foreach e ∈ b do
8                  // using Definition 9
9                  Q.push(e, EntityCooScore(e, b, B, WS))
10             limit ← ⌊|b| * λ⌋
11             for i = 1 to limit do
12                 b ← b\Q.pop()
13     return B
```

Example 4 (*Low entity co-occurrence pruning (lECP)*) Considering the parameters and calculations presented in Table 2, the block $b = R\{d_7, d_8, d_{12}\}$ is pruned by the *lECP* algorithm as follows: (i) the priority queue $Q = \{(d_7, \frac{2+1}{2}), (d_8, \frac{2+1}{2}), (d_{12}, \frac{1+1}{2})\}$ (see Definition 9) is generated (lines 5–9); (ii) the $limit$ is calculated as $\lfloor |b| * \lambda \rfloor = \lfloor 3 * 0.4 \rfloor = 1$ (line 10); and (iii) thus, a single entity ($Q.pop() = d_{12}$) is excluded from b (lines 11–12). Note that this pruning operation reduces the aggregated cardinality result from 69 to 67 comparisons between entities and it does not decrease the PC result of the pruned blocking collection.

[6] We assume that the changes in the blocks performed by the algorithm are reflected in the input blocking collection $\mathcal{B}$. We use the same assumption for Algorithms 3–7.

Table 2 Parameters and calculations used by *lECP* heuristic in Example 4

Blocking collection	$\mathcal{B}_{D_1}$ (Example 2)
Input block	$b = R\{d_7, d_8, d_{12}\}$
lECP parameters	$WS = CBS, \lambda = 0.4$
Calculations	$CBS(d_7, d_8) = 2, CBS(d_7, d_{12}) = 1, CBS(d_8, d_{12}) = 1$

4.6.2 Low block co-occurrence pruning (lBCP)

Different from the *lECP* algorithm, which evaluates the entities that should be removed from each block, the *low block co-occurrence pruning* heuristic aims to determine a set of blocks from which each entity should be removed. In other words, given an entity e, the algorithm tries to identify which blocks are less likely to present duplicates of e. To this end, the *lECP* algorithm evaluates the co-occurrence score of an entity e in every block b, such that $e \in b$. Afterward, e is removed from the blocks in which it presents the lowest mean co-occurrence scores.

The pseudocode of the *lBCP* heuristic is presented in Algorithm 3. For each entity e in the blocking collection $\mathcal{B}$ (line 2), the algorithm performs the following tasks: (i) it creates an empty priority queue ordered by entity co-occurrence score (line 4); (ii) it creates a set ($\mathcal{B}_e$) containing every block b in the blocking collection, such that $e \in b$ (line 6); (iii) it pushes every block b of $\mathcal{B}_e$ to the queue (lines 8–10); and (iv) it removes e from the $|\mathcal{B}_e| * \lambda$ blocks in which e presents the lowest co-occurrence score values (lines 12–14). Finally, the pruned blocking collection is returned in line 15. We present an illustration of this pruning approach in Example 5.

Algorithm 3: *lBCP*

```
   Input  : B: input blocking collection
            WS: a weighting scheme
            λ: a percentage (0,1) that determines the reach of the pruning
   Output: a pruned blocking collection
1  begin
2      foreach e in the blocking collection B do
3          // create an empty priority queue ordered by entity
               co-occurrence score
4          Q ← GeneratePriorityQueue()
5          // select the blocks of entity e
6          B_e ← select every block b, s.t. e ∈ b
7          // push every block of e into the queue
8          foreach b ∈ B_e do
9              // using Definition 9
10             Q.push(b, EntityCooScore(e, b, B, WS))
11         limit ← ⌊|B_e| * λ⌋
12         for i = 1 to limit do
13             b ← Q.pop()
14             b ← b\e
15     return B
```

Table 3 Parameters and calculations used by *lBCP* heuristic in Example 5

Blocking collection	$\mathcal{B}_{D_1}$ (Example 2)
lBCP parameters	$WS = CBS, \lambda = 0.4$
Input entity	d_{10}
Calculations	$CBS(d_2, d_{10}) = 1, CBS(d_3, d_{10}) = 1, CBS(d_5, d_{10}) = 1, CBS(d_6, d_{10}) = 1, CBS(d_8, d_{10}) = 1, CBS(d_9, d_{10}) = 2, CBS(d_{14}, d_{10}) = 1$

Example 5 (*Low block co-occurrence pruning (lBCP)*) Considering the parameters and calculations presented in Table 3, the blocks of entity d_{10} are pruned by the *lBCP* algorithm as follows: (i) the priority queue $Q = \{(G\{d_5, d_6, d_9, d_{10}\}, \frac{4}{3}), (Spain\{d_2, d_3, d_8, d_9, d_{10}\}, \frac{5}{4}), (L\{d_{10}, d_{14}\}, \frac{1}{1})\}$ (see Definition 9) is generated (lines 4–10); (ii) the $limit$ is calculated as $\lfloor |\mathcal{B}_{d_{10}}| * \lambda \rfloor = \lfloor 3 * 0.4 \rfloor = 1$ (line 11); (iii) thus, entity d_{10} is removed from block $L\{d_{10}, d_{14}\}$ (lines 12–14). Note that this pruning operation reduces the aggregated cardinality result from 69 to 68 comparisons between entities and it does not decrease the PC result of the pruned blocking collection.

4.6.3 Large block size pruning (LBSP)

The *large block size pruning* heuristic employs a reasoning similar to the *lECP* algorithm in order to prune only the large blocks in the input blocking collection. The intuition behind this idea is based on three principles: (i) the larger the block size, the more attenuating tends to be the differences between the co-occurrence scores of the entities in the block, and thus, the pruning becomes more accurate (i.e., the pruning tends to remove entities from the blocks without sacrificing quality results); (ii) by restricting the shrink process over large blocks, we reduce the overall time required to perform the shrink step, and thus, we execute the proposed approach more efficiently; and (iii) the pruning of large blocks leads to a more significant impact over the reduction ratio result than the pruning of small blocks, and thus, it is not worthwhile to shrink small blocks at the cost of a possible reduction in PC.

The pseudocode of the *LBSP* heuristic is presented in Algorithm 4. First, the block size mean μ is calculated (line 2). Then, for each block b in the blocking collection $\mathcal{B}$, such that $|b| \geq \mu$, the algorithm performs the following tasks: (i) it creates an empty priority queue ordered by entity co-occurrence score (line 7) and (ii) it removes the $\lfloor |b| * \lambda \rfloor$ entities from b that present the lowest co-occurrence score values (lines 13–14). Lastly, the pruned blocking collection is returned in line 15. We present an illustration of this pruning approach in Example 6.

Example 6 (*Large block size pruning (LBSP)*) Considering the parameters and calculations presented in Table 4, the block $b = Spain\{d_2, d_3, d_8, d_9, d_{10}\}$ is pruned by the *LBSP* algorithm as follows: (i) the priority queue $Q = \{(d_2, \frac{5}{4}), (d_3, \frac{5}{4}), (d_9, \frac{5}{4}), (d_{10}, \frac{5}{4}), (d_8, \frac{4}{4})\}$ (see Definition 9) is generated (lines 7–11); (ii) the $limit$ is calculated as $\lfloor |b| * \lambda \rfloor = \lfloor 5 * 0.2 \rfloor = 1$ (line 12); and (iii) thus, a single entity ($Q.pop() = d_8$) is removed from b (lines 13–14). Note that this pruning operation reduces the aggregated cardinality result from 69 to 65 comparisons between entities and it does not decrease the PC result of the pruned blocking collection.

Algorithm 4: $LBSP$

Input : $\mathcal{B}$: an input blocking collection
WS: a weighting scheme
λ: a percentage (0,1) that determines the reach of the pruning
Output: a pruned blocking collection

```
1  begin
2      μ ← (∑_{B∈B} ∑_{b∈B} |b|) / ∑_{B∈B} |B|
3      foreach B ∈ B do
4          foreach b ∈ B do
5              if (|b| ≥ μ) then
6                  // create an empty priority queue ordered by entity
                      co-occurrence score
7                  Q ← GeneratePriorityQueue()
8                  // push every entity e in b into the queue
9                  foreach e ∈ b do
10                     // using Definition 9
11                     Q.push(b, EntityCooScore(e, b, B, WS))
12                 limit ← ⌊|b| * λ⌋
13                 for i = 1 to limit do
14                     b ← b\Q.pop()
15     return B
```

Table 4 Parameters and calculations used by *LBSP* heuristic in Example 6

Blocking collection	$\mathcal{B}_{D_1}$ (Example 2)
LBSP parameters	$WS = CBS$ and $\lambda = 0.2$
Input block	$b = Spain\{d_2, d_3, d_8, d_9, d_{10}\}$
Calculations	$\mu = \frac{49}{16} = 3.06$, $CBS(d_2, d_3) = 2$, $CBS(d_2, d_8) = 1$, $CBS(d_2, d_9) = 1$, $CBS(d_2, d_{10}) = 1$, $CBS(d_3, d_8) = 1$, $CBS(d_3, d_9) = 1$, $CBS(d_3, d_{10}) = 1$, $CBS(d_8, d_9) = 1$, $CBS(d_8, d_{10}) = 1$, $CBS(d_9, d_{10}) = 2$

4.7 Heuristic for splitting blocks

In this section, we propose a heuristic that aims to perform the second step of the proposed *MCBS* metaheuristic, i.e., to ensure a maximum block size for all blocks in the input blocking collection. The proposed heuristic (*CooSlicer*) splits large blocks ($|b| > S_{max}$) based on the values of the co-occurrence scores (Definition 9) of its entities. To this end, the subblocks are iteratively generated by grouping the entities that present the highest co-occurrence score values among the remaining entities in the block that is being split.

For each block b in the blocking collection, s.t. $|b| > S_{max}$, b is split by the *CooSlicer* algorithm into $\#blocks = \lceil \frac{|b|}{S_{max}} \rceil$ subblocks. To this end, $\lceil \frac{|b|}{\#blocks} \rceil$ entities that present the highest co-occurrence values are removed from b to generate a new subblock. This process is repeated with the remaining entities in b until $|b| \leq \lceil \frac{|b|}{\#blocks} \rceil$. After that, the remaining entities in b forms the last subblock. Note that each subblock is generated containing at most $\lceil \frac{|b|}{\#blocks} \rceil$ entities. Since $\lceil \frac{|b|}{\#blocks} \rceil <= S_{max}$, then it follows that each generated subblock honors the maximum block size restriction (S_{max}). Also note that, if a subblock b' is generated with less than S_{min} entities, then b' will be merged with other blocks in the next step of the proposed approach (*merge* step).

Algorithm 5: *CooSlicer*

Input : $\mathcal{B}$: an input blocking collection
S_{max}: maximum block size
WS: a weighting scheme
Output: a pruned blocking containing block sizes under (or equal to) S_{max}

```
1  begin
2      // select large blocks
3      B+ ← select every block b, s.t. |b| > Smax
4      foreach b ∈ B+ do
5          // Create an empty priority queue, ordered by entity
               co-occurrence score
6          Q ← GeneratePriorityQueue()
7          // push every entity in b into the queue
8          foreach e ∈ b do
9              // using Definition 9
10             Q.push(e, EnittyCooScore(e, b, WS, B))
11         #blocks ← ⌈|b| / Smax⌉
12         bSize ← ⌈|b| / #blocks⌉
13         Bb ← B ∈ B, s.t. b ∈ B
14         while (Q is not empty) do
15             b' ← ∅
16             for i = 1 to bSize do
17                 if (Q is not empty) then
18                     b' ← b' ∪ Q.pop()
19             Bb ← Bb ∪ b'
20         Bb ← Bb\b
21     return B
```

The pseudocode of the *CooSlicer* heuristic is presented in Algorithm 5. Initially, the algorithm creates the set B^+, which contains every block b in the blocking collection such that $|b| > S_{max}$ (line 3). Then, for every block b in B^+ (lines 4), the algorithm performs the following tasks: (i) it creates an empty priority queue, ordered by entity co-occurrence score (line 6); (ii) it pushes every entity e in b to the queue (lines 8–10); (iii) it calculates the amount of blocks ($\#blocks$) and the maximum size ($bSize$) of the blocks that will be generated by splitting the block b (lines 11–12); (iv) it creates $\#blocks$ subblocks containing (note that the last generated subblock may contain $c < bSize$ entities) at most $bSize$ entities (lines 14–19); and (v) it removes block b from the blocking collection. Lastly, the pruned blocking collection is returned in line 21. We present an illustration of this heuristic in Example 7.

Example 7 (*CooSlicer*) Considering the parameters and calculations presented in Table 5, the block $b = R\{d_7, d_8, d_{12}\}$ is split by the $CooSlicer$ algorithm as follows: (i) the priority queue $Q = \{(d_7, \frac{3}{2}), (d_8, \frac{3}{2}), (d_{12}, \frac{2}{2})\}$ (see Definition 9) is generated (lines 6–10); (ii) $\#blocks = \lceil \frac{|b|}{S_{max}} \rceil = \lceil \frac{3}{2} \rceil = 2$ (line 11), and thus, $bSize = \lceil \frac{|b|}{\#blocks} \rceil = \lceil \frac{3}{2} \rceil = 2$ (line 12); (iii) blocks $\{d_7, d_8\}$ and $\{d_{12}\}$ are generated and inserted into the blocking collection (lines 14–19); and (iv) lastly, block $R\{d_7, d_8, d_{12}\}$ is excluded from the blocking collection (line 20). Note that this split operation reduces the aggregated cardinality result from 69 to 67 comparisons between entities and it does not decrease the PC result of the pruned blocking collection.

Table 5 Parameters and calculations used by the *CooSlicer* heuristic in Example 7

Blocking collection	$\mathcal{B}_{D_1}$ (Example 2)
CooSlicer parameters	$WS = CBS$, $S_{max} = 2$
Input block	$b = R\{d_7, d_8, d_{12}\}$
Calculations	$CBS(d_7, d_8) = 2$, $CBS(d_7, d_{12}) = 1$, $CBS(d_8, d_{12}) = 1$

4.8 Heuristic for merging blocks

In this section, we propose a heuristic that aims to merge blocks from an input blocking collection, which corresponds to the third step of the proposed metaheuristic (*MCBS*). This task ensures a minimum block size for all blocks in the blocking collection. A possible approach to perform this step is to calculate a similarity measure between a block b in B^- (i.e., the set of blocks in the blocking collection, such that $|b| < S_{min}$) with every other block $b' \in B^- \backslash b$. In order to speed up this process, we employ an approach that is more efficient than calculating the co-occurrence score between all pairs of entities in $b \times b'$ (for all b in B^- and $b' \in B^- \backslash b$).

To this end, instead of calculating the co-occurrence between pairs of entities, we propose a heuristic that exploits the number of entities contained in the intersection between small blocks ($|b| < S_{min}$). The intuition behind this idea is that the more entities two blocks share, the greater is the probability of their intersection present duplicated entities which were not covered by the original blocks (i.e., the greater tends to be the pairs completeness of the resulting block when compared to the pairs completeness of the original blocks). Therefore, the proposed heuristic (*MaxIntersectionMerge*) aims to merge small blocks based on the size of their intersections.

The pseudocode of the *MaxIntersectionMerge* heuristic is presented in Algorithm 6. Initially, the algorithm creates the sets B^- and B^*, which contains every block b in the blocking collection such that $|b| < S_{min}$ and $S_{min} \leq |b| \leq S_{max}$, respectively (lines 3–5). Then, while the set B^- is not empty (line 6), the algorithm performs the following tasks: (i) the largest block (b_1) from B^- (note that, by prioritizing the largest blocks, it becomes easier to find pairs of blocks that meet the conditions expressed in line 9 when the further iterations of the loop are executed) is selected (line 7); (ii) block (b_2) that presents the maximum intersection with b_1 (and produces $|b_1 \cap b_2| \leq S_{max}$) is selected from B^- (if $|B^-| > 1$) or B^+ (if $|B^-| = 1$) (line 9 or line 11); (iii) blocks b_1 and b_2 are merged into block b' (line 13); (iv) b_1 and b_2 are removed from B^- (lines 15–16); and (v) b' is inserted into B^- (line 19) or B^* (line 21), depending on its size. Lastly, the pruned blocking collection is returned in line 22.

A specific case[7] may occur when none of the blocks in $B^- \backslash b_1$ (line 9) or B^* (line 11) are eligible to be merged with the largest block $b_1 \in B^-$ (line 7), i.e., $\forall b_2 \in B^- \backslash b_1 : (|b_1| + |b_2|) > S_{max}$ and $\forall b_2 \in B^* : (|b_1| + |b_2|) > S_{max}$. In this case, b_2 is selected as shown in line 11 (but removing the size restriction, i.e., $(|b_1| + |b_2|) > S_{max}$) and we prune the generated block b' (line 13) by removing the $|b'| - S_{max}$ entities from b' that present the lowest co-occurrence score values (using the reasoning employed by the *lECP* heuristic using $\lambda = \frac{|b'| - S_{max}}{|b'|}$) in order to ensure that $|b'| \leq S_{max}$. We present an illustration of the *MaxIntersectionMerge* heuristic in Example 8.

[7] This special case is not shown in the *MaxIntersectionMerge* algorithm in order to simplify its representation.

Algorithm 6: *MaxIntersectionMerge*

Input : $\mathcal{B}$: an input blocking collection
S_{min}: minimum block size
S_{max}: maximum block size
Output: a pruned blocking containing block sizes above (or equal to) S_{min}

1 **begin**
2 `// select small blocks`
3 $B^- \leftarrow$ select every block b, $s.t.\ |b| < S_{min}$
4 `// select blocks with correct size`
5 $B^* \leftarrow$ select every block b, $s.t.\ S_{min} \leq |b| \leq S_{max}$
6 **while** $(|B^-| > 0)$ **do**
7 $b_1 \leftarrow \arg\max_{b \in B^-} |b|$
8 **if** $(|B^-| > 1)$ **then**
9 $b_2 \leftarrow \arg\max_{b \in B^- \setminus b_1,\ s.t.(|b1|+|b|) \leq S_{max}} |b \cap b1|$
10 **else**
11 $b_2 \leftarrow \arg\max_{b \in B^*,\ s.t.(|b1|+|b|) \leq S_{max}} |b \cap b1|$
12 `// merge blocks` b_1 `and` b_2
13 $b' \leftarrow b_1 \cup b_2$
14 `// remove blocks` b_1 `and` b_2 `from the blocking collection`
15 $B^- \leftarrow B^- \setminus b1$
16 $B^- \leftarrow B^- \setminus b2$
17 `// insert` b' `into` B^- `or` B^*, `depending on its size`
18 **if** $(|b'| < S_{min})$ **then**
19 $B^- \leftarrow B^- \cup b'$
20 **else**
21 $B^* \leftarrow B^* \cup b'$
22 **return** $\mathcal{B}$

Example 8 (*MaxIntersectionMerge*) Considering the parameters presented in Table 6, then the block $b_1 = Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}$ (line 7) is merged by the *MaxIntersectionMerge* algorithm as follows: (i) the block $b_2 = N\{d_1, d_3, d_{16}\}$ is selected because it presents the largest intersection with the block $Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}$ and $|Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\} \cap N\{d_1, d_3, d_{16}\}| \leq 8$ (line 9); (ii) the block $b' = (N \cup Salamanca)\{d_1, d_3, d_4, d_5, d_6, d_7, d_{16}\}$ is created (line 13); (iii) the blocks $Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}$ and $N\{d_1, d_3, d_{16}\}$ are excluded from the blocking collection (lines 15–16); and (iv) finally, the block $(N \cup Salamanca)\{d_1, d_3, d_4, d_5, d_6, d_7, d_{16}\}$ is inserted into B^* because $7 \leq |(N \cup Salamanca)\{d_1, d_3, d_4, d_5, d_6, d_7, d_{16}\}| \leq 8$ (line 21). Note that this merge operation increases the aggregated cardinality result from 69 to 72 comparisons between entities. Also note that this merge operation could have enhanced the PC result of the pruned blocking collection by allowing the comparison between the following pair of duplicated entities: (d_3, d_4) (which is not detectable by the blocks $Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}$ and $N\{d_1, d_3, d_{16}\}$).

Table 6 Parameters used by the *MaxInterssectionMerge* heuristic in Example 8

Blocking collection	$\mathcal{B}_{D_1}$ (Example 2)
MaxInterssectionMerge parameters	$WS = CBS$, $S\_\{min\} = 7$, $S_{max} = 8$
Input block	$b_1 = Salamanca\{d_1, d_4, d_5, d_6, d_7, d_{16}\}$

4.9 Heuristic for excluding blocks

In this section, we propose a heuristic that excludes blocks from an input blocking collection, which corresponds to the fourth step of the proposed metaheuristic (*MCBS*). The *low block co-occurrence excluder* (lBCE) heuristic aims to identify which blocks from the block collection present a low likelihood of generating comparisons between duplicated entities. This likelihood is inferred based on the block co-occurrence score (Definition 10) associated with the blocks. The higher is the block co-occurrence score mean associated with a block b, the higher is the likelihood that there is at least a pair $(e_1, e_2) \in b \times b$, such that e_1 and e_2 are duplicated entities. For this reason, the *lBCE* heuristic excludes the blocks that present the lowest block co-occurrence scores (Definition 10) from the blocking collection. The reach of the heuristic (i.e., the number of blocks to be excluded) is defined based on the value of the parameter λ. Thus, $\lfloor(\sum_{B\in\mathcal{B}}|B|) * \lambda\rfloor$ blocks are excluded from the blocking collection.

The pseudocode of the *low block co-occurrence excluder* heuristic is presented in Algorithm 7. First, the algorithm creates an empty priority queue ordered by block co-occurrence score (line 2). Then, every block b in the blocking collection (lines 4–5) is pushed to the queue (line 7). After that, the *lBCE* algorithm excludes $\lfloor(\sum_{B\in\mathcal{B}}|B|) * \lambda\rfloor$ blocks from the blocking collection that present the lowest block co-occurrence score values (lines 9–12). We present an illustration of the *lBCE* heuristic in Example 9.

Algorithm 7: *lBCE*

```
   Input  : B: an input blocking collection
            WS: a weighting scheme
            λ: a percentage (0,1) that determines the reach of the pruning
   Output: a pruned blocking collection
1  begin
2      // create empty priority queue ordered by block co-occurrence
           score mean
3      Q ← GeneratePriorityQueue()
4      foreach B ∈ B do
5          foreach b ∈ B do
6              // using Definition 10
7              Q.push(b, BlockCooScore(b, B, WS))
8      limit ← ⌊(Σ_{B∈B} |B|) * λ⌋
9      for i = 1 to limit do
10         b ← Q.pop()
11         B_b ← B ∈ B, s.t. b ∈ B
12         B_b ← B_b\b
13     return B
```

Table 7 Parameters and calculations used by the *lBCE* heuristic in Example 9

Blocking collection	$\mathcal{B}_{D_1}$
lBCE parameters	$WS = CBS, \lambda = 0.25$
Input blocks	$R\{d_7, d_8, d_{12}\}$, $Spain\{d_2, d_3, d_8, d_9, d_{10}\}$, $C\{d_2, d_3, d_4\}$ and $L\{d_{10}, d_{14}\}$
Calculations	$BlockCooScore(C\{d_2, d_3, d_4\}) = \frac{5.5}{3}$, $BlockCooScore(R\{d_7, d_8, d_{12}\}) = \frac{4}{3}$, $BlockCooScore(Spain\{d2, d3, d8, d9, d10\}) = \frac{6}{5}$, $BlockCooScore(L\{d_{10}, d_{14}\}) = \frac{1}{1}$

Example 9 (*Low block co-occurrence excluder (lBCE)*) Considering the parameters and calculations presented in Table 7, the blocks $R\{d_7, d_8, d_{12}\}$, $Spain\{d_2, d_3, d_8, d_9, d_{10}\}$, $C\{d_2, d_3, d_4\}$ and $L\{d_{10}, d_{14}\}$[8] are processed by the *lBCE* algorithm as follows: (i) the priority queue $Q = \{(C\{d_2, d_3, d_4\}, \frac{5.5}{3}), (R\{d_7, d_8, d_{12}\}, \frac{4}{3}), (Spain\{d_2, d_3, d_8, d_9, d_{10}\}, \frac{6}{5}), (L\{d_{10}, d_{14}\}, \frac{2}{2})\}$ (see Definition 10) is generated (lines 3–7); (ii) the $limit$ is calculated as $\lfloor |\{R\{d_7, d_8, d_{12}\}, Spain\{d_2, d_3, d_8, d_9, d_{10}\}, C\{d_2, d_3, d_4\}, L\{d_{10}, d_{14}\}\}| * \lambda \rfloor = \lfloor 4 * 0.25 \rfloor = 1$ (line 8); and (iii) thus, a single block ($Q.pop() = L\{d_{10}, d_{14}\}$) is excluded from the blocking collection (lines 9–12). Note that this exclusion operation reduces the aggregated cardinality result from 69 to 68 comparisons between entities and it does not decrease the PC result of the pruned blocking collection (since d_{10} and d_{14} are not duplicated entities).

4.10 Impact of the approach over different blocking distributions

In this section, we discuss possible implications of the approach usage in three different scenarios, regarding the distribution of entities among blocks and the number of duplicated entities in the dataset:

- *Scenario A*: the indexing generated by the employed blocking keys produces a high-quality blocking collection. Also, all blocks are small, except a few of them. The few large blocks do not need to be resized, because each of them is representative of a large set of records referring to the same entity;
- *Scenario B*: the blocking keys produce a skewed blocking collection. In this scenario, splitting of large blocks and merging small blocks can provide a significant benefit;
- *Scenario C*: the dataset is almost clean except for a set of duplicated or maybe triplicated records. Therefore, keeping block sizes small is fundamental.

First, note that the quality results (PC, PQ and RR) generated by the proposed approach in all aforementioned scenarios may vary significantly, depending on the employed block size restrictions ($\{S_{min}, S_{max}\}$). In scenario A, the input blocks should not be resized significantly, since the merge of small blocks will decrease RR (without increasing PC or PQ) and the split of large blocks will enhance RR at the cost of reducing PC. For these reasons, the negative impact of the approach usage will be determined by the amount of split and merge operations that are performed over the input blocks, which are, in turn, dependent on the user-defined size restrictions ($\{S_{min}, S_{max}\}$). Therefore, regarding scenario A, the closer are the smallest

[8] For simplification, we consider that only the blocks $R\{d_7, d_8, d_{12}\}$, $Spain\{d_2, d_3, d_8, d_9, d_{10}\}$, $C\{d_2, d_3, d_4\}$ and $L\{d_{10}, d_{14}\}$ are pruned by the *lBCE* algorithm, but all blocks in $\mathcal{B}_{D_1}$ are considered in order to calculate the co-occurrence score between entities.

and the largest blocks in input blocking collection to S_{min} and S_{max} (respectively), the less significant is the impact of the approach over the pruned blocking result. In turn, the impact of the *shrink* and *exclude* steps over the pruned blocking collection ($\mathcal{B}'$) in scenario A will depend on the overall quality (PC an PQ) of the input blocking collection ($\mathcal{B}$). If $PC(\mathcal{B}) = 1$, then any shrink or block exclude operation will increase RR at the cost of decreasing PC. On the other hand, if $PC(\mathcal{B}) < 1$ and $PQ(\mathcal{B}) < 1$, then it is possible to shrink or exclude blocks to improve RR, without sacrificing PC and PQ. Thus, the lower the quality of the input blocking collection $\mathcal{B}$, the more significant should be the reach of the *shrink* and *exclude* steps. In other words, a user should set high λ values for tackling low-quality blocks.

Regarding scenario B, the usage of the proposed approach may potentially enhance the PC, PQ and RR results of the pruned blocking collection. Since the distribution of the entities among the blocks is skewed, then it follows that: (i) it is possible to split blocks to improve PQ and RR, without sacrificing PC; (ii) it is possible to merge blocks to improve PC, PQ and RR; and (iii) it is possible to shrink and exclude blocks to increase PQ and RR, without sacrificing PC. For these reasons, in this scenario, the proposed approach is less sensitive to the employed block size interval ($\{S_{min}, S_{max}\}$) and can be potentially effective to control the size of the pruned blocks as well as to improve PC, PQ and RR. In practice, the smaller is S_{max} in comparison with the size of the largest block in the input blocking collection $\mathcal{B}$, the greater tends to be the positive impact of the approach over $PQ(\mathcal{B}')$ and $RR(\mathcal{B}, \mathcal{B}')$. Similarly, the greater is S_{min} in comparison with the size of the smallest block in $\mathcal{B}$, the more significant tends to be the positive impact of the approach over $PC(\mathcal{B}')$. Lastly, the reach of the *shrink* and *exclude* steps should be inversely proportional to the expected quality of the input blocking collection, as discussed earlier in this section.

In scenario C, the input blocking collection may greatly benefit from the shrink and split operations performed by the proposed approach. In other words, the approach can shrink and split large blocks to improve PQ and RR, without decreasing PC. Assuming scenario C, let us suppose that there are at most k records in the dataset that refer to the same real-world entity. Also, let S_{avg} be averaged size of the blocks in the input blocking collection $\mathcal{B}$. If $k \ll S_{avg}$, then the split and shrink operations of the proposed approach may generate a pruned blocking collection $\mathcal{B}'_k$, which contains only blocks with at most k entities, and still produce $||\mathcal{B}'_k|| \ll ||\mathcal{B}||$ and $PQ(\mathcal{B}'_k) \gg PQ(\mathcal{B})$, while maintaining $PC(\mathcal{B}'_k) \simeq PC(\mathcal{B})$. For doing so, the closer a user is able to approximate k and S_{max}, the greater tends to be the positive impact of the approach over PQ and RR.

The negative or positive impacts of the proposed approach over the quality results (PC, PQ and RR) of the pruned blocking collection, which are mentioned in the related discussion of the aforementioned scenarios (A–C), will depend on how meaningful is the co-occurrence score of the entities among the blocks. In turn, this meaningfulness depends on the efficacy of the set of blocking keys employed to index the dataset, which generates the input of the approach. Thus, the usage of a proper set of blocking key functions can increase the impact of the proposed approach over the quality of the pruned blocking collection. Discovering effective blocking key functions is outside the scope of this work. We assume that a set of effective blocking key functions has been preidentified by specialists or by an automatic approach [3,19].

5 Evaluation

In this section, we present an empirical evaluation of the heuristics proposed in Sect. 4. The main goal of this assessment is to evaluate the efficiency and efficacy of different heuristics for controlling block sizes, as well as the influence of parameters such as the weighting scheme and λ (percentage that defines the reach of the shrinking heuristics) over the obtained results.

5.1 Datasets

We have selected four datasets in order to evaluate the proposed heuristics: *CoraATDV*[9], *DBLPM4*[10], *FebrlData*[11] and *SongsDataset*.[12] The first dataset consists of publication data and it was investigated by many other entity resolution works. In turn, the *DBLPM4* dataset contains data from publication titles and it was generated by the authors of [17] to evaluate collective classification techniques in batch mode. The third dataset contains publication data from DBLP and Google Scholar repositories. Fourth, we have generated the *FebrlData* dataset, which contains personal data, using the Febrl[13] tool by employing the following parameters: $max\_duplicate\_per\_record = 10$, $max\_modification\_per\_record = 3$, $max\_modification\_per\_field = 2$ and $\#duplicates \approx 5\%$ of dataset size. Lastly, the *SongsDataset* is a subset of a dataset that contains details about songs, such as title, artist name and release.

The *Cora*, *DBLPM4*, *FebrlData* and *SongsDataset* datasets contain approximately $2*10^3$, $4*10^3$, $7*10^4$ and $1.5*10^5$ entities, respectively. These datasets have been selected mainly because they present ground truth results.

5.2 Experimental design

Note that by varying the values of parameters employed by the *MCBS* metaheuristic (for example, λ, WS, S_{min}, S_{max}, *shrink step*, *merge step*, *split step* and *exclude step*) it is possible to generate a large amount of different algorithms. However, we have limited these values to generate a reasonable amount of algorithms taking into account quite different strategies. We explore six combinations of heuristics in order to tackle the investigated problem. These combinations are shown in Table 8, in which we present: (i) the algorithm denomination and (ii) the heuristics that are employed in order to execute the four steps of the *MCBS* metaheuristic.

Since we are also interested in investigating the influence of the weighting scheme and the λ value over the obtained results, we vary the value of these parameters in the experimental design. For each dataset, we evaluate all the algorithms defined in Table 8 combined with three λ values ($\{0.2, 0.3, 0.4\}$)[14] and two different weighting schemes ($ECBS$ and $ARCS$). Besides, for each dataset, we evaluate two different interval sizes (i.e., S_{min} and S_{max}).

[9] https://github.com/TeamCohen/secondstring/tree/master/data.

[10] http://dblab.cs.toronto.edu/project/stringer/.

[11] https://sites.google.com/site/febrldata/febrl70k/febrl_70k.csv.

[12] https://sites.google.com/site/anhaidgroup/useful-stuff/data.

[13] http://sourceforge.net/projects/febrl/.

[14] λ values greater than 0.4 have produced low-quality results in the conducted experiments and thus are not reported in this paper.

Table 8 List of algorithms generated using the *MCBS* metaheuristic

Algorithm denomination	Steps of the *MCBS* metaheuristic			
	Shrink	Split	Merge	Exclude
lECP	lECP	CooSlicer	MaxIntersectionMerge	–
lBCP	lBCP	CooSlicer	MaxIntersectionMerge	–
LBSP	LBSP	CooSlicer	MaxIntersectionMerge	–
lBCE	–	CooSlicer	MaxIntersectionMerge	lBCE
lECP+lBCE	lECP	CooSlicer	MaxIntersectionMerge	lBCE
Default	–	CooSlicer	MaxIntersectionMerge	–

Therefore, for each dataset, we evaluate a total of (i) five algorithms combined with three different λ values, two different weighting schemes and two interval sizes, and (ii) one algorithm[15] combined with two different weighting schemes and two interval sizes. Therefore, we explore $(4*5*3*2*2)+(4*1*2*2) = (240)+(16) = 256$ combinations of parameter values using the proposed heuristics. The sets of blocking key functions employed to index the datasets are presented in "Appendix A."

As a baseline, we use the sorted neighborhood [4] algorithm with a fixed window size (w). This algorithm has been selected because it is possible to configure beforehand the number of comparisons generated by the indexing phase, based on the employed w value. Therefore, we can limit the number of comparisons generated by the SN indexing taking into account the maximum number of comparisons that the pruned blocking collection $\mathcal{B}'$ (containing block sizes between S_{min} and S_{max}) may produce. Hence, we can compare the results generated by the proposed approach with the results produced by the SN method under identical constraints, i.e., the generation of a maximum number of comparisons $||\mathcal{B}'_{max}||$ ("Appendix B") between entities by the indexing phase.

For each evaluated dataset, we execute $|\mathcal{F}|$ passes of the SN algorithm, such that each pass uses a different blocking key function $f \in \mathcal{F}$. Also, we evaluate the SN algorithm using two variations of the maximum window size (w_{max}), as described in "Appendix B."

5.2.1 Measures

For each algorithm execution, we measure both the efficiency and the efficacy of the evaluated algorithms. The efficiency of the algorithms is measured by the execution time required for shrinking, splitting, merging and excluding blocks (the four steps of the *MCBS* metaheuristic). In turn, we measure the efficacy of the algorithms using the quality metrics presented in Sect. 3.1, i.e., pairs completeness (PC), reduction ratio (RR) and pair quality (PQ).

The reported efficiency results are the average of three executions of each algorithm. For each algorithm in Table 8, we report the execution time of the parameter combination which has generated the best F-measure between PC and RR (for each evaluated dataset and interval size). More formally, let $D_{set} = \{CoraATDV, DBLPM4, FebrlData, SongsDataset\}$, $WS_{set} = \{ARCS, ECBS\}$ and $\lambda_{set} = \{0.2, 0.3, 0.4\}$. For each algorithm alg in Table 8, we

[15] The "Default" combination (in Table 8) is the only algorithm that is not influenced by the λ parameter.

report the execution time of the algorithm that generates the following quality result[16] (for each evaluated dataset and interval size):

$$\underset{D \in D_{set}, WS \in WS_{set}, \lambda \in \lambda_{set}}{\arg\max} 2.\frac{PC(alg(D, WS, \lambda)) * RR(alg(D, WS, \lambda))}{PC(alg(D, WS, \lambda)) + RR(alg(D, WS, \lambda))}$$

In order to calculate the SN indexing time, we have only measured the time required to sort the entities (for each employed blocking key) and generate the pairs of entities to be compared (by sliding the employed window size). The reported efficiency results of the SN algorithm are the average of three executions for each parameter combination. For each dataset, we report the efficiency result of the SN algorithm using the window size that has generated the highest harmonic mean[17] between PC and RR.

5.2.2 Implementation and environment

We have implemented the proposed heuristics in Java and executed the tests using a Core i-7 CPU machine with 16GB of RAM. We have neither measured the execution time required for reading and indexing (i.e., the generation of the input blocking collection $\mathcal{B}$ by employing the input set of blocking key functions) the datasets nor the time required to calculate the evaluated quality metrics, since we are mainly interested in measuring the efficiency of the proposed heuristics. On average, the time required for reading and indexing the datasets used in the experiments is equal to 2.5×10^{-1} s.

5.3 Results

The quality results of the evaluated algorithms employing the *CoraATDV*, *DBLPM4*, *Febrl-Data* and *SongsDataset* datasets are shown in Tables 9, 10, 11 and 12, respectively. Regarding efficiency, the execution times of the evaluated algorithms using the *CoraATDV* dataset are presented in Figs. 2 and 3. In turn, the efficiency results on the *DBLPM4* dataset are depicted in Figs. 4 and 5. The efficiency of the algorithms employing the *FebrlData* dataset is shown in Figs. 6 and 7. Lastly, The execution times of the evaluated algorithms using the *SongsDataset* dataset are shown in Figs. 8 and 9.

5.4 Discussion

First of all, note that for all scenarios: (i) the greater is the value of λ, the lower tends to be the value of PC and the greater tends to be the value of RR; (ii) the greater is the employed interval size, the greater tends to be the value of PC and the lower tends to be the value of RR; and (iii) the lower is the value of the fixed window size w, the lower tends to be the value of PC and the greater tends to be the value of RR. These three patterns are important in order to better understand the reported results and highlight that these parameters may have a strong influence over the reported results.

Results on Cora (*Quality results*)

Regarding Table 9, note that the ARCS weighting scheme has produced the best quality results. Also note that the λ value has caused quite different influences over the quality

[16] These results are highlighted in bold in Tables 9, 10, 11 and 12.

[17] These results are highlighted in bold in Tables 9, 10, 11 and 12.

Table 9 Quality results on Cora

Algorithm	Cora							
	$\{S_{min}, S_{max}\}$		{50, 150}			{100, 200}		
	λ		0.2	0.3	0.4	0.2	0.3	0.4
lECP	WS = ARCS	PC	0.94	0.94	**0.95**	0.98	0.97	**0.98**
		RR	0.80	0.82	**0.85**	0.75	0.78	**0.80**
	WS = ECBS	PC	0.94	0.93	0.92	0.97	0.96	0.96
		RR	0.89	0.82	0.85	0.75	0.78	0.80
lBCP	WS = ARCS	PC	**0.92**	0.85	0.80	0.95	**0.90**	0.84
		RR	**0.80**	0.83	0.86	0.74	**0.79**	0.82
	WS = ECBS	PC	0.88	0.82	0.76	0.92	0.87	0.80
		RR	0.80	0.83	0.86	0.74	0.79	0.82
LBSP	WS = ARCS	PC	**0.92**	0.86	0.83	0.96	0.91	**0.87**
		RR	**0.79**	0.82	0.85	0.74	0.78	**0.82**
	WS = ECBS	PC	0.89	0.84	0.80	0.93	0.89	0.83
		RR	0.79	0.82	0.84	0.74	0.78	0.82
lBCE	WS = ARCS	PC	0.92	0.91	0.89	0.95	0.94	**0.93**
		RR	0.77	0.81	0.84	0.73	0.77	**0.80**
	WS = ECBS	PC	0.93	0.91	**0.90**	0.96	0.93	0.90
		RR	0.80	0.83	**0.85**	0.74	0.78	0.81
LECP+lBCE	WS = ARCS	PC	0.91	0.87	**0.87**	**0.95**	0.89	0.85
		RR	0.84	0.88	**0.91**	**0.81**	0.85	0.89
	WS = ECBS	PC	0.92	0.87	0.83	0.93	0.86	0.83
		RR	0.85	0.87	0.91	0.81	0.85	0.89
Default	WS = ARCS	PC	**0.96**			**0.99**		
		RR	**0.72**			**0.65**		
	WS = ECBS	PC	0.93			0.98		
		RR	0.72			0.65		
SN (baseline)	$w = 2^{-1}\sqrt{w_{max}}$	PC	**0.74**					
		RR	**0.83**					
	$w = 2^{-2}\sqrt{w_{max}}$	PC	0.51					
		RR	0.92					

results, depending on the employed algorithm. For the *lECP* algorithm, greater λ values have effectively improved the RR results without compromising the PC values, which means that the algorithm was able to perform an effective pruning by removing entities from the blocks without preventing the comparison between duplicated entities. This result is especially true when the ARCS scheme is employed.

In turn, the λ value has produced much more influence over the results of the *lBCP* algorithm. It is easy to note the significantly different PC results reported by this algorithm the employed λ values. For this reason, the *lBCP* has reported slightly lower PC values and better RR results than those reported by *lECP*. Similarly to *lBCP*, the results reported by the *LBSP* algorithm were significantly influenced by the employed λ values. (Note the great difference between the PC values reported by this algorithm.) In fact, the quality results reported by the *lECP* and *LBSP* algorithms are very similar, which is best explained by the

Table 10 Quality results on DBLPM4

Algorithm	DBLPM4							
	$\{S_{min}, S_{max}\}$		{20, 60}			{60, 100}		
	λ		0.2	0.3	0.4	0.2	0.3	0.4
lECP	WS = ARCS	PC	**0.85**	0.83	0.76	**0.86**	0.86	0.79
		RR	**0.87**	0.87	0.92	**0.78**	0.78	0.85
	WS = ECBS	PC	0.73	0.72	0.66	0.78	0.78	0.72
		RR	0.87	0.87	0.92	0.78	0.78	0.85
lBCP	WS = ARCS	PC	**0.82**	0.77	0.72	**0.84**	0.8	0.74
		RR	**0.83**	0.86	0.88	**0.72**	0.76	0.8
	WS = ECBS	PC	0.71	0.67	0.61	0.77	0.73	0.66
		RR	0.83	0.86	0.88	0.72	0.76	0.80
LBSP	WS = ARCS	PC	**0.84**	0.80	0.77	0.86	**0.83**	0.79
		RR	**0.83**	0.85	0.88	0.71	**0.75**	0.79
	WS = ECBS	PC	0.75	0.72	0.69	0.8	0.77	0.74
		RR	0.83	0.85	0.88	0.72	0.75	0.79
lBCE	WS = ARCS	PC	**0.88**	0.87	0.84	0.9	**0.88**	0.83
		RR	**0.82**	0.84	0.86	0.7	**0.74**	0.78
	WS = ECBS	PC	0.77	0.74	0.68	0.81	0.78	0.71
		RR	0.82	0.84	0.87	0.70	0.74	0.77
LECP+lBCE	WS = ARCS	PC	**0.78**	0.75	0.59	**0.81**	0.76	0.6
		RR	**0.90**	0.91	0.95	**0.82**	0.94	0.91
	WS = ECBS	PC	0.69	0.64	0.53	0.74	0.69	0.58
		RR	0.9	0.91	0.95	0.82	0.85	0.91
Default	WS = ARCS	PC	**0.91**			**0.92**		
		RR	**0.78**			**0.63**		
	WS = ECBS	PC	0.80			0.85		
		RR	0.78			0.63		
SN (baseline)	$w = 2^{-1}\sqrt{w_{max}}$	PC	0.87					
		RR	0.47					
	$w = 2^{-2}\sqrt{w_{max}}$	PC	**0.86**					
		RR	**0.75**					

fact that the pruning of large blocks (see line 5 of Algorithm 4) is sufficient to produce a significant impact over the RR and PC results. In general, the *LBSP* algorithm has produced slightly better PC results and lower RR values than the *lECP* algorithm (which is obviously caused by the fact that *LBSP* algorithm performs the pruning over a subset of the blocking collection that is pruned by the *lECP* algorithm, i.e., the large blocks).

Concerning the Cora dataset, the *lBCE* algorithm was the only combination which produced better quality results employing the ECBS weighting scheme. This algorithm has produced similar results to those reported by the *lBCP* and *LBSP* algorithms (and slightly lower results than those reported by the *lECP* algorithm). This result highlights that the exclusion of blocks that present low block co-occurrence score (Definition 10) can be as effective as the pruning of blocks by excluding entities. As expected, the *ECP*+*lBCE* algorithm has reported better RR results than all the other algorithms (due to the application of two pruning

Table 11 Quality results on FebrlData

Algorithm	FebrlData							
	$\{S_{min}, S_{max}\}$		{200, 300}			{300, 400}		
	λ		0.2	0.3	0.4	0.2	0.3	0.4
lECP	WS = ARCS	PC	**0.96**	0.96	0.84	**0.96**	0.96	0.84
		RR	**0.93**	0.93	0.96	**0.90**	0.90	0.95
	WS = ECBS	PC	0.83	0.82	0.81	0.85	0.83	0.82
		RR	0.93	0.93	0.93	0.9	0.9	0.95
lBCP	WS = ARCS	PC	**0.92**	0.88	0.75	**0.90**	0.88	0.76
		RR	**0.91**	0.92	0.93	**0.88**	0.89	0.91
	WS = ECBS	PC	0.80	0.73	0.68	0.82	0.79	0.69
		RR	0.91	0.92	0.93	0.88	0.89	0.91
LBSP	WS = ARCS	PC	**0.92**	0.89	0.80	**0.91**	0.89	0.81
		RR	**0.91**	0.92	0.92	**0.87**	0.89	0.90
	WS = ECBS	PC	0.81	0.77	0.72	0.82	0.81	0.74
		RR	0.91	0.92	0.93	0.87	0.89	0.90
lBCE	WS = ARCS	PC	**0.96**	0.94	0.87	**0.98**	0.92	0.86
		RR	**0.91**	0.92	0.93	**0.87**	0.89	0.91
	WS = ECBS	PC	0.87	0.85	0.82	0.87	0.84	0.82
		RR	0.90	0.92	0.93	0.86	0.89	0.9
LECP+lBCE	WS = ARCS	PC	**0.87**	0.80	0.47	**0.88**	0.8	0.46
		RR	**0.94**	0.95	0.98	**0.92**	0.92	0.97
	WS = ECBS	PC	0.79	0.76	0.57	0.82	0.76	0.49
		RR	0.94	0.95	0.97	0.92	0.93	0.97
Default	WS = ARCS	PC	**0.99**			**0.99**		
		RR	**0.88**			**0.85**		
	WS = ECBS	PC	0.87			0.88		
		RR	0.88			0.85		
SN (baseline)	$w = 3^{-1}2^{-4}\sqrt{w_{max}}$	PC	**0.99**					
		RR	**0.88**					
	$w = 2^{-6}\sqrt{w_{max}}$	PC	0.99					
		RR	0.76					

strategies). In this particular dataset, the *lECP* + *lBCE* algorithm has reported better results than the *lBCP*, LBSP and lBCE algorithms (considering the harmonic mean between PC and RR), but slightly lower results than the lECP algorithm (especially for PC). Lastly, note that the *Default* algorithm has produced the highest PC results (especially, when employing the ARCS scheme and the large interval sizes). However, since this algorithm does not employ any pruning approach (other than splitting and merging blocks), its RR results are significantly lower than those reported by the other evaluated approaches.

In the great majority of cases, the proposed heuristics have outperformed the results generated by the SN algorithm, considering the harmonic mean between PC and RR. Regarding PQ, all the evaluated algorithms have reported results between 0.3 and 1.1. In general, the more pruning approaches are employed, the greater it tends to be the PQ result of the algo-

Table 12 Quality results on SongsDataset

Algorithm	SongsDataset							
	$\{S_{min}, S_{max}\}$		{10, 400}			{50, 500}		
	λ		0.2	0.3	0.4	0.2	0.3	0.4
lECP	WS = ARCS	PC	0.89	0.88	0.88	**0.91**	0.91	0.91
		RR	0.91	0.91	0.91	**0.9**	0.9	0.9
	WS = ECBS	PC	0.89	**0.89**	0.89	0.91	0.9	0.9
		RR	0.91	**0.91**	0.91	0.9	0.9	0.9
lBCP	WS = ARCS	PC	0.72	0.6	0.5	0.73	0.62	0.51
		RR	0.84	0.87	0.9	0.82	0.85	0.88
	WS = ECBS	PC	**0.73**	0.6	0.49	**0.74**	0.62	0.5
		RR	**0.84**	0.87	0.9	**0.82**	0.85	0.88
LBSP	WS = ARCS	PC	0.78	0.69	0.6	0.79	0.7	0.62
		RR	0.84	0.87	0.9	0.82	0.85	0.88
	WS = ECBS	PC	**0.8**	0.7	0.6	**0.8**	0.71	0.62
		RR	**0.84**	0.87	0.9	**0.82**	0.85	0.88
lBCE	WS = ARCS	PC	0.89	0.84	0.76	**0.9**	0.84	0.76
		RR	0.82	0.84	0.86	**0.82**	0.84	0.87
	WS = ECBS	PC	0.85	**0.99**	0.98	0.98	0.97	0.95
		RR	0.91	**0.78**	0.78	0.75	0.75	0.76
LECP+lBCE	WS = ARCS	PC	0.69	0.64	0.51	0.7	0.65	0.52
		RR	0.88	0.94	0.96	0.92	0.94	0.95
	WS = ECBS	PC	**0.87**	0.85	0.83	**0.82**	0.78	0.36
		RR	**0.91**	0.91	0.92	**0.90**	0.90	0.92
Default	WS = ARCS	PC	0.96			0.98		
		RR	0.78			0.74		
	WS = ECBS	PC	**0.99**			**0.99**		
		RR	**0.78**			**0.74**		
SN (baseline)	$w = 3^{-1}\sqrt{w_{max}}$	PC	0.98					
		RR	0.56					
	$w = 5^{-1}\sqrt{w_{max}}$	PC	**0.97**					
		RR	**0.74**					

rithm. For this reason, the algorithm $lECP + lBCE$ employing high λ values has produced the highest PQ results.

Results on Cora (*Efficiency*)

Regarding Fig. 2, it is easy to notice that all the evaluated algorithms have produced quite efficient execution times. In general, the *shrink* and *split* steps of the process require more time than the *merge* and *exclusion* steps. Also note that when the *shrink* step is not performed by the algorithm (for instance, when the algorithms *lBCE* and *Default* are employed), the *split* step becomes more costly. Regarding the lower interval size {50, 150} (Fig. 2), the *lECP* and *Default* algorithms have reported the lowest execution times (among the proposed heuristics), followed by the remaining approaches. Also note that the execution time reported by the *lBCE(ECBS,0.4)* is significantly higher than the other algorithms (especially in the

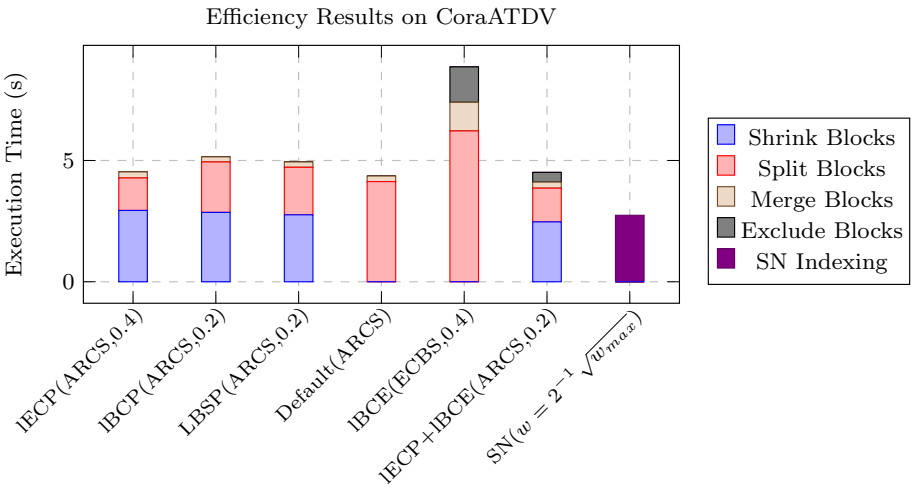


Fig. 2 Efficiency results on CoraATDV ($S_{min} = 50$, $S_{max} = 150$)

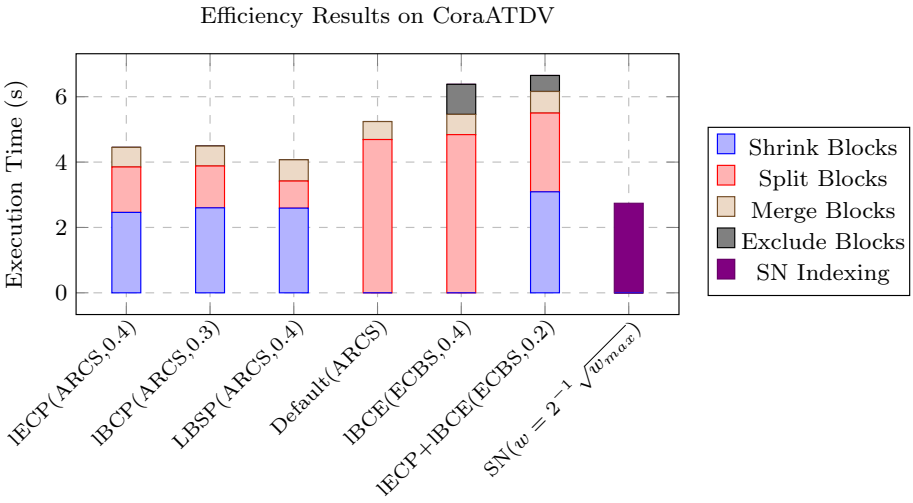


Fig. 3 Efficiency results on CoraATDV ($S_{min} = 100$, $S_{max} = 200$)

split phase). This is because the *ECBS* scheme employs a more complex formula than the *ARCS* scheme, and thus, the algorithms that used the former scheme have usually reported higher execution times.

Results on DBLPM4 (*Quality results*)

Regarding Table 10, note that the results reported by the majority of the algorithms have been strongly influenced by the employed λ value. In all cases, lower λ values have produced higher PC values and lower RR results (when compared to higher λ values). Also note that similarly to the Cora dataset, all the algorithms have produced better results (considering the harmonic mean between PC and RR) when the ARCS scheme was employed. Also, regarding

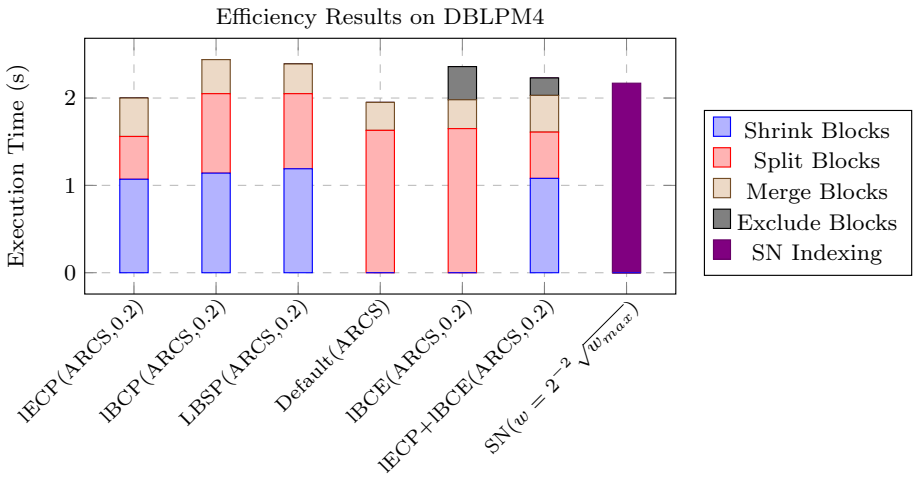


Fig. 4 Efficiency results on DBLPM4 ($S_{min} = 20$, $S_{max} = 60$)

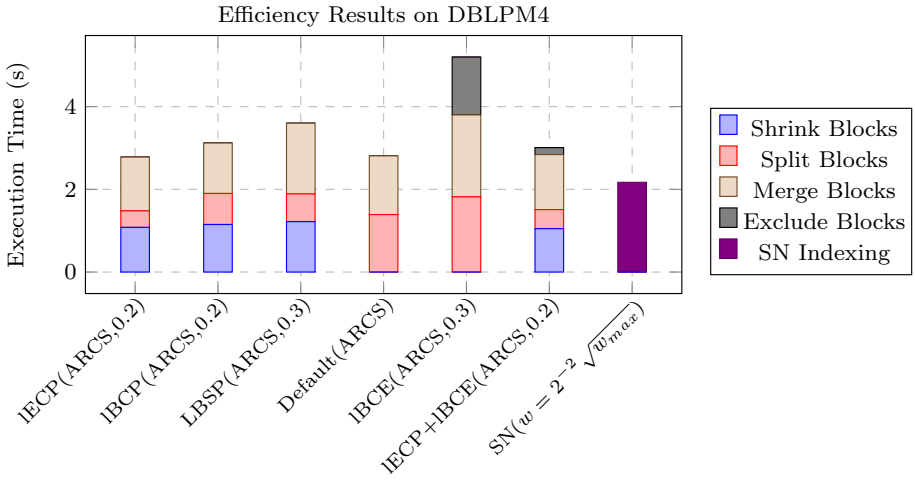


Fig. 5 Efficiency results on DBLPM4 ($S_{min} = 60$, $S_{max} = 100$)

the interval size variation, note that in all cases the higher interval size has produced slightly lower RR results and higher PC values (when compared to the lower interval size).

The single pruning approaches (*lECP*, *lBCP* and *LBSP*) have produced similar results. In particular, the *lECP* algorithm has reported slightly better results when compared to the other two approaches (*lBCP* and *LBSP*), especially when employing the ARCS scheme combined with low λ values. In turn, the *lBCE* algorithm has reported quite high PC results, but slightly lower RR values than the *lECP* algorithm. For this dataset, the *lECP*+*lBCE* approach has produced much lower PC results and higher RR results than all the other approaches, which indicates that as datasets become larger, the results tend to become more sensitive to high λ values and to the double pruning (block shrinking + block exclusion) approach. Similarly to the Cora dataset, the *Default* algorithm has reported the highest PC values at the cost of producing low RR results (see Table 10).

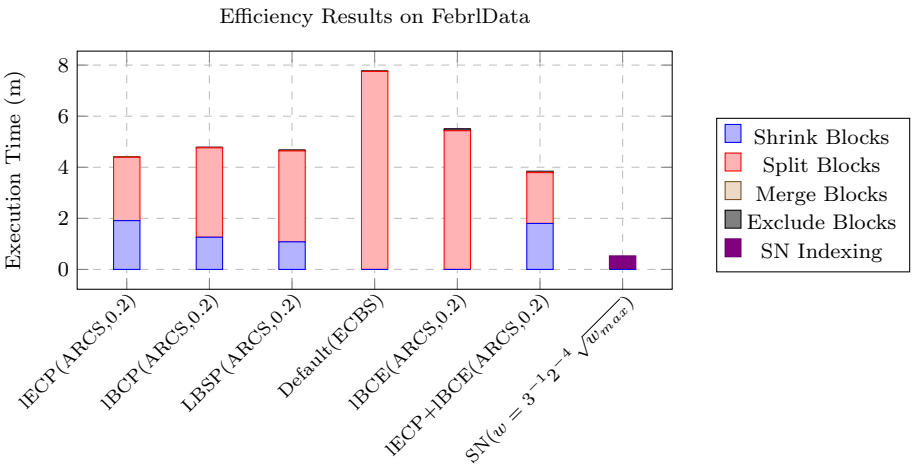


Fig. 6 Efficiency results on FebrlData ($S_{min} = 200$, $S_{max} = 300$)

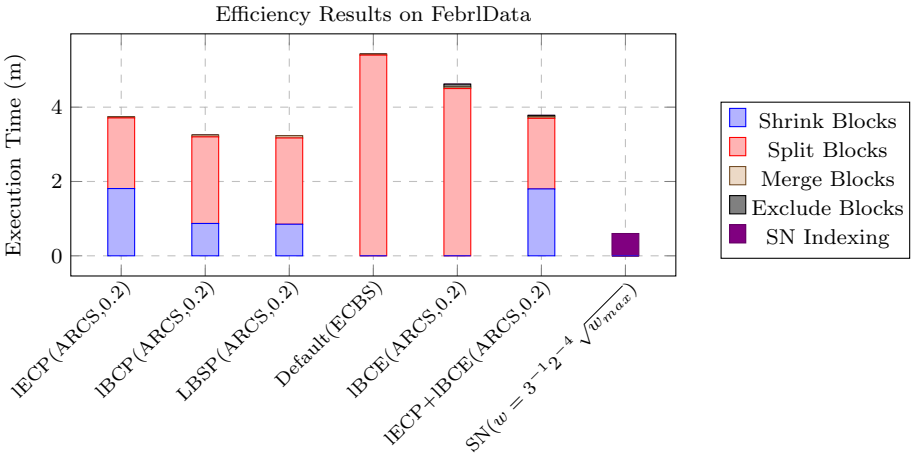


Fig. 7 Efficiency results on FebrlData ($S_{min} = 300$, $S_{max} = 400$)

Considering the DBLPM4 dataset, the PC result produced by the SN algorithm have been slightly influenced by the employed w value. This algorithm was able to improve the RR result by employing small window sizes without sacrificing the PC results significantly. However, the SN algorithm has produced lower results than those reported by the best performing heuristic (i.e., *lECP* algorithm using the ARCS scheme and $\lambda = 0.2$). Regarding PQ, all the algorithms have reported results between 0.03 and 0.1, whereas the SN algorithm has reported PQ results between 0.02 and 0.09.

Results on DBLPM4 (*Efficiency*)

Similarly to the Cora dataset, in the *DBLPM4* dataset, the algorithms that employed ARCS scheme have generated lower execution times than those that employed the ECBS scheme. Note that, for the lower interval size (Fig. 4), the algorithms have spent more time performing *prune* and *split* operations, whereas for the higher interval size (Fig. 5) the algorithms have spent more time performing *prune* and *merge* operations. Thus, it is easy to notice that the

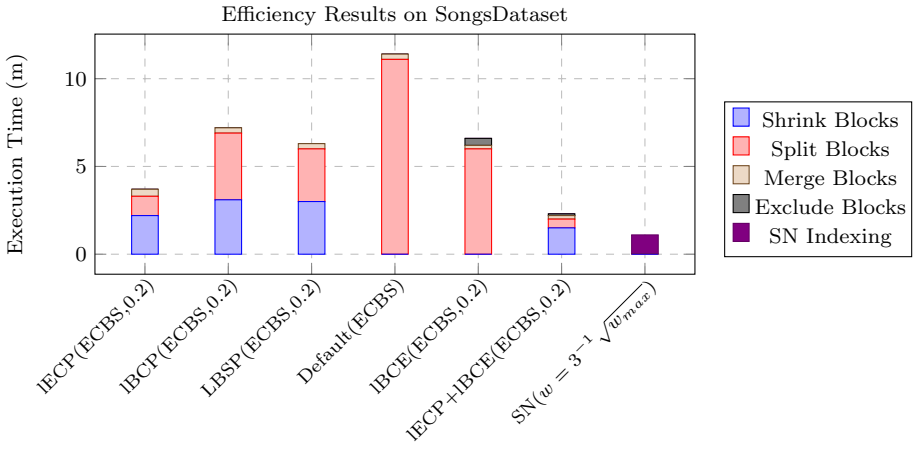


Fig. 8 Efficiency results on SongsDataset ($S_{min} = 10$, $S_{max} = 400$)

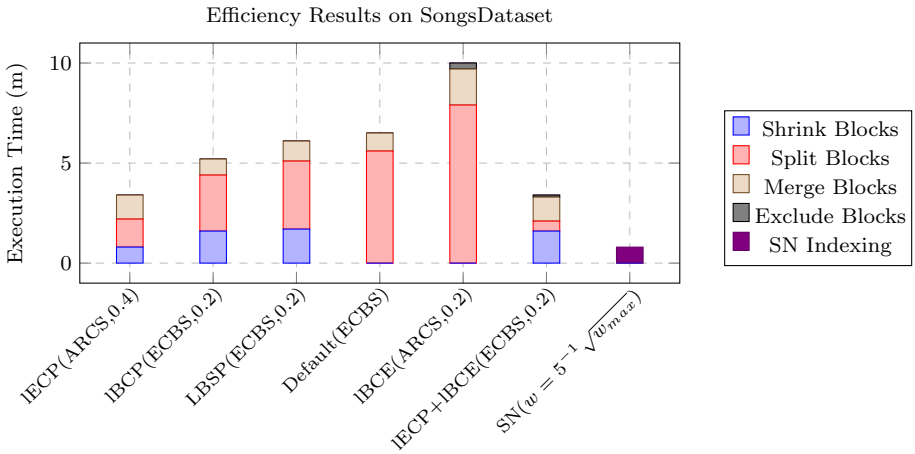


Fig. 9 Efficiency results on SongsDataset ($S_{min} = 50$, $S_{max} = 500$)

interval size parameter can significantly influence both the time required to perform each step of the *MCBS* metaheuristic and the overall execution time. For both interval sizes, two of the single pruning algorithms (*lECP* and Default) have generated the lowest executions times, followed by the remaining algorithms.

Results on FebrlData (*Quality results*)

Similarly to the *Cora* dataset, in Table 11 the quality results of all algorithms have also been strongly influenced by the employed λ value (for example, note the difference between the quality results generated by $\lambda = 0.2$ and $\lambda = 0.4$). Also similar to the *DBLPM4* and *CoraATDV* datasets, the quality results of the algorithms that employ the ARCS scheme have been better than the ones of the algorithms that employ the ECBS scheme.

The *lECP* and *lBCE* algorithms have reported significantly better quality results (considering the harmonic mean between PC and RR) than the other approaches (especially for low λ values). Also, note that the *LECP*+*lBCE* algorithm has produced quite low PC results

(when compared to the other results). In turn, the *Default* algorithm has reported quite high ($\approx$ 100%) PC results (especially when the ARCS scheme was employed), but at the same time it has generated the lowest RR results (Table 11). Once again, the best performing heuristic (*lECP* using the ARCS scheme and $\lambda = 0.2$) has overcome the results generated by the SN algorithm (considering the harmonic mean between PC and RR). Regarding PQ, all the algorithms have reported results between 1.10^{-4} and 7.10^{-4}.

Results on FebrlData (*Efficiency*)

Similarly to all other datasets, the algorithms that employed ARCS scheme have generated lower execution times than those that employed the ECBS scheme. In Figs. 8 and 9, the single pruning algorithms *lECP*, *lBCP*, *LBSP* and *lBCE* have presented similar execution times, followed by the Default algorithm. In particular, the *Default* algorithm has reported a significantly high execution time, which can be explained by the use of the ECBS schema and the execution of a costly *merge* step (which is caused by the absence of the pruning step). In all cases, the algorithms have spent the majority of time performing the *merge* step, which indicates that a large number of merge operations was necessary in order to ensure a minimum block size in the pruned blocking collection.

Results on SongsDataset (*Quality results*)

The *lECP*, *lBCE* and *Default* algorithms have produced satisfactory results (Table 12) by processing the SongsDataset. These algorithms have generated F-measure results higher than 0.9, considering the harmonic mean between PC and RR. Similarly to the other datasets, the *lECP* algorithm using the ARCS weighting scheme has produced the best quality results. In turn, the *Default* and *lBCE* have generated high PC results, at the cost of producing lower RR results, when compared to the *lECP*, *LBSP* and *LECP*+*lBCE* algorithms. In general, the ECBS scheme has produced higher-quality results than the ARCS scheme, which indicates that the former scheme generates more meaningful co-occurrence scores when the algorithm needs to process larger datasets.

Results on SongsDataset (*Efficiency*)

The algorithms *lECP* and *lECP* +*lBCE* have produced the lowest execution times by processing the SongsDataset. This result confirms that the reasoning employed by the *lECP* heuristic is very efficient to shrink the blocks, even when the sizes of the dataset is significant. The *lBCP* and *LBSP* algorithms have executed slightly slower than the *lECP* algorithm. In turn, the *Default* and *lBCE* algorithms have produced the highest execution times. Note that this result indicates that the shrink step produces a significant impact over the efficiency of the subsequent steps of the proposed approach (*split*, *merge* and *exclude*), i.e., the algorithms which do not employ the shrink step will need to process more entities in the remaining steps of the approach, and thus, they produce higher execution times. Finally, note that the SN indexing has executed faster than the proposed algorithms at the cost of producing low RR results. This result entails that, considering the employed parameters, the SN indexing generates more comparisons between entities, and thus, the comparison phase will become more costly.

6 Related work

Many works [13,20,32,39] have proposed algorithms that aim to control the size of clusters generated by clustering techniques. Our work is similar to those proposed by [13,20,32, 39] because we employ algorithms for controlling the size of blocks, but we evaluate the

proposed algorithms in the context of indexing using multiple blocking key functions for entity resolution, instead of dealing with data clustering. Besides, the approaches proposed by the authors of [13,20,32,39] require the similarity computation between all pairs of records in the processed dataset, whereas our work aims to avoid the comparison between all pairs of entities by pruning an input blocking collection.

The authors of [11] have proposed a locality sensitive hashing (LSH)-based blocking approach that takes all attributes of a tuple into consideration and produces much smaller blocks, compared with traditional methods that consider only a few attributes. Their approach combines LSH and distributed representations of tuples to transform the problem of blocking into finding tuples in a high-dimensional similarity space. For doing so, the distributed representation of every tuple is indexed based on hash tables using the random hyperplane method. Since the complexity of their proposed algorithm is bounded by the number of hash tables and the size of the largest block (b_{max}) in any hash table, the authors explore the theoretical guarantees associated with LSH functions in order to tune the values of parameters employed by their approach to improve its performance. Although the approach proposed by [11] has presented significant contributions by using LSH functions and semantic similarity between tuples for blocking, in practice they are only able to reduce the occupancy rate (the expected number of tuples in any given block) by increasing the number of hash functions in order to generate an expected occupancy of the blocks. In other words, their approach does not guarantee hard size constraints ($\{S_{min}, S_{max}\}$) over the generated blocks. Although the authors of [11] mention the possibility of adapting a method proposed in [8] in the context of LSH-based blocking to achieve guaranteed limits, this attempt is not explored in their work.

In [26], the authors propose an agnostic (schema free) blocking approach, denominated meta blocking, which tackles the pruning of a blocking collection (Definition 5) by exploiting block co-occurrence and different weighting schemes. The approach consists on producing an undirected blocking graph (UBG) and then pruning the UBG by employing node- and edge-centric approaches. Lastly, the pruned UBG is processed (blocking collecting step) in order to generate the final blocking result. A supervised meta blocking approach has also been proposed by the authors of [27], which tackles the supervised meta blocking problem by employing a generalized blocking graph, labeled examples and similar pruning approaches to those proposed in [26]. Our work is similar to those in [26,27] because we also explore block co-occurrence in order to prune a blocking collection. Nevertheless, the approaches proposed in [26,27] give no control over the size of the blocks that are produced, whereas our work aims to prune a blocking collection in order to ensure hard size constraints over the generated blocks (i.e., we tackle the problem described in Definition 6). Besides, our approach does not require labeled examples (as required in [27]).

The work presented in [36] explores an iterative blocking approach that uses labeled training examples and proposes merge and split operations in order to block datasets. Our work is different from [36] because our approach is unsupervised and is able to ensure hard size constraints over the produced blocks. Lastly, in [12] the authors propose an approach to control block sizes for entity resolution based on two recursive algorithms (*SimilarityBasedClustering* and *SizeBasedClustering*) that (i) employ blocking key functions iteratively in order to split large blocks; (ii) merge small blocks based on the similarity between the blocking key values of the blocks; and (iii) use a penalty function in order to determine whether or not two blocks should be merged. Our work is similar to [12] because we also aim to ensure hard size restrictions over blocks for entity resolution. However, the approach proposed by [12] is not suitable for pruning a blocking collection that is initially indexed by multiple blocking key functions (Definition 6) because the algorithms *SimilarityBasedClustering* and *SizeBasedClustering* work by indexing the dataset using a single blocking key function. (The

remaining functions are iteratively employed in order to split large blocks.) On the other hand, our approach processes a dataset that is initially indexed by multiple blocking key functions and exploits block co-occurrence in order to split large blocks. Moreover, we perform merge operations based on the intersection of the blocks (instead of computing string similarity between blocking key values). Finally, we also propose heuristics for removing entities from blocks (*lECP*, *lBCP* and *LBSP*) and excluding blocks (*lBCE*), whereas in [12] the authors do not tackle these problems. To the best of our knowledge, we are the first to propose algorithms to control block sizes in the context of multiple blocking keys that are all initially employed in order to index a dataset.

7 Conclusions and future works

In this paper, we have investigated the problem of pruning a blocking collection indexed by multiple blocking key functions in order to control block sizes for entity resolution and maintain reasonable blocking quality results. For doing so, we have proposed a metaheuristic, named *MCBS*, which defines four generic steps for tackling the problem. We have also proposed heuristics that exploit block co-occurrence for shrinking (*lECP*, *lBCP* and *LBSP*), splitting (*CooSlicer*), merging (*MaxIntersectionMerge*) and excluding (*lBCE*) blocks. By employing the *MCBS* metaheuristic, we have combined the proposed heuristics in order to generate six different algorithms for controlling block sizes. We have evaluated these algorithms employing an exhaustive experimental design which employs four datasets and a variation of interval sizes, λ values and weighting schemes.

The results of the experiments have shown that the proposed algorithms are able to efficiently control block sizes as well as maintain reasonable blocking quality results. Regarding three of the evaluated datasets (Cora, DBLPM4 and FebrlData), the proposed heuristics have outperformed the baseline algorithm (SN with fixed window size), considering the harmonic mean between PC and RR. The *lECP* and *lBCE* algorithms have generated the best quality results (considering the harmonic mean between PC and RR) when compared to the other proposed algorithms. Also, the ARCS weighting scheme usually produces better quality results on small datasets, whereas the ECBS scheme generates better quality results on larger datasets. The double pruning approach (*lECP*+*lBCP*) produces quite low PC results (especially when low λ values are employed), whereas the *Default* algorithm generates the best PC results (at the cost of sacrificing RR results). In turn, the λ parameter has presented strong influence over the quality results and thus can be used to control the tradeoff between PC and RR. In other words, in a practical scenario, a user may decrease the employed λ value (< 0.3) to favor the PC result, whereas the λ value can be increased (> 0.3) to enhance the RR result, depending on the user requirements. Lastly, a user should choose either a *shrink* or *exclude* step to effectively prune the blocks based on the proposed approach without sacrificing blocking quality results significantly.

Regarding efficiency, the proposed heuristics have presented good scalability when processing the evaluated datasets. In general, the algorithms have spent the majority of time performing the *shrink* and *split* steps of the *MCBS* metaheuristic. In turn, the *MaxIntersectionMerge* and *lBCE* (heuristic for excluding blocks) algorithms have generated very efficient results in the conducted experiments. Also, in the majority of cases, the *lECP* heuristic presents better scalability than the other proposed pruning approaches (*lBCP* and *LBSP*) and the appliance of the ARCS scheme produces lower execution times than the ECBS scheme. Even though the SN algorithm has reported better efficiency results than the proposed heuris-

tics when dealing with three evaluated datasets, in the majority of cases the proposed heuristics have either outperformed all the efficacy results reported by SN or reported better PC results than the SN (with similar RR values).

For future work, we intend to extend the proposed approach to consider the semantics of the attribute values in order to shrink/merge/split/exclude blocks (instead of relying only on co-occurrence scores). Furthermore, we aim to propose other heuristics for tackling the generic steps of the *MCBS* metaheuristic to improve even more the obtained quality results (for instance, by automatically tuning the λ value, depending on the size of the block). We also intend to propose approaches for executing the evaluated algorithms in a distributed manner using consolidated frameworks, such as Hadoop and Spark. Finally, we intend to evaluate the proposed approach using different combinations of blocking key functions to index the datasets.

A Employed blocking keys

Let S be a string, $F_n(S)$ be the first n letters of S, $nB(S)$ be the first n bigrams [4] of S and $nT(S)$ be the first n trigrams [4] of S. Using this notation, in Table 13 we present the blocking key functions which have been used in order to index the datasets in the conducted experiments (Sect. 5). Since the entities in the CoraATDV dataset do not present a predefined schema, we have split the entities by white space. Each part of the split result is denoted as $SplitLine[k]$, such that k is the index of the resulting array.

Table 13 Employed blocking key functions

Dataset	Employed blocking keys
CoraATDV	$\{4B(SplitLine[0]), 4B(SplitLine[1]), 2B(SplitLine[2]), 2T(SplitLine[3])\}$
DBLPM4	$\{F_1(title), 4B(title)\}$
FebrlData	$\{1T(name), 1T(surname), 1T(address_1)\}$
SongsDataset	$\{1T(title), 1T(artist\_name)\}$

B Fixed window size for SN

Let S_{min} and S_{max} be two size parameters, D be a dataset, $\mathcal{F}$ be a set of blocking key functions and $\mathcal{B}$ be the blocking collection generated by indexing D using $\mathcal{F}$. Taking into account the problem presented in Definition 6, the pruned blocking collection $\mathcal{B}'_{max}$ that yields to the maximum aggregated cardinality is composed by $(|D| \cdot |\mathcal{F}| \text{ div } S_{max})$ blocks containing S_{max} entities and a block containing $(|D| \cdot |\mathcal{F}| \bmod S_{max})$ entities. Following Definition (1), we can calculate the aggregated cardinality generated by $||\mathcal{B}'_{max}||$ as shown in Eq. (5).

$$||\mathcal{B}'_{max}|| = |\mathcal{F}| \cdot 2^{-1} \cdot \Big((|D| \text{ div } S_{max}) \cdot S_{max} \cdot (S_{max} - 1) + \cdot (|D| \bmod S_{max}) \cdot ((|D| \bmod S_{max}) - 1) \Big) \tag{5}$$

Therefore, by employing a sorted neighborhood (SN) approach with a fixed window size equal to w over the blocks in $\mathcal{B}$, the number of generated comparisons cannot exceed $||\mathcal{B}'_{max}||$. Since the SN algorithm is executed $|\mathcal{F}|$ times (i.e., for each employed blocking key function), the number of comparisons generated by employing the SN method for each blocking key function cannot exceed $\frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|}$. Thus, we need to configure a fixed window size (w) that generates at most $\frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|}$ comparisons.

Since the number of comparisons generated by the SN algorithm can be theoretically estimated [5], we can calculate the maximum window size allowed (w_{max}) based on Eq. (6).

$$\begin{aligned}
&(w-1)\cdot(|D|-\frac{w}{2}) = \frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|}\\
&w\cdot|D| - \frac{w^2}{2} - |D| + \frac{w}{2} = \frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|}\\
&\quad -\frac{w^2}{2} + \frac{(2\cdot w\cdot|D|)+w}{2} - \frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|} - |D| = 0\\
&w^2 - (2\cdot|D|+1)\cdot w + 2\cdot\left(\frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|} + |D|\right) = 0
\end{aligned} \tag{6}$$

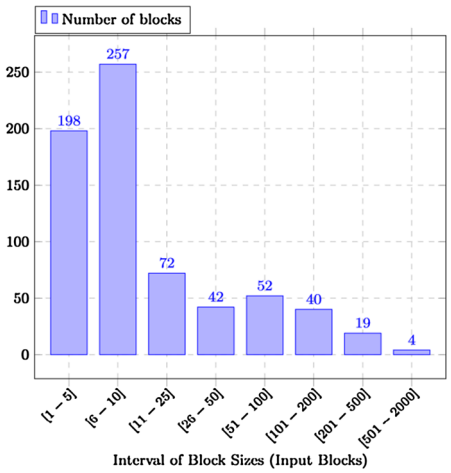
From Eq. (6), we conclude that w_{max} can be calculated as shown in Eq. (7), i.e., the maximum value of w that generates at most $\frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|}$ comparisons between entities.

$$w_{max} = \frac{(2\cdot|D|+1) + \sqrt{(-2\cdot|D|+1)^2 - 4\cdot 2\cdot\left(\frac{||\mathcal{B}'_{max}||}{|\mathcal{F}|} + |D|\right)}}{2} \tag{7}$$

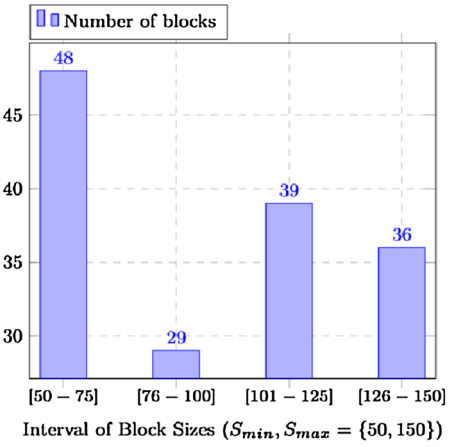
Note that, since our approach can perform *shrink* and *exclude* operations over the input blocking collection $\mathcal{B}$, we cannot calculate the minimum number of comparisons to be generated by the pruned blocking collection $\mathcal{B}'$. For this reason, for each dataset, we employ the SN method using two variations of the w_{max} (following Eq. (7)) result.

C Block size distribution

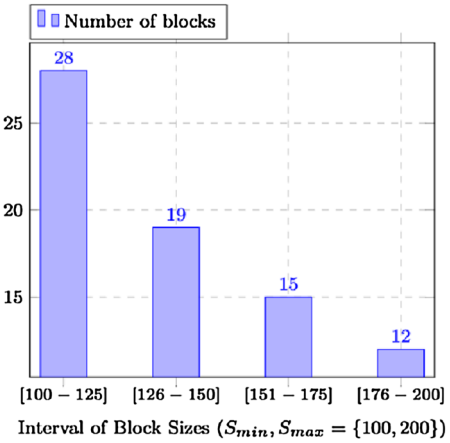
We present the distribution of block sizes in the input blocking collection of two datasets (*Cora* and *DBLPM4*) employed in the experiments and in the pruned blocking collection generated by one of the proposed algorithms (*lECP*—see Table 8), in Figs. 10 and 11.



(a) Distribution of the input block sizes on CoraATDV dataset.

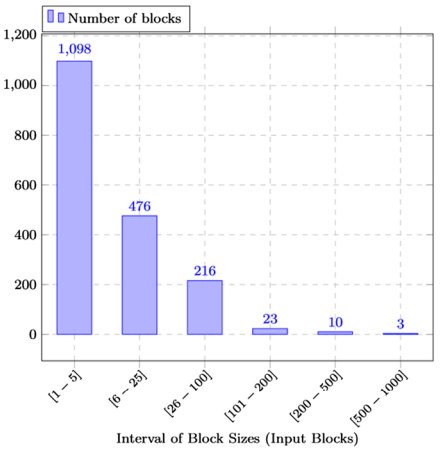


(b) Distribution of the block sizes on CoraATDV dataset after running the proposed approach ($S_{min} = 50, S_{max} = 150$).

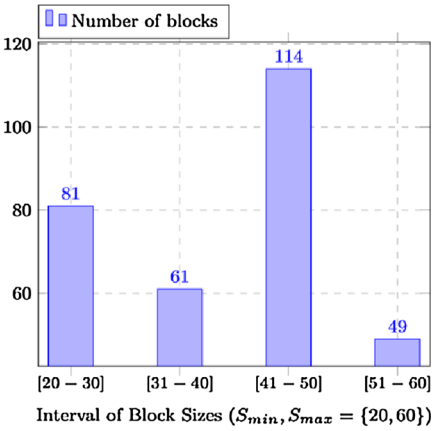


(c) Distribution of the block sizes on CoraATDV dataset after running the proposed approach ($S_{min} = 100, S_{max} = 200$).

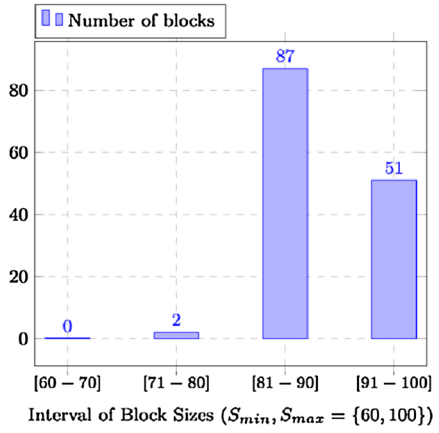
Fig. 10 Distribution of block sizes on CoraATDV dataset



(a) Distribution of the input block sizes on DBLPM4 dataset.



(b) Distribution of the block sizes on DBLPM4 dataset after running the proposed approach ($S_{min} = 20, S_{max} = 60$).



(c) Distribution of the block sizes on DBLPM4 dataset after running the proposed approach ($S_{min} = 60, S_{max} = 100$).

Fig. 11 Distribution of block sizes on DBLPM4 dataset

References

1. Batini C, Scannapieco M (2016) Data quality dimensions. Springer, Cham, pp 21–51
2. Batini C, Cappiello C, Francalanci C, Maurino A (2009) Methodologies for data quality assessment and improvement. ACM Comput Surv (CSUR) 41(3):16
3. Bilenko M, Kamath B, Mooney RJ (2006) Adaptive blocking: learning to scale up record linkage. In: Sixth international conference on data mining, ICDM'06. IEEE, pp 87–96
4. Christen P (2012) Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer, New York
5. Christen P (2012) A survey of indexing techniques for scalable record linkage and deduplication. IEEE Trans Knowl Data Eng 24(9):1537–1555
6. Cohen WW, Richman J (2002) Learning to match and cluster large high-dimensional data sets for data integration. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp 475–480

7. Costa G, Manco G, Ortale R (2010) An incremental clustering scheme for data de-duplication. Data Min Knowl Discov 20(1):152–187
8. Covell M, Baluja S (2009) Lsh banding for large-scale retrieval with memory and recall constraints. In: IEEE international conference on acoustics, speech and signal processing, ICASSP 2009. IEEE, pp 1865–1868
9. De Vries T, Ke H, Chawla S, Christen P (2009) Robust record linkage blocking using suffix arrays. In: Proceedings of the 18th ACM conference on Information and knowledge management. ACM, pp 305–314
10. do Nascimento DC, Pires CES, Mestre DG (2018) Heuristic-based approaches for speeding up incremental record linkage. J Syst Softw 137:335–354
11. Ebraheem M, Thirumuruganathan S, Joty S, Ouzzani M, Tang N (2018) Distributed representations of tuples for entity resolution. Proc VLDB Endow 11(11):1454–1467
12. Fisher J, Christen P, Wang Q, Rahm E (2015) A clustering-based framework to control block sizes for entity resolution. In: Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 279–288
13. Ganganath N, Cheng CT, Chi KT (2014) Data clustering with cluster size constraints using a modified *k*-means algorithm. In: 2014 International conference on cyber-enabled distributed computing and knowledge discovery (CyberC). IEEE, pp 158–161
14. Giraud-Carrier C, Goodliffe J, Jones BM, Cueva S (2015) Effective record linkage for mining campaign contribution data. Knowl Inf Syst 45(2):389–416
15. Gomes Mestre D, Pires CES (2013) Improving load balancing for mapreduce-based entity matching. In: 2013 IEEE symposium on computers and communications (ISCC). IEEE, pp 000618–000624
16. Gruenheid A, Dong XL, Srivastava D (2014) Incremental record linkage. Proc VLDB Endow 7(9):697–708
17. Hassanzadeh O, Chiang F, Lee HC, Miller RJ (2009) Framework for evaluating clustering algorithms in duplicate detection. Proc VLDB Endow 2(1):1282–1293
18. Kolb L, Thor A, Rahm E (2012) Multi-pass sorted neighborhood blocking with mapreduce. Comput Sci Res Dev 27(1):45–63
19. Koudas N, Sarawagi S, Srivastava D (2006) Record linkage: similarity measures and algorithms. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, pp 802–803
20. Malinen MI, Fränti P (2014) Balanced *k*-means for clustering. In: Joint IAPR international workshops on statistical techniques in pattern recognition (SPR) and structural and syntactic pattern recognition (SSPR). Springer, pp 32–41
21. Mann W, Augsten N, Bouros P (2016) An empirical evaluation of set similarity join techniques. Proc VLDB Endow 9(9):636–647
22. Mestre DG, Pires CE, Nascimento DC (2015) Adaptive sorted neighborhood blocking for entity matching with mapreduce. In: Proceedings of the 30th annual ACM symposium on applied computing. ACM, pp 981–987
23. Mestre DG, Pires CES, Nascimento DC (2017) Towards the efficient parallelization of multi-pass adaptive blocking for entity matching. J Parallel Distrib Comput 101:27–40
24. Michelson M, Knoblock CA (2006) Learning blocking schemes for record linkage. In: AAAI, pp 440–445
25. Nascimento DC, Pires CE, Mestre D (2015) Data quality monitoring of cloud databases based on data quality SLAs. In: Trovati M, Hill R, Anjum A, Zhu S, Liu L (eds) Big-data analytics and cloud computing. Springer, Cham, pp 3–20
26. Papadakis G, Koutrika G, Palpanas T, Nejdl W (2014) Meta-blocking: taking entity resolutionto the next level. IEEE Trans Knowl Data Eng 26(8):1946–1960
27. Papadakis G, Papastefanatos G, Koutrika G (2014) Supervised meta-blocking. Proc VLDB Endow 7(14):1929–1940
28. Papenbrock T, Heise A, Naumann F (2015) Progressive duplicate detection. IEEE Trans Knowl Data Eng 27(5):1316–1329
29. Ramadan B, Christen P, Liang H, Gayler RW (2015) Dynamic sorted neighborhood indexing for real-time entity resolution. J Data Inf Qual 6(4):15
30. Ranbaduge T, Vatsalan D, Christen P (2015) Clustering-based scalable indexing for multi-party privacy-preserving record linkage. In: Pacific-Asia conference on knowledge discovery and data mining. Springer, pp 549–561
31. Ranbaduge T, Vatsalan D, Christen P, Verykios V (2016) Hashing-based distributed multi-party blocking for privacy-preserving record linkage. In: Pacific-Asia conference on knowledge discovery and data mining. Springer, pp 415–427
32. Rebollo-Monedero D, Solé M, Nin J, Forné J (2013) A modification of the *k*-means method for quasi-unsupervised learning. Knowl Based Syst 37:176–185

33. Vatsalan D, Christen P, Verykios VS (2013) A taxonomy of privacy-preserving record linkage techniques. Inf Syst 38(6):946–969
34. Vatsalan D, Christen P (2013) Sorted nearest neighborhood clustering for efficient private blocking. In: Pacific-Asia conference on knowledge discovery and data mining. Springer, pp 341–352
35. Verykios VS, Karakasidis A, Mitrogiannis VK (2009) Privacy preserving record linkage approaches. Int J Data Min Model Manag 1(2):206–221
36. Whang SE, Menestrina D, Koutrika G, Theobald M, Garcia-Molina H (2009) Entity resolution with iterative blocking. In: Proceedings of the 2009 ACM SIGMOD international conference on management of data. ACM, pp 219–232
37. Whang SE, Marmaros D, Garcia-Molina H (2013) Pay-as-you-go entity resolution. IEEE Trans Knowl Data Eng 25(5):1111–1124
38. Yan S, Lee D, Kan MY, Giles LC (2007) Adaptive sorted neighborhood methods for efficient record linkage. In: Proceedings of the 7th ACM/IEEE-CS joint conference on digital libraries. ACM, pp 185–194
39. Zhu S, Wang D, Li T (2010) Data clustering with size constraints. Knowl Based Syst 23(8):883–889

Dimas Cassimiro Nascimento holds a PhD in Computer Science from Federal University of Campina Grande, Brazil. He is a Computer Science Professor at Federal Rural University of Pernambuco. His research interests include databases, data quality and artificial intelligence.

Carlos Eduardo Santos Pires holds a PhD in Computer Science from Universidade Federal de Pernambuco (Brazil). Since November 2009, he is a professor in Computer Science at the Computing and Systems Department of the Universidade Federal de Campina Grande (UFCG), where he currently collaborates with research in the area of Information Systems and Databases at the Data Quality Laboratory of UFCG. He has experience in computer science, with emphasis on databases, acting on the following topics: decision support systems, knowledge discovery, data quality, and information integration.

Demetrio Gomes Mestre holds a PhD in Computer Science from Federal University of Campina Grande, Brazil. His research interests include data quality, big data and cloud computing.