Comparative Analysis of Approximate Blocking Techniques for Entity Resolution

George Papadakis[$], Jonathan Svirsky[#], Avigdor Gal[#], Themis Palpanas[◇]

[$]Dep. of Informatics & Telecommunications, University of Athens, Greece gpapadis@di.uoa.gr
[#]Technion, Israel Institute of Technology js@tx.technion.ac.il, avigal@ie.technion.ac.il
[◇]Paris Descartes University, France themis@mi.parisdescartes.fr

ABSTRACT

Entity Resolution is a core task for merging data collections. Due to its quadratic complexity, it typically scales to large volumes of data through blocking: similar entities are clustered into blocks and pair-wise comparisons are executed only between co-occurring entities, at the cost of some missed matches. There are numerous blocking methods, and the aim of this work is to offer a comprehensive empirical survey, extending the dimensions of comparison beyond what is commonly available in the literature. We consider 17 state-of-the-art blocking methods and use 6 popular real datasets to examine the robustness of their internal configurations and their relative balance between effectiveness and time efficiency. We also investigate their scalability over a corpus of 7 established synthetic datasets that range from 10,000 to 2 million entities.

1. INTRODUCTION

The need for data integration stems from the heterogeneity of data arriving from multiple sources, the lack of sufficient semantics to fully understand data meaning, and errors originating from incorrect data insertion and modifications (*e.g.*, typos and eliminations) [5]. *Entity Resolution* (ER) aims at "cleaning" noisy data collections by identifying entity profiles, or simply entities, that represent the same real-world object. With a body of research that spans over multiple decades, ER has a wealth of formal models [7, 11], efficient and effective algorithmic solutions [18, 26], as well as a bulk of systems and benchmarks that allow for comparative analyses of solutions [4]. Elmagarmid et al. provide a comprehensive survey covering the complete deduplication process [6].

Exhaustive ER methods cannot scale to large volumes of data, due to their inherently quadratic complexity: in principle, each entity has to be compared with all others in order to find its matches. To improve their efficiency, approximate techniques are typically employed, sacrificing a small number of true matches in order to save a large part of the time-consuming comparisons. Among them, *blocking* is the most common and popular approach, clustering similar entities into blocks so that it suffices to execute comparisons only within the resulting blocks.

Proceedings of the VLDB Endowment, Vol. 9, No. 9

Christen's survey [5] examines the main blocking methods for structured data, concluding that they all require careful fine-tuning of internal parameters in order to yield high performance. Their most important parameter is the definition of *blocking keys*, *i.e.*, the signatures that are extracted from the entity profiles in order to facilitate their placement into blocks. Its configuration can be simplified to some extent through the unsupervised, schema-agnostic keys proposed in [24]: they achieve higher recall than the manually-defined, schema-based ones, at the cost of more comparisons and higher processing time. Yet, every method involves additional internal parameters that are crucial for their performance, but their effect has not been carefully studied in the literature.

In this context, our work aims to answer the following four questions: (1) *How easily can we configure the parameters of the main blocking methods?* (2) *Are there any default configurations that are expected to yield good performance in versatile settings?* (3) *How robust is the performance of every blocking method with respect to its internal parameters?* and (4) *Which blocking method offers the best balance between recall and precision in most cases?*

We start by introducing a novel taxonomy that relies on the lifecycle of blocks in order to elucidate the functionality of all types of blocking methods. Based on it, we analytically examine 17 state-of-the-art blocking methods, comparing their relative performance and scalability over 13 established benchmarks with widely different characteristics: their sizes range from few thousand entities to few million, they involve both real-world and synthetic data and they cover both structured (homogeneous) and semi-structured (heterogeneous) data. We performed an exhaustive set of experiments, considering all performance aspects of blocking: the effectiveness in terms of recall and precision as well as the time efficiency with respect to overhead and total running time.

Special care has been taken to examine the effect of the internal configuration on blocking methods. To this end, we assess the performance of each method under a wide range of plausible values for its internal parameters. We also define a formal approach for identifying the best configuration per blocking method and dataset. The same formalization can be used for inferring the settings that achieve the best average performance across all datasets, thus providing a default configuration per method. By comparing the best and the default parameters, we are able to assess the robustness of each blocking method.

Relation to Previous Work. The blocking techniques for ER [6, 9, 10, 12] can be distinguished in two broad categories: the *approximate* methods sacrifice recall to a minor extent in an effort to achieve significantly higher precision, whereas the *exact* methods guarantee that all pairs of duplicates are identified [17, 23]. More popular in the literature are methods of the former type, because

they are more flexible in reducing the number of executed comparisons. For this reason, we only consider approximate methods.

Our experimental analysis builds on two previous surveys of approximate blocking methods for structured data [5, 24]. We use all datasets and blocking methods that were examined in [24] and we combine them with the same schema-agnostic blocking keys. These cover all datasets and almost all techniques considered in [5]. Yet, we go beyond these works in the following seven ways: *(i)* Our study involves two large heterogeneous datasets, which were absent from previous studies, even though they are quite common in the era of Big Data. *(ii)* We consider three additional blocking methods, which capture more recent developments in blocking, especially the handling of heterogeneous data. *(iii)* We take into account the lifecycle of blocks, examining the performance of blocking methods when coupled with block processing techniques. *(iv)* We focus on the configuration of blocking methods, being the first to systematically examine a wide range of internal parameters for every method, to formally define its best and default configuration per dataset, and to assess its robustness. *(v)* Our emphasis on internal configurations enables the estimation of relative performance and scalability of the main blocking methods in a systematic way. This is impossible with prior works that consider a limited number of possible configurations. *(vi)* Our experiments suggest default configurations for all blocking methods, thus minimizing the effort required for using them. *(vii)* We introduce a novel taxonomy of blocking methods that emphasizes the lifecycle of blocks and the role of their internal parameters, thus providing useful insights into our experimental results.

Contributions. This paper makes the following contributions.

- We provide a novel categorization of blocking methods and their internal parameters based on the lifecycle of blocks, which leads to a better understanding of their behavior.
- We thoroughly compare 17 state-of-the-art blocking methods through experiments on 13 established benchmarks with different characteristics. All datasets, along with the testing code (in Java), are publicly available.[1]
- We formally define the best and the default configuration of blocking methods over a set of datasets.
- We report comprehensive findings from our experiments and provide new insights on the scalability and robustness of the main blocking methods. In this way, we guide practitioners on how to select the most suitable technique and its configuration for each data integration problem.

Paper Structure. Section 2 introduces the terminology and the measures used in the evaluation. We define our taxonomy of blocking methods in Section 3 and describe the state-of-the-art ones in Section 4. Section 5 presents the experimental setup, and Section 6 the results. We summarize the main findings in Section 7.

2. PRELIMINARIES

Entities constitute uniquely identified collections of name-value pairs that describe real-world objects. An entity with id i is denoted by e_i. A set of entities is termed an *entity collection* (E), and the number of entities it involves is called *collection size* ($|E|$). Two entities $\{e_i, e_j\} \subseteq E$ with $i \neq j$ are called *duplicates* or *matches* if they represent the same real-world object. Given one or more entity collections, ER aims to identify the duplicate entities they contain.

We distinguish ER into two main tasks, depending on the form of the input: *(i) Clean-Clean ER* receives as input two duplicate-free, but overlapping entity collections, E_1 and E_2, and tries to identify the entities they have in common, $D(E_1 \cap E_2)$. *(ii)* All other cases of ER are considered equivalent to *Dirty ER*, where the input comprises a single entity collection, E, that contains duplicates in itself. The goal is to partition E into a set of equivalence clusters, $D(E)$. In the context of homogeneous data (databases), the former task is called *Record Linkage* and the latter *Deduplication* [5].

ER involves an inherently quadratic complexity: the number of comparisons executed by the brute-force approach equals $\|E\| = |E_1| \times |E_2|$ for Clean-Clean ER and $\|E\| = |E| \times (|E|-1)/2$ for Dirty ER. In order to scale to large entity collections, we use blocking, which restricts the computational cost to comparisons between similar entities: it clusters them into a set of blocks B, called *block collection*, and performs comparisons only among the co-occurring entities.

A block with id i is denoted by b_i. Its *size* ($|b_i|$) refers to the number of its elements, while its *cardinality* ($\|b_i\|$) refers to the number of comparisons it contains: $\|b_i\| = |b_i| \cdot (|b_i|-1)/2$ for Dirty ER and $\|b_i\| = |b_{i,1}| \cdot |b_{i,2}|$ for Clean-Clean ER, where $b_{i,1} \subseteq E_1$ and $b_{i,2} \subseteq E_2$ are the disjoint, inner blocks of b_i. The total number of pairwise comparisons in B is called *total cardinality* ($\|B\|$) and is equal to the sum of the individual block cardinalities: $\|B\| = \sum_{b_i \in B} \|b_i\|$.

Typically, two duplicate entities are considered *detected* as long as they co-occur in at least one block – independently of the selected entity matching technique [2, 5, 22, 26]. We actually follow the best practice in the literature and consider entity matching as an orthogonal task to blocking [5, 24, 27]. Provided that the vast majority of duplicate entities are co-occurring, the performance of ER depends on the accuracy of the method that is used for entity comparison. The set of co-occurring duplicate entities is denoted by $D(B)$, while $|D(B)|$ denotes its size.

Effectiveness Measures. Three measures are used for estimating the *effectiveness* of a block collection B [5, 24]:

(E1) *Pairs Completeness* (PC) assesses the portion of the duplicate entities that co-occur at least once in B (also known as *recall* in other research fields). Formally, $PC(B, E) = |D(B)|/|D(E)|$ for Dirty ER and $PC(B, E_1, E_2) = |D(B)|/|D(E_1 \cap E_2)|$ for Clean-Clean ER.

(E2) *Pairs Quality* (PQ) corresponds to precision, as it estimates the portion of non-redundant comparisons that involve matching entities. Formally, $PQ(B) = |D(B)|/\|B\|$.

(E3) *Reduction Ratio* (RR) estimates the portion of comparisons that are avoided in B with respect to the naive, brute-force approach. Formally, $RR(B, E) = 1 - \|B\|/\|E\|$.

All these measures take values in the interval $[0, 1]$, with higher values indicating higher effectiveness. Ideally, the goal of blocking is to maximize all of them at once, identifying all existing duplicates with the minimum number of comparisons. However, a linear increase in $|D(B)|$ typically requires a quadratic increase in $\|B\|$ [12], thus yielding a trade-off between PC and PQ-RR. Therefore, blocking should aim for a balance between these measures, minimizing the executed comparisons, while ensuring that most matching entities are co-occurring ($PC > 0.80$) [24].

Efficiency Measures. We use two measures to estimate the *time efficiency* of the blocking process [24]:

(T1) *Overhead Time* ($OTime$) is the time between receiving the input entity collection(s) and returning the set of blocks as output.

(T2) *Resolution Time* ($RTime$) adds to $OTime$ the time required to perform all pair-wise comparisons in B with an entity matching technique. Following [24], we use the Jaccard similarity of all tokens in all attribute values for this purpose, without involving any optimization for higher efficiency (*e.g.*, inverted indices). This is a generic approach that applies to all contexts, unlike more advanced techniques that are usually specialized for a particular domain, or for a particular type of data, *e.g.*, structured data [19].

For both measures, lower values indicate higher time efficiency.

[1] See `http://sourceforge.net/projects/erframework` and `https://bitbucket.org/TERF/mfib`.

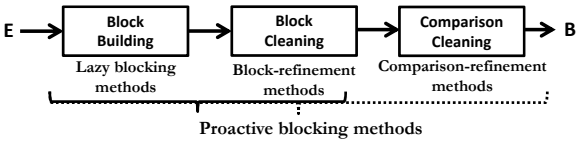

Figure 1: The sub-tasks of blocking and the respective methods.

3. BLOCKING METHODS TAXONOMY

We now introduce a taxonomy of blocking methods to facilitate their understanding, use and combination. Our taxonomy is based on the internal functionality of blocking, which we divide into three sub-tasks (Figure 1 presents them in the order of execution):

Block Building (*BlBu*) takes as input one or two entity collections and clusters them into blocks. Typically, each entity is represented by multiple blocking keys that are determined a-priori, except for MFIBlocks [18] , where the keys are the result of mining; blocks are then created based on the similarity, or equality of these keys. As an illustrating example, consider Figure 2, which demonstrates the functionality of Standard Blocking [5] when using the unsupervised, schema-agnostic keys proposed in [24]. All attribute values in Figure 2(a) contribute to the blocking keys: for every one of their tokens, a separate block is created with all associated entities, as shown in Figure 2(b). The internal parameters of *BlBu* may fit into one of two types: *(i) key definition* parameters pertain to the specification of blocking keys, *e.g.*, how to extract keys from an attribute value; *(ii) key use* parameters determine the measures that support the creation of blocks from a set of existing keys, *e.g.*, the minimum similarity threshold for constructing a new block.

Block Cleaning (*BlCl*) receives as input a block collection and removes blocks that primarily contain unnecessary comparisons. These may come in one of two forms: *redundant comparisons* are repeated comparisons across different blocks, and *superfluous comparisons* involve non-matching entities. While the former are easy to detect, the latter can only be estimated during entity matching, in the final step of ER. In the example of Figure 2(b), e_1 and e_3 are first compared in block b_1 and, thus, their comparison in block b_2 is redundant, since there is no gain in recall if we repeat it; in addition, the non-redundant comparison between e_1 and e_4 in block b_3 is superfluous, because there is no gain in recall if we execute it. In this context, *BlCl* aims to discard both redundant and superfluous comparisons at a limited cost in recall. Its internal parameters enforce either *block-level* constraints, affecting individual blocks (*e.g.*, maximum entities per block), or *entity-level* constraints, affecting individual entities (*e.g.*, maximum blocks per entity).

Comparison Cleaning (*CoCl*) takes as input a block collection and outputs a set of executable pairwise comparisons. Similar to *BlCl*, its goal is to discard redundant and superfluous comparisons at a small cost in recall. Instead of targeting entire blocks, though, it operates at a finer level of granularity, targeting individual comparisons. Hence, its internal parameters are partitioned into those relating to *redundant comparisons* and to *superfluous comparisons*.

Based on these three sub-tasks, we identify four categories of blocking methods, which are depicted in Figure 1:

(i) **Lazy blocking methods** involve a coarse functionality that solely employs *BlBu*.

(ii) **Block-refinement methods** exclusively perform *BlCl*.

(iii) **Comparison-refinement methods** apply only *CoCl*.

(iv) **Proactive blocking methods** involve a fine-grained functionality that aims to create self-contained blocks. To this end, they either perform *BlBu+BlCl*, or all three, *BlBu+BlCl+CoCl*.

Table 1 lists the blocking methods we test in this work, partitioned into these four categories. For each method, we also map its internal parameters to their type.

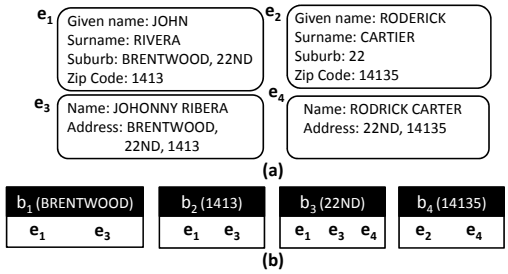

Figure 2: (a) A set of entities, (b) creating a block for every token that appears in the attribute values of at least two entities.

4. BLOCKING METHODS

We now briefly present the 17 blocking techniques we consider in our experimental analysis: 5 lazy, 7 proactive, 2 block- and 3 comparison-refinement methods. Emphasis is placed on their internal parameters and the sub-tasks they run. For the lazy and proactive methods, we exclusively consider the schema-agnostic configuration of their blocking keys that was examined in [24]; this approach simplifies their configuration, enhances their recall, and turns them applicable to heterogeneous entity collections. Based on this configuration, Figure 3 outlines the relationships among these methods; an edge $A \rightarrow B$ indicates that method B improves on method A either by modifying the definition of its blocking keys, or by changing the way they are used for the creation of blocks.

4.1 Lazy Blocking Methods

Standard Blocking (StBl) [5, 26] has a parameter-free functionality: every distinct token t_i in the input creates a separate block b_i that contains all entities having t_i in their attribute values as long as t_i is shared by at least 2 entities (Figure 2). StBl serves as the starting point for the other lazy and proactive methods (see below), which rely on its schema-agnostic, unsupervised blocking keys for creating their blocks.

Attribute Clustering (ACl) [26] partitions attribute names into a set K of non-overlapping clusters according to the similarity of their values, and applies StBl independently inside each cluster. Every token t_i of the values in a cluster $k \in K$ creates a block with all entities that have t_i assigned to an attribute name belonging to k. A token t_i associated with n attribute names from $m(\leq n)$ attribute clusters, creates at most m blocks, while for StBl, the same token creates a single block. Thus, ACl aims to offer the same recall as StBl for significantly fewer comparisons. Its functionality is configured by the representation model of the attribute name textual values and the corresponding similarity measure. In our experiments, we consider 12 such models: *character n-grams* (CnG), with $n \in \{2, 3, 4\}$, *character n-gram graphs* [13] (CnGG), with $n \in \{2, 3, 4\}$, *token n-grams* (TnG), with $n \in \{1, 2, 3\}$, and *token n-gram graphs* [13] (TnGG), with $n \in \{1, 2, 3\}$; CnG is coupled with the Jaccard similarity, TnG with the cosine similarity, and the graph models with the graph value similarity metric [13].

Extended Sorted Neighborhood (ESoNe) [5] sorts the blocking keys of StBl in alphabetical order and slides a window of size w over them. In every iteration, it creates a new block for all keys that are placed within the current window. In the example of Figure 2, w=3 creates two blocks: the first with all entities associated to keys 1413, 14135, 22*ND* (each entity appears at most once): $b_1 = \{e_1, e_2, e_3, e_4\}$; and the second with all entities associated to keys 14135, 22*ND*, *BRENTWOOD* (its content is identical to b_1).

Q-grams Blocking (QGBl) [15] transforms the blocking keys of StBl into a format more resilient to noise: it converts every token into sub-sequences of q characters (q-grams) and builds blocks on

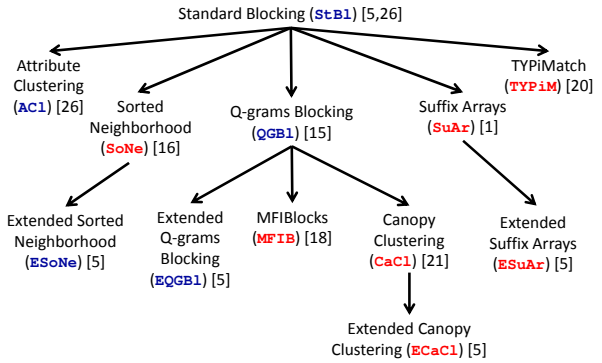

Figure 3: The relations between lazy and proactive methods.

their equality. For example, using q=3, *BRENTWOOD* is transformed into the keys *BRE*, *REN*, *ENT*, *NTW*, *TWO*, *WOO* and *OOD*. This is applied to all tokens of all entity values, and a block is created for every q-gram that appears in at least two entities.

Extended Q-grams Blocking (EQGBl) [5] aims to increase the discriminativeness of the blocking keys of QGBl so as to reduce the cardinality of its blocks. Instead of individual q-grams, it uses keys that stem from the concatenation of at least L q-grams. L is derived from a user-defined threshold $T \in [0, 1)$: $L = max(1, \lfloor k \cdot T \rfloor)$, where k is the number of q-grams in the original blocking key (token). The larger T is, the larger L gets, yielding less keys from the k q-grams. In our example, the key *BRENTWOOD* is transformed into the following combinations for T=0.9 and L=6: *BRERENENTNTWWOOOOD*, *BRERENENTTWOWOOOOD*, *BRERENNTWTWOWOOOOD*, *BREENTNTWTWOWOOOOD*, *BRERENENTNTWTWOWOOOOD*, *BRERENENTNTWTWOWOO*, *RENENTNTWTWOWOOOOD*, *BRERENENTNTWTWOOOD*. This is applied to the tokens from all attribute values of all entities, creating a block for every key that appears in at least two entities.

4.2 Block-refinement Methods

Block Purging (BlPu) sets an upper limit either on the size [8], or on the cardinality [26] of individual blocks and purges those exceeding it. We employ a simplified version that specifies the maximum size of retained blocks through a ratio $\rho \in [0, 1]$ of the input collection size: $|b|_{max} = \rho \times |E|$ for Dirty ER and $|b|_{max} = \rho \times (|E_1| + |E_2|)$ for Clean-Clean ER.

Block Filtering (BlFi) [28] operates on individual entities with the goal of removing them from their least important blocks. At its core lies the assumption that the larger a block is, the less important it is for its entities. Hence, BlFi first sorts all blocks globally, in ascending order of cardinality. Then, it retains every entity e_i in the N_i smallest blocks in which it appears. This threshold is locally defined as $N_i = \lfloor r \times |B_i| \rfloor$, where B_i stands for the set of blocks containing e_i and $r \in [0, 1]$ specifies a percentage.

4.3 Comparison-refinement Methods

Comparison Propagation (CoPr) [25] eliminates all redundant comparisons from any block collection, without missing any detected duplicates. A comparison e_i-e_j in block b_k is redundant and, thus, omitted if k differs from the least common block id of e_i and e_j. Hence, every pair of co-occurring entities is exclusively compared in the first block they share. This is a parameter-free procedure that scales well to billions of comparisons.

Iterative Blocking (ItBl) [29] is a parameter-free method that discards redundant comparisons between matching entities, while attempting to detect more duplicates. Blocks are placed in a queue and processed one at a time. For each block b_k, ItBl executes all comparisons in b_k and for every new pair of detected duplicates

Aggregate Reciprocal Comparisons Scheme (**ARCS**)	$ARCS(e_i, e_j, B) = \sum_{b_k \in B_{ij}} \frac{1}{\|b_k\|}$
Common Blocks Scheme (**CBS**)	$CBS(e_i, e_j, B) = \|B_{ij}\|$
Enhanced Common Blocks Scheme (**ECBS**)	$ECBS(e_i, e_j, B) = CBS(e_i, e_j, B) \cdot \log\frac{\|B\|}{\|B_i\|} \cdot \log\frac{\|B\|}{\|B_j\|}$
Jaccard Scheme (**JS**)	$JS(e_i, e_j, B) = \frac{\|B_{ij}\|}{\|B_i\| + \|B_j\| - \|B_{ij}\|}$
Enhanced Jaccard Scheme (**EJS**)	$EJS(e_i, e_j, B) = JS(e_i, e_j, B) \cdot \log\frac{\|V_B\|}{\|v_i\|} \cdot \log\frac{\|V_B\|}{\|v_j\|}$

Figure 4: The formal definitions of MeBl weighting schemes.

e_i-e_j, it merges their profiles and updates their representation in all blocks that contain them. The blocks that involve e_i or e_j, but have already been processed, are pushed back to the queue. This approach relies on entity matching and applies only to Dirty ER.

Meta-blocking (MeBl) [27] targets redundant and superfluous comparisons in *redundancy-positive* block collections, where the more blocks two entities share, the more likely they are to match. It creates an undirected graph, where the nodes correspond to entities and the edges connect the co-occurring entities. This automatically eliminates all redundant comparisons. To discard superfluous comparisons, edge weights are set in proportion to the similarity of the blocks shared by the adjacent entities: high weights indicate adjacent entities that are more likely to match, while edges with low weights indicate superfluous comparisons that should be pruned. Two parameters determine this procedure. The first is the weight assignment scheme. Figure 4 defines the five available generic schemes, where $B_{i,j}$ denotes the set of blocks shared by e_i and e_j, $|V_B|$ is the total number of nodes in the blocking graph of B, and $|v_i|$ is the node degree corresponding to e_i. The second parameter is the pruning algorithm. Six algorithms have been proposed [27, 28]: *(i) Cardinality Edge Pruning* (CEP) retains the top-K weighted edges of the graph. *(ii) Cardinality Node Pruning* (CNP) retains the top-k weighted edges from the neighborhood of each node. *(iii) Reciprocal Cardinality Node Pruning* (ReCNP) retains the edges that are among the top-k weighted ones for both adjacent entities. *(iv) Weight Edge Pruning* (WEP) iterates over all edges and discards those with a weight lower than a global threshold. *(v) Weight Node Pruning* (WNP) iterates over all nodes and prunes the adjacent edges that are weighted lower than a local threshold. *(vi) Reciprocal Weight Node Pruning* (ReWNP) retains the edges that exceed the local threshold of both adjacent node neighborhoods. The pruning threshold of each algorithm is set automatically.

4.4 Proactive Blocking Methods

MFIBlocks (MFIB) [18] uses the blocking keys of QGBl to create blocks by iteratively applying an algorithm for mining Maximal Frequent Itemsets [14]. MFIB receives as input a set of minimum support values (S), where every $minsup_i \in S$ is equal to the largest expected equivalence cluster in the input entities in iteration i. MFIB is an iterative algorithm, whose goal is to find a set of blocks of maximum cardinality, where each block satisfies the sparse neighborhood constraint (p) [3] and maintains a score above a threshold t, which is adjusted dynamically. In each iteration, the MFI mining algorithm is run on the entities that have not been processed yet, using a smaller minimum support (until $minsup_i$<2). Each MFI run creates a set of blocks, calculating a block score with a variation of the Jaccard coefficient that is extended to operate on q-grams [18]. Only blocks with support size smaller than $minsup_i \cdot p$ are kept (*BlCl*). Next, distinct pairs in the retained blocks are recorded. If the sparse neighborhood criterion is violated for some entity in the new blocks, t is updated. The final value of t is used to discard entity pairs with a low score (*CoCl*).

	Block Building (*BlBu*)		Block Cleaning (*BlCl*)		Comparison Cleaning (*CoCl*)		Step	Number of values	Number of settings
	key definition	key use	block level	entity level	redundant comparisons	superfluous comparisons			
StBl	*inherent*						parameter-free		
ACl		representation model					-	12	12
ESoNe		$w \in [2, 100]$					1	99	99
QGBl	$q \in [2, 6]$						1	5	5
EQGBl	$q \in [2, 6]$ $t \in [0.8, 1.0)$						1 0.05	5 4	20
(a) Lazy blocking methods									
BlPu			$\rho \in [0.05, 1.0]$				0.05	19	19
BlFi				$r \in [0.05, 1.0]$			0.05	19	19
(b) Block-refinement methods									
CoPr					*inherent*		parameter-free		
ItBl					(*inherent*)		parameter-free		
MeBl					inherent	weighting scheme pruning algorithm	- -	5 6	30
(c) Comparison-refinement methods									
MFIB	$q \in [2, 6]$	$S \in [2, 500]$	$t \in [0, 1]$		$p \in [2, 1000]$		1 varying 0.05 varying	5 25 20 15	1,500
CaCl	$q \in [2, 6]$	$w_1 \in [0.05, 1.0)$ $w_2 \in [w_1{+}0.05, 1.0)$		*inherent*	(*inherent*)		1 0.05 0.05	5 19 ≤18	855
ECaCl	$q \in [2, 6]$	$n_1 \in [1, 10]$	$n_2 \in [n_1, 100]$		(*inherent*)		1 1 1	5 10 ≤100	4,775
SuAr	$l_m \in [2, 6]$		$b_M \in (1, 100]$				1 1	5 99	495
ESuAr	$l_m \in [2, 6]$		$b_M \in (1, 100]$				1 1	5 99	495
SoNe			$w \in [2, 100]$				1	99	99
TYPiM		$\epsilon \in [0.05, 1.0]$ $\theta \in [0.05, 1.0]$					0.05 0.05	19 19	361
(d) Proactive blocking methods									

Table 1: Taxonomy of the blocking methods and their internal parameters; *inherent* denotes a parameter-free functionality of a specific category, while (*inherent*) indicates that this parameter-free functionality is performed with certain limitations. For each parameter, we present the values that are used in our experiments along with the total number of configurations per method.

The candidate pairs in the final set of blocks are recorded and their entities are excluded from further processing.

Canopy Clustering (CaCl) [21] inserts all input entities in a pool P, iteratively removes a random *seed* entity e_i from P, and creates a block with all entities still in P that have a Jaccard similarity with e_i higher than weight threshold $w_1 \in (0, 1)$. The most similar entities, which exceed a threshold $w_2 \in (0, 1) (> w_1)$, are removed from P. The Jaccard similarity is derived from the keys that QGBl assigns to every entity [5]. For Clean-Clean ER, CaCl employs two pools: P_1 contains the entities of E_1 and P_2 those of E_2. The seeds for the blocks are only selected from P_1, while their profile similarities are only computed with entities from P_2. Thus, the resulting blocks contain no redundant comparisons (*CoCl*). Given that CaCl creates at most one block per entity, it performs *BlCl*, as well.

Extended Canopy Clustering (ECaCl) [5] extends CaCl so as to ensure that every entity is placed in at least one block (CaCl will not place entities in a block if w_1 is too large). ECaCl performs *BlCl* based on the nearest neighbors of each entity: for each random seed entity e_i, it creates a sorted stack with the n_1 most similar entities still in P. These entities are placed in a new block together with e_i, while the $0 < n_2 (\leq n_1)$ most similar entities are removed from P. In the case of Clean-Clean ER, it employs two pools, performing *CoCl*, just like CaCl.

Suffix Arrays (SuAr) [1] improves StBl by enhancing the noise-tolerance of its blocking keys. Instead of using the entire tokens, it transforms them into the suffixes that are longer than a minimum length l_m. In our example, the key *BRENTWOOD* is converted into the suffixes *BRENTWOOD*, *RENTWOOD*, *ENTWOOD* and *NTWOOD* for l_m=6. This transformation applies to all tokens from all attribute values of the input entities. Blocks are then created based on the equality of keys. SuAr also performs *BlCl*, by discarding the suffixes that appear in more than b_M entities.

Extended Suffix Arrays (ESuAr) [5] uses noise reduction to make SuAr more robust. Instead of considering only suffixes, it transforms every key of StBl into all substrings that are longer than l_m characters. Continuing our example, *BRENTWOOD* is transformed into the keys (for $l_m = 6$): *BRENTWOOD*, *BRENTWOO*, *RENTWOOD*, *BRENTWO*, *ENTWOOD*, *RENTWOO*, *BRENTW*, *ENTWOO*, *NTWOOD*, *RENTWO*. This transformation is applied to all tokens from all attribute values of the input entities, while b_M sets an upper limit on the size of the resulting blocks (*BlCl*).

Sorted Neighborhood (SoNe) [16] sorts the blocking keys of StBl alphabetically and orders the corresponding entities accordingly. Then, it slides a window of size w over the list of ordered entities, and compares the entities within the same window. As a result, it only produces blocks with w entities, inherently performing *BlCl*. In the example of Figure 2, the sorted list of keys would be {1413, 14135, 22*ND*, *BRENTWOOD*}, and that of entities $\{e_1, e_3, e_2, e_4, e_1, e_3, e_4, e_1, e_3\}$; for w=3, the first three iterations yield the blocks $b_1 = \{e_1, e_3, e_2\}$, $b_2 = \{e_3, e_2, e_4\}$, $b_3 = \{e_2, e_4, e_1\}$ (for simplicity, entities with the same key were sorted by their id).

TYPiMatch (TYPiM) [20] classifies entities of heterogeneous data collections to different, possibly overlapping types: *e.g.*, the products in Google Base can be distinguished into computers, cameras, *etc.* TYPiM first extracts all tokens from all attribute values, creating a co-occurrence graph. Every node corresponds to a token and every edge connects two tokens if both conditional probabilities of co-occurrence exceed a threshold $\theta \in (0, 1)$. Next, it extracts maximal cliques from the co-occurrence graph and merges them if their overlap exceeds a threshold $\epsilon \in (0, 1)$. The resulting clusters

	|E|	|D(E)|	|N|	|P|	$\|\bar{p}\|$	||E||	RT(E)
D_{cens}	841	344	5	3,913	4.65	$3.53{\cdot}10^5$	2 sec
D_{rest}	864	112	5	4,319	5.00	$3.73{\cdot}10^5$	4 sec
D_{cora}	1,295	17,184	12	7,166	5.53	$8.38{\cdot}10^5$	19 sec
D_{cddb}	9,763	299	106	183,072	18.75	$4.77{\cdot}10^7$	39 min
D_{mvs}	27,615	22,866	4	155,436	5.63	$6.40{\cdot}10^8$	9 hrs
	23,182		7	816,012	35.20		
D_{dbp}	$1.2{\cdot}10^6$	892,586	30,688	$1.7{\cdot}10^7$	14.19	$2.58{\cdot}10^{12}$	~4 years
	$2.2{\cdot}10^6$		52,489	$3.5{\cdot}10^7$	16.18		

(a) Real datasets

D_{10K}	10,000	8,705	12	106,108	10.61	$5.00{\cdot}10^7$	12 min
D_{50K}	50,000	43,071	12	530,854	10.62	$1.25{\cdot}10^9$	5 hrs
D_{100K}	100,000	85,497	12	$1.1{\cdot}10^6$	10.61	$5.00{\cdot}10^9$	20 hrs
D_{200K}	200,000	172,403	12	$2.1{\cdot}10^6$	10.62	$2.00{\cdot}10^{10}$	~3 days
D_{300K}	300,000	257,034	12	$3.2{\cdot}10^6$	10.62	$4.50{\cdot}10^{10}$	~7 days
D_{1M}	$1.0{\cdot}10^6$	857,538	12	$1.1{\cdot}10^7$	10.62	$5.00{\cdot}10^{11}$	~81 days
D_{2M}	$2.0{\cdot}10^6$	$1.7{\cdot}10^6$	12	$2.1{\cdot}10^7$	10.62	$2.00{\cdot}10^{12}$	~1 year

(b) Synthetic datasets

Table 2: The datasets used in our experiments, ordered by size.

of tokens indicate different entity types, which may overlap. Every entity participates in all types to which its tokens belong. Finally, it applies StBl independently inside each cluster, excluding keys with high frequency (*e.g.*, stop-words) to avoid oversized blocks (*BlCl*).

5. EXPERIMENTAL SETUP

All methods were implemented in Java 8 and ran on a server with Intel Xeon E5620 (2.4GHz, 16 cores), 64GB RAM and Ubuntu 12.04. We repeated the experiments 10 times and report the mean values for *OTime* and *RTime*; both measures disregard the time spent on method configuration.

Datasets. Our empirical study involves 13 data collections of various sizes that cover both Clean-Clean and Dirty ER. Six of them contain real-world data, while the rest are synthetic. Their technical characteristics are presented in Table 2. $|E|$ denotes the number of input entities, $|D(E)|$ the number of duplicate pairs, $|N|$ the number of distinct attribute names, $|P|$ the total number of name-value pairs, $|\bar{p}|$ the average number of name-value pairs per entity, $||E||$ the number of comparisons executed by the brute-force approach, and $RT(E)$ the respective resolution time. Note that for D_{dbp} and the four larger synthetic datasets, $RT(E)$ was estimated using the average time required for executing 10^8 random pairwise comparisons in each dataset (0.045 and 0.014 msecs, respectively).

The real-world datasets have been widely used in the literature [5, 18, 24, 26, 27]. The four smaller ones involve homogeneous, Dirty ER data: D_{cens} contains records from US Census Bureau, D_{rest} from the Fodor and Zagat restaurant guides, D_{cora} from machine learning publications, and D_{cddb} from random audio CDs of freeDB (`http://www.freedb.org`). The two larger datasets involve heterogeneous, Clean-Clean ER data: D_{mvs} involves movies from IMDB and DBPedia, while D_{dbp} contains entities from the Infoboxes in English DBPedia, versions 3.0rc and 3.4.

The synthetic datasets constitute established benchmarks [24] that were generated with Febrl [4] using standard parameters [5]. First, duplicate-free entities were extracted from frequency tables for real names (given and surname) and addresses (street number, name, postcode, suburb, and state names). Then, duplicates were randomly created based on real error characteristics and modifications (*e.g.*, inserting, deleting or substituting characters or words). The resulting structured, Dirty ER datasets contain 40% duplicate entities with up to 9 duplicates per entity, no more than 3 modifications per attribute value, and up to 10 modifications per entity.

Parameter Tuning. We applied the blocking methods to all real datasets using a wide range of meaningful internal parameter configurations, as summarized in Table 1. The maximum value for the key definition parameters q, l_m (*BlBu*) was set to 6, because the tokens' average size in our datasets is 6.8. Therefore, larger values would incur no change in the keys of StBl. The maximum value for the block level parameters n_2, w, b_M (*BlCl*) was set to 100 so as to restrict the total number of configurations per method to manageable levels (see the last column of Table 1). Among all configurations, we only present the performance of the best and the default ones per method and dataset.

A *best configuration* achieves the optimal balance between recall and precision. More formally, the best configuration for *BlBu* over an input entity collection E is the one producing the block collection B that maximizes the measure $\boldsymbol{\alpha(B,E) = RR(B,E) \times PC(B,E)}$. For *BlCl* and *CoCl*, α is defined by replacing B with the output block collection B' and E with the input block collection B. The rationale behind this definition is that the fewer comparisons are contained in the output blocks, the higher α gets. To ensure that only redundant and superfluous comparisons are discarded, there is a discount for reducing recall: the lower PC gets, the lower α is for the specific configuration. Thus, the best configuration allows for examining whether a blocking method achieves a sufficiently high recall, while reducing the executed comparisons to a significant extent. A blocking method that fails to place the vast majority of matching entities in at least one common block is of limited use, even if it minimizes the number of pair-wise comparisons.

Finally, the *default configuration* corresponds to the setting that achieves the highest average α across all real datasets. Its purpose is to offer a realistic setup in the absence of ground-truth for fine-tuning, and to assess the robustness of a blocking method with respect to its internal parameters: the closer the best and the default configurations are, the more robust the blocking method is.

6. EMPIRICAL EVALUATION

At the core of our empirical analyses lies the comparison between lazy and proactive blocking methods. To compare them on an equal basis, we consider the performance of their entire blocking workflow, from the input entity collection(s) to *CoCl*. For the lazy methods, we analytically present the performance of the three blocking sub-tasks, while for the proactive ones, we consider two sub-tasks: the creation of blocks, *BlBu*+*BlCl*, and *CoCl*. Note that the best and default configuration of each sub-task depends on the best performance of the previous one, *e.g.*, the configuration of *BlCl* is based on the output of the best configuration for *BlBu*. For simplicity, we consider a single method per sub-task, even if a combination is possible (*e.g.*, using both BlPu and BlFi).

Section 6.1 checks the robustness of blocking methods with respect to their internal configuration. Section 6.2 examines their (relative) effectiveness. Section 6.3 investigates their (relative) time efficiency, and Section 6.4 assesses their scalability, using the synthetic datasets. All other experiments use the real-world datasets.

6.1 Robustness of Internal Configuration

Table 3 presents the best and the default configurations of the lazy and proactive methods across all real datasets. Note that CaCl, ECaCl, MFIB and TYPiM do not scale to D_{dbp}.

Starting with the lazy methods, we observe that most of them have a stable configuration for *BlBu* across all datasets. The only exception is ACl, which exhibits a different configuration for almost every dataset. Still, the actual difference in the performance of its representation models is insignificant, unless there is a large diversity of attribute names, as in D_{dbp} (see Figures 5 and 6).

Both *BlCl* and *CoCl* are sensitive to parameter settings. For *BlCl*, BlPu appears in just 4 out of the 35 cases, with none of

		Default	D_{cens}	D_{rest}	D_{cora}	D_{cddb}	D_{mvs}	D_{dbp}
StBl	*BlBu*	parameter-free	parameter-free	parameter-free	parameter-free	parameter-free	parameter-free	parameter-free
	BlCl	BlFi(r=0.55)	BlFi(r=0.55)	BlFi(r=0.15)	BlFi(r=0.45)	BlFi(r=0.25)	BlFi(r=0.50)	BlFi(r=0.55)
	CoCl	WEP(*CBS*)	WEP(*ARCS*)	ReCNP(*CBS*)	WEP(*ECBS*)	WEP(*CBS*)	ReCNP(*ECBS*)	ReCNP(*ARCS*)
ACl	*BlBu*	*T1GG*	*C3G*	*T2G*	*C2G*	*T3GG*	*T2GG*	*T2GG*
	BlCl	BlFi(r=0.50)	BlFi(r=0.55)	BlFi(r=0.15)	BlPu(ρ=0.10)	BlFi(r=0.25)	BlFi(r=0.45)	BlFi(r=0.40)
	CoCl	WEP(*CBS*)	CEP(*ARCS*)	ReCNP(*CBS*)	WEP(*EJS*)	WEP(*CBS*)	ReCNP(*ECBS*)	CNP(*ARCS*)
ESoNe	*BlBu*	w=2	w=2	w=2	w=2	w=2	w=2	w=2
	BlCl	BlFi(r=0.45)	BlFi(r=0.45)	BlFi(r=0.25)	BlFi(r=0.40)	BlFi(r=0.25)	BlFi(r=0.45)	BlFi(r=0.45)
	CoCl	WEP(*JS*)	ReWNP(*CBS*)	WEP(*CBS*)	WEP(*JS*)	ReCNP(*CBS*)	ReCNP(*ECBS*)	ReCNP(*ECBS*)
QGBl	*BlBu*	q=6	q=6	q=6	q=6	q=6	q=6	q=6
	BlCl	BlFi(r=0.50)	BlFi(r=0.60)	BlFi(r=0.15)	BlPu(ρ=0.10)	BlFi(r=0.35)	BlFi(r=0.45)	BlFi(r=0.55)
	CoCl	WEP(*ECBS*)	WEP(*EJS*)	WEP(*CBS*)	WNP(*ARCS*)	ReWNP(*ARCS*)	ReCNP(*ECBS*)	ReCNP(*ECBS*)
EQGBl	*BlBu*	q=6 t=0.95	q=6 t=0.95	q=6 t=0.95	q=6 t=0.95	q=6 t=0.95	q=6 t=0.95	q=6 t=0.95
	BlCl	BlFi(r=0.50)	BlPu(ρ=0.05)	BlFi(r=0.10)	BlPu(ρ=0.10)	BlFi(r=0.35)	BlFi(r=0.50)	BlFi(r=0.50)
	CoCl	WEP(*EJS*)	CEP(*ARCS*)	ReCNP(*CBS*)	ReWNP(*ECBS*)	ReWNP(*ARCS*)	WNP(*ARCS*)	ReCNP(*ARCS*)

(a) Lazy blocking methods

MFIB	*BlBu+BlCl*	q=3 S = {30, 2}	q=2 S = {4, 2}	q=5 S = {2}	q=3 S = {25, 5, 2}	q=4 S = {6, 2}	q=4 S = {10, 8, 6, 4, 2}	-
	CoCl	S = {50, [10 : 2 : 2]} p=10 t=0.0	S = {2} p=4 t=0.5	S = {3, 2} p=4 t=0.625	S = {500(...)2} p=4 t=0.1	S = {8, 6, 5, 4, 2} p=4 t=0.5	S = {20 : 2 : 2} p=1000 t=0.0	- - -
CaCl	*BlBu+BlCl*	q=2 w_1=0.45 w_2=0.60	q=2 w_1=0.40 w_2=0.70	q=2 w_1=0.45 w_2=0.75	q=3 w_1=0.40 w_2=0.60	q=3 w_1=0.15 w_2=0.35	q=6 w_1=0.05 w_2=0.85	- - -
	CoCl	CoPr	CoPr	CoPr	CoPr	CoPr	parameter-free	-
ECaCl	*BlBu+BlCl*	q=4 n_1=1 n_2=13	q=2 n_1=1 n_2=6	q=6 n_1=1 n_2=7	q=4 n_1=10 n_2=42	q=4 n_1=1 n_2=18	q=6 n_1=1 n_2=100	- - -
	CoCl	CoPr	CoPr	CoPr	CoPr	CoPr	parameter-free	-
SuAr	*BlBu+BlCl*	l_m=6 b_M=53	l_m=6 b_M=28	l_m=6 b_M=7	l_m=6 b_M=54	l_m=6 b_M=15	l_m=6 b_M=100	l_m=5 b_M=100
	CoCl	WEP(*EJS*)	CNP(*EJS*)	ReWNP(*ARCS*)	WNP(*ECBS*)	ReCNP(*CBS*)	CNP(*ECBS*)	ReCNP(*ECBS*)
ESuAr	*BlBu+BlCl*	l_m=6 b_M=39	l_m=6 b_M=19	l_m=6 b_M=7	l_m=6 b_M=39	l_m=6 b_M=16	l_m=6 b_M=100	l_m=6 b_M=100
	CoCl	ReCNP(*JS*)	CEP(*ARCS*)	WEP(*CBS*)	CoPr	ReWNP(*EJS*)	CNP(*ECBS*)	ReCNP(*ECBS*)
SoNe	*BlBu+BlCl*	w=4	w=4	w=2	w=4	w=3	w=12	w=57
	CoCl	CoPr	CoPr	CoPr	CoPr	CoPr	CoPr	CoPr
TYPiM	*BlBu+BlCl*	ϵ=0.60 θ=0.20	ϵ=0.05 θ=0.80	ϵ=0.70 θ=0.30	ϵ=0.50 θ=0.55	ϵ=0.40 θ=0.45	ϵ=0.05 θ=0.10	- -
	CoCl	WEP(*CBS*)	WEP(*CBS*)	WEP(*ARCS*)	WEP(*CBS*)	WEP(*ARCS*)	WEP(*ARCS*)	-

(b) Proactive blocking methods

Table 3: Best and default configurations for every blocking method across all real datasets.

them corresponding to the default configuration. As a result, BlFi is clearly the prevalent approach. Yet, its internal parameter, r, varies significantly in the interval [0.10, 0.55] for the best configurations of all lazy methods; the less blocks a method associates with every entity on average and the less duplicates are contained in the input data, the lower gets the optimal value for r. However, r becomes more stable for the default configurations of lazy methods, ranging from 0.45 to 0.55. This means that on average, BlFi should retain every entity in the first half of its blocks for all lazy methods.

For *CoCl*, the best configurations of all lazy methods differ substantially across all datasets. CoPr and ItBl are completely absent, because they execute much more comparisons than MeBl in their effort to retain high recall. CEP is one of the least frequent configurations, appearing just twice, because it performs a deep pruning that shrinks recall. ReCNP prunes the comparisons to a similar extent, but is the most frequent best configuration with 12 appearances. The reason is that it maintains high recall, retaining the most promising comparisons per entity instead of the overall most promising ones. WEP performs a less aggressive pruning that offers an even more stable performance. It corresponds to the default configuration for all lazy methods, while appearing 9 times as best configuration, as well.

For the proactive methods, there is no clear pattern of robustness. Starting with *BlBu+BlCl*, we observe a significant variation in the internal configuration of most methods, as their parameters take almost all possible values across the six datasets. For MFIB, we have $2 \leq q \leq 5$ and $S \subseteq \{2, 4, 5, 6, 8, 25\}$, for CaCL and ECaCL, we have $2 \leq q \leq 6$, $0.05 \leq w_1 \leq 0.45$, $0.35 \leq w_2 \leq 0.75$, $1 \leq n_1 \leq 10$ and $6 \leq n_2 \leq 100$, and for TYPiM, we have $0.05 \leq \epsilon \leq 0.70$ and $0.10 \leq \theta \leq 0.80$. Limited robustness appears in SuAr and ESuAr, which retain the same value for l_m in almost all cases. Their block level parameter is more volatile ($7 \leq b_M \leq 100$), but its optimal value is analogous to the number of existing duplicates: the higher $|D(E)|$ is for the input entity collection E, the larger b_M gets so as to retain more duplicates in the resulting blocks. The same applies to SoNe's parameter, w.

For *CoCl*[2], some of the proactive methods become robust, due to their limited configuration options: CaCl, ECaCl, SoNe are incompatible with MeBl and are exclusively combined with CoPr, which executes less comparisons than ItBl at the cost of slightly lower *PC*. TYPiM also becomes robust, as it consistently achieves the best performance in combination with WEP. On the other hand, SuAr and ESuAr become unstable, as their best configurations are different for every dataset; even their default configurations do not excel in any dataset. Finally, MFIB remains quite sensitive and its internal parameters vary significantly across the datasets.

On the whole, we can conclude that the lazy methods are quite robust with respect to their internal parameters for *BlBu*. With the exception of ACl, their default configurations coincide with their best ones across all datasets. For *BlCl*, their default configuration is BlFi with $r \in [0.45, 0.55]$, which is very close to their best configuration in most cases. For *CoCl*, their default configuration is WEP,

[2] {500(...)2} = {500,100,70,60,50,40,38,36,...8,6,5,4,3,2} for MFIB over D_{cora}.

which matches their best one in just 1/3 of the cases; depending on the dataset, another pruning scheme, typically ReCNP, may yield better performance for this sub-task. The proactive methods can be distinguished into two main categories: those being robust only for *CoCl* and those being partly robust for *BlBu+BlCl*. To the former category belong CaCl, ECaCl, SoNe and TYPiM, while the latter one includes SuAr and ESuAr. In all other cases, the configuration of proactive methods covers a wide range of values and depends heavily on the dataset at hand.

6.2 Effectiveness

Figures 5 and 6 present the performance of lazy and proactive methods with respect to the number of executed comparisons ($||B||$) and *PC* (**E1** in Section 2), respectively.[3] The lazy methods appear in the left column and the proactive ones in the right column. To facilitate their comparison, we use the same scale for their diagrams in all datasets. For each method and dataset, we consider the best and the default configuration for every sub-task.

For the lazy methods, we see that $||B||$ is drastically reduced with every sub-task. The larger the input dataset is, the more unnecessary comparisons it involves and the larger is the impact of *BlCl* and *CoCl*; on average, across all methods and configurations, the two sub-tasks reduce the comparisons of *BlBu* by 2 orders of magnitude over D_{cens}, while for D_{dbp}, the average reduction rises to 3 and 6 orders for the default and best configurations, respectively. As a result, even though the lazy methods typically execute more comparisons than the brute-force approach after *BlBu*, they end up saving at least 2 orders of magnitude after *BlCl* and *CoCl*.

Inevitably, lower $||B||$ values come at the cost of lower *PC* values. The recall of lazy methods is actually inversely proportional to the average number of name-value pairs per entity, $|\bar{p}|$ in Table 2. The larger $|\bar{p}|$ is, the more blocking keys are extracted from each entity and the more blocks are associated with it, reducing the likelihood of missed matches. For this reason, *BlCl* and *CoCl* reduce the recall of *BlBu* by 54%, on average, over the dataset with the lowest $|\bar{p}|$, D_{cens}. In all other datasets, all lazy methods maintain *PC* well over 0.80, with *BlCl* and *CoCl* each reducing it by 2.5%, on average.

Note that the two sub-tasks differ substantially in their performance. In fact, *BlCl* is more effective in detecting unnecessary comparisons: it reduces $||B||$ to a much greater extent than *CoCl*, but its impact on *PC* tends to be smaller. As an example, consider the best configurations over D_{cddb}; both sub-tasks reduce *PC* by 2%, on average, across all methods. However, $||B||$ is reduced from $\sim 10^8$ comparisons to $\sim 10^5$ by *BlCl* and to $\sim 10^4$ by *CoCl*, on average. Similar patterns appear in all datasets, indicating that *BlCl* saves much more comparisons for every pair of duplicates that is missed. Nevertheless, *CoCl* is indispensable for lazy methods, as without it, their computational cost ($||B||$) remains prohibitive.

It is also interesting to examine the relative performance of best and default configurations. In general, this depends on the blocking sub-task to which they pertain. For *BlBu*, the two configurations coincide for all lazy methods except ACl. Even for this method, though, their difference is minor across all datasets with respect to both $||B||$ and *PC*. For *BlCl* and *CoCl*, the best configurations execute less comparisons than the default ones at the cost of slightly lower *PC*. This difference is more intense for *CoCl* and increases with larger datasets; for D_{dbp}, the best *CoCl* configuration saves 2 orders of magnitude more comparisons than the default one for just 5% lower *PC*, on average. The few exceptions to this rule correspond to identical parameters for both configurations (see Table 3) and to cases where the default parameters are more aggressive than the best ones (*e.g.*, lower r for BlFi over D_{cens}). In the latter cases, though, the cost in *PC* does not justify the benefit in $||B||$.

Regarding the proactive methods, we can divide them into two groups based on the impact of *CoCl* on their performance. The first group involves methods that have their comparisons cut down by *CoCl* at no cost in recall; they are exclusively combined with CoPr, which reduces $||B||$ by 41%, 40% and 53% for CaCl, ECaCl and SoNe, respectively. The second group involves methods that have their comparisons cut down to a larger extent at the cost of lower recall. On average, *CoCl* reduces $||B||$ by a whole order of magnitude, while *PC* drops by 4%, 10% and 5% for SuAr, ESuAr and TYPiM, respectively. This applies to all datasets except for D_{cens}, where the very low $|\bar{p}|$ accounts for much lower recall. This pattern is consistent for both configurations. Only for MFIB, the impact of *CoCl* depends on its configuration: for the best one, $||B||$ drops by 2 to 3 orders of magnitude, while reducing *PC* by 10%, on average, across all datasets; for the default configuration, $||B||$ drops by 38% and *PC* by 14%, on average.

With respect to recall, we observe that the proactive methods maintain a sufficiently high *PC*, which exceeds 0.80 in most cases. Yet, it is not very robust, as there are quite a few exceptions. For example, the recall of TYPiM is usually much lower, because it falsely divides pairs of matching entities into different entity types, even though there is a single type in all datasets but D_{dbp}. Most other exceptions correspond to methods with block level parameters, which are configured without considering the input collection size $|E|$ (see Table 1). Their restrictive effect, which aggravates with the default configuration, shows itself in large datasets like D_{dbp} or in datasets with a large number of duplicates like D_{cora}.

Comparing lazy with proactive methods, we observe that the former typically entail a much higher and robust recall, at the cost of a significantly higher number of comparisons. To quantify their relative effectiveness, we rely on precision (**E2** in Section 2): we define the best method per dataset and configuration as the one maximizing *PQ* for *PC*>0.80 after *CoCl*. In this context, the four smaller, structured datasets are dominated by proactive methods: MFIB offers the most effective best configuration, with $0.55 \leq PQ \leq 0.91$, and CaCl the most effective default configuration, with $0.08 \leq PQ \leq 0.18$. The two larger, heterogeneous datasets are dominated by ACl for both configurations, with $0.0003 \leq PQ \leq 0.11$. There are just two exceptions: the default configuration of EQGBl is the most effective for D_{cora} (PQ=0.48), and the best configuration of SuAr is the most effective for D_{dbp} (PQ=0.12).

6.3 Time Efficiency

Figures 7 and 8 present the performance of lazy and proactive methods with respect to overhead and resolution time, respectively (**T1** and **T2** in Section 2). The lazy methods appear on the left and the proactive ones on the right. All diagrams use the same scale for both types of methods and consider the best and the default configuration for every blocking sub-task across all datasets. In all cases, lower bars indicate higher time efficiency.

Starting with the lazy methods, we observe minor differences in the *OTime* of *BlBu* not only among the various methods, but between their configurations as well. StBl is consistently the fastest method, as it involves the simplest functionality that lies at the core of the other methods. ACl is the most time-consuming method when using character or token n-gram graphs as a representation model (see Table 3); in all other cases, EGQBl is the slowest technique, due to the high number of blocking keys it produces. These

[3]Note that we consider $||B||$ instead of *RR* (**E3** in Section 2), because *RR* takes high values of low discriminativeness; for example, the difference between 0.90 and 0.95 might be a whole order of magnitude in terms of comparisons.

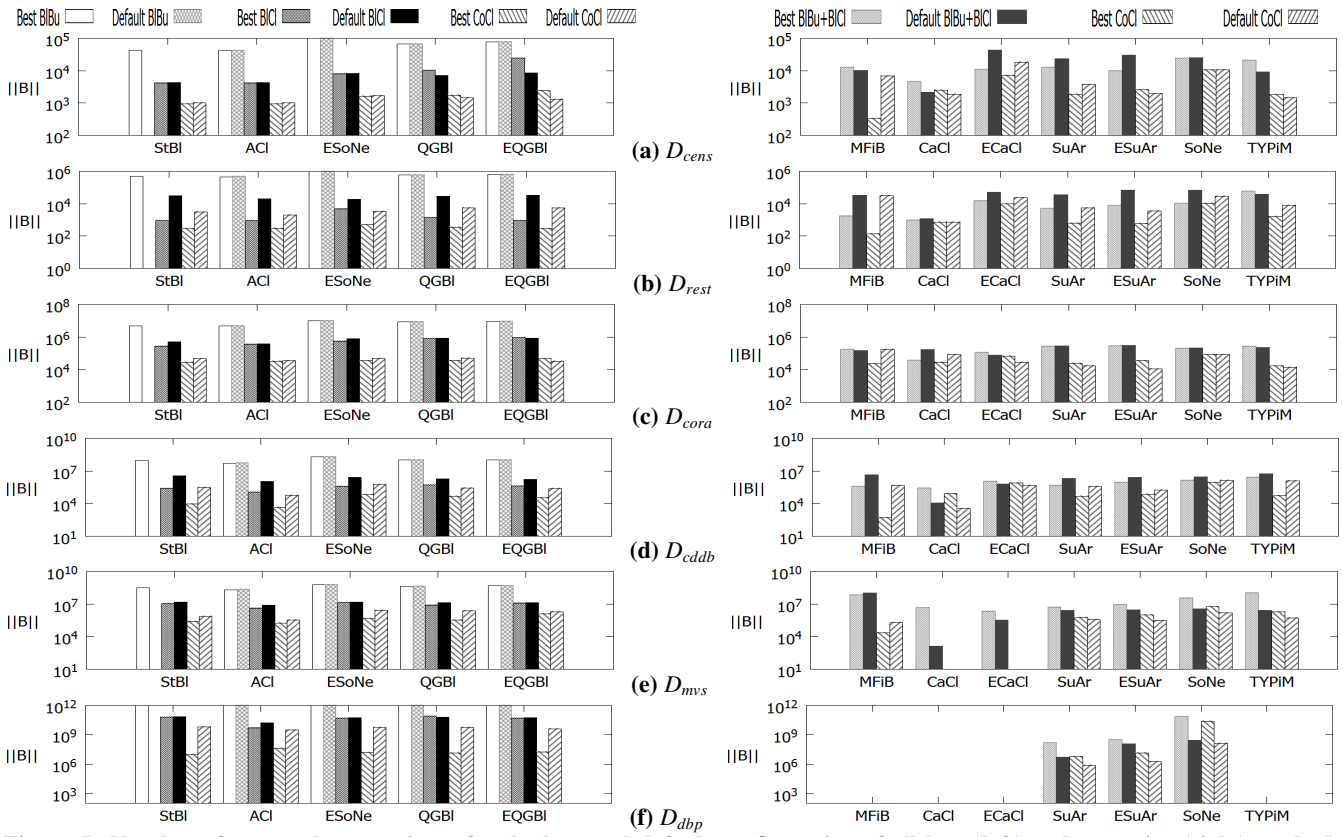

Figure 5: Number of executed comparisons for the best and default configuration of all lazy (left) and proactive (right) methods across all real datasets. The vertical axis is logarithmic. Lower bars indicate better effectiveness.

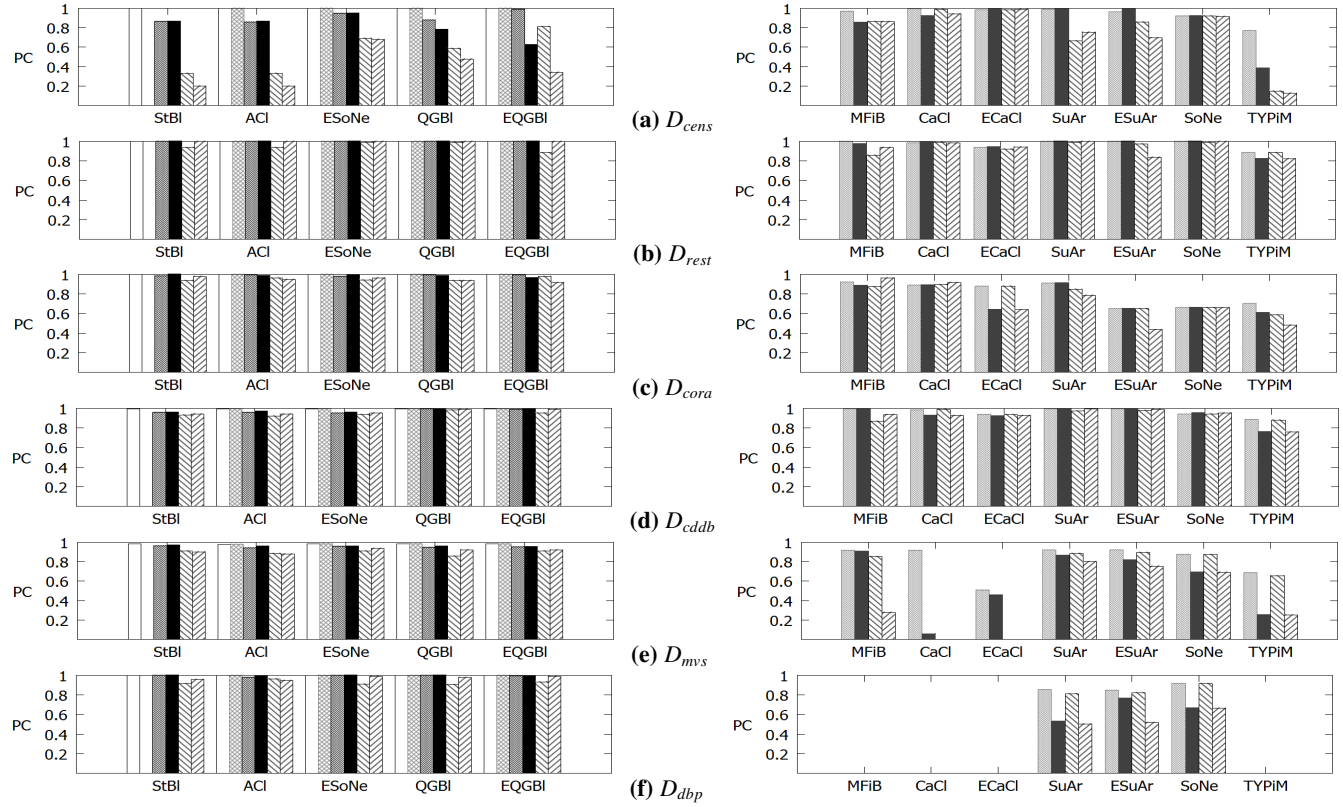

Figure 6: Recall for the best and default configuration of all lazy (left) and proactive (right) methods across all real datasets. Higher bars indicate better effectiveness.

patterns apply to *BlCl*, as well, since the corresponding methods involve a crude, rapid operation that increases *OTime* just slightly. *CoCl* increases *OTime* to a greater extent that depends linearly on the number of comparisons it receives as input, due to MeBl: the larger $||B||$ is after *BlCl*, the more *OTime* rises during *CoCl*.

The *RTime* of lazy methods is dominated by the number of executed comparisons, $||B||$: the larger it is for a method with respect to a specific sub-task, the larger the corresponding *RTime* gets. This means that *OTime* merely accounts for a small portion of *RTime*, a situation that is reflected in the difference of their scales for the diagrams of D_{cora}, D_{cddb} and D_{dbp}. It also means that *RTime* is consistently reduced by *BlCl* and *CoCl* to a significant extent. In fact, the resolution time of *BlBu* is typically worse than that of the brute-force approach, $RTime(E)$ in Table 2, for all lazy methods. However, *BlCl* and *CoCl* reduce *RTime* by at least 1 order of magnitude across all datasets, outperforming $RTime(E)$ in all cases. For instance, their best configuration requires less than 5 hours to resolve D_{dbp} for most lazy methods.

Among the lazy methods, ACl exhibits the lowest *RTime* for *BlBu*, despite its higher overhead, because it consistently executes much fewer comparisons. The same applies to *BlCl* for the three largest datasets, where ACl maintains a clear advantage in terms of $||B||$. After *CoCl*, though, StBl takes the lead as the fastest lazy method in most cases. Even though it executes much more comparisons than ACl, it achieves a lower *RTime* thanks to its significantly lower *OTime*. In case we used a more complex and time-consuming method for entity matching, ACl would be the fastest lazy method for *CoCl*, as well.

As for the proactive methods, we can divide them into 2 groups: the methods that scale to all datasets (SuAr, ESuAr and SoNe) and those that failed to complete on D_{dblp}. The former group exhibits a behavior similar to the lazy methods. They involve low overhead times that increase with *CoCl*, especially when involving MeBl. Their *OTime* accounts for a small part of *RTime*, which is dominated by the number of executed comparisons. Unlike the lazy methods, though, these techniques outperform $RTime(E)$ to a great extent already by *BlBu+BlCl*, with SuAr and ESuAr achieving the fastest best and default configurations, respectively, across most datasets. For example, SuAr and ESuAr are able to resolve D_{dbp} in less than 1 hour, under all settings.

The second group of proactive methods includes MFIB, CaCl, ECaCl and TYPiM. In most cases, their overhead time is at least an order of magnitude higher than all other methods. It accounts for a large portion of *RTime*, particularly for their best configurations, which execute a very low number of comparisons. However, their resolution time exceeds $RTime(E)$ in many cases, especially for the three smaller datasets. It is no surprise, therefore, that they do not scale to the largest dataset, D_{dbp}. The reason is that they build blocks based on the similarity of blocking keys, unlike most other methods, which rely on the equality of keys.

6.4 Scalability

To assess the scalability of lazy and proactive methods, we applied their default configuration for all sub-tasks to the synthetic datasets in Table 2(b). Figure 9 presents the outcomes with respect to *PC* (**E1** in Section 2), the number of executed comparisons $||B||$, *OTime* (**T1** in Section 2) and *RTime* (**T2** in Section 2). All diagrams use the same scale for both method types. Recall that the higher *PC* is, the better the performance and the other way around for the rest of the measures.

Starting with the lazy methods, we see in Figure 9(a) that their recall remains well over 0.80 across all datasets. The *PC* of ESoNe and EQGBl remains practically stable, fluctuating around 0.87. In contrast, the *PC* of StBl, QGBl and ACl decreases slightly with the increase of the input collection size $|E|$: it drops by 5%, 6% and 7%, respectively, when moving from D_{10K} to D_{2M}.

With respect to $||B||$, all lazy methods scale super-linearly (>200× increase), yet sub-quadratically (<40,000× increase), when moving from D_{10K} to D_{2M}. Figure 9(b) demonstrates that the lowest $||B||$ corresponds to ACl and StBl (6,000× and 8,000× increase, respectively) and the highest to ESoNe (31,000× increase) across all datasets. In absolute terms, though, ESoNe consistently executes 3 orders of magnitude less comparisons than the brute-force approach – $||E||$ in Table 2(b). Note also that there is a clear trade-off between recall and the number of comparisons executed by the lazy methods with decreasing recall: QGBl>StBl>ACl for $||B||$ and an opposite trend for *PC*.

Regarding *OTime*, Figure 9(c) indicates that all lazy methods scale better than with $||B||$: from D_{10K} to D_{2M}, their overhead time rises from 1,300 times (EQGBl) to 4,000 times (StBl). In absolute terms, ACl is consistently the most time-consuming method, requiring at least double the time of all other lazy methods, which exhibit comparable overhead times. Nevertheless, ACl can process D_{2M} within just 2.5 hours (8×10^6 milliseconds).

Similar patterns correspond to *RTime*, which rises by less than 5,000 times for most lazy methods, when moving from D_{10K} to D_{2M}. The only exception is ESoNe, whose resolution time rises by almost 10,000 times and is the worst across all datasets, because it executes the highest number of comparisons. In absolute terms, though, all lazy methods outperform the brute-force approach by several orders of magnitude: $RTime(E)$ for D_{2M} drops from 1 year – see Table 2(b) – to just 4 hours for ESoNe and to a mere 1.5 hours for StBl, which consistently constitutes the fastest lazy method.

Regarding the proactive methods, we can group them as in Section 6.3. On the one hand, MFIB, CaCl, ECaCl and TYPiM only scale to D_{300K}. They achieve excellent effectiveness, scaling linearly with respect to $||B||$, while their recall remains close to 0.80 in the worst case; the only exception is TYPiM, which scales quadratically in terms of $||B||$ and detects around half the existing duplicates. Their *RTime* is entirely dominated by *OTime*, due to their time-consuming, similarity-based functionality. Their overhead scales quadratically from D_{10K} to D_{300K} and, thus, they require at least 2 hours for processing D_{300K}. Inevitably, they fail to process the two largest datasets within 1 day.

On the other hand, SuAr, ESuAr and SoNe scale to process all datasets. Their $||B||$ scales linearly, while their *PC* drops steadily when moving from D_{10K} to D_{2M}; it starts from higher than 0.97 and ends lower than 0.80, reduced by 26%, on average. Their time efficiency is excellent, with *OTime* and *RTime* scaling linearly to the larger datasets. This should be attributed to the equality-based functionality of SuAr and ESuAr and to the $O(n \log n)$ complexity of SoNe. Compared to lazy methods, they execute less comparisons by one order of magnitude, on average, and are 3 times faster than the fastest lazy method, StBl.

On the whole, we conclude that the lazy methods scale slightly super-linearly to large datasets, emphasizing recall at the cost of a high number of comparisons. In contrast, most proactive methods cannot process large entity collections within a reasonable time, despite their excellent effectiveness. Only SoNe, SuAr and ESuAr scale linearly to large datasets, outperforming the lazy methods, at the cost of lower recall.

7. CONCLUDING REMARKS

We conclude with the following four observations. *First,* all blocking methods depend heavily on the fine-tuning of at least one of the three blocking sub-tasks. To this attests the large performance

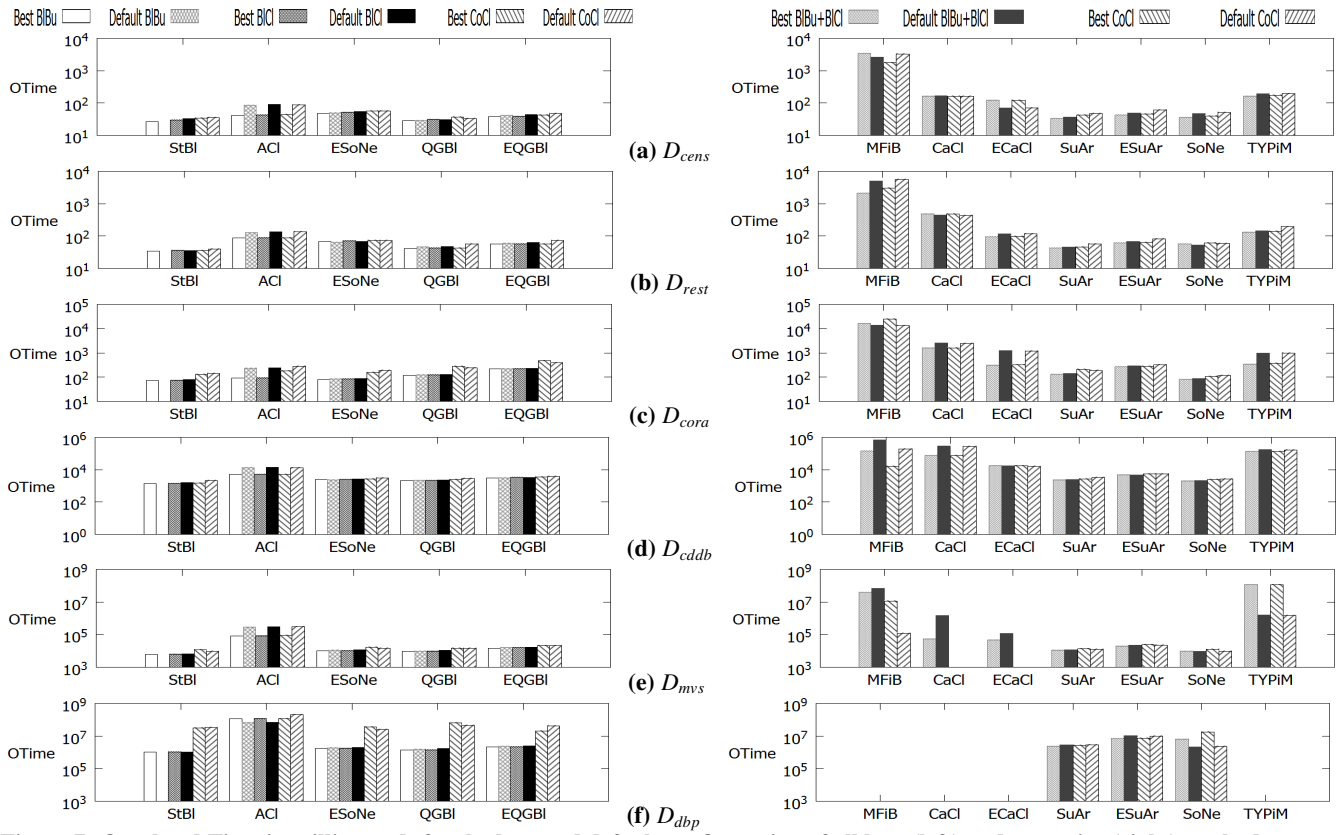

Figure 7: Overhead Time in milliseconds for the best and default configuration of all lazy (left) and proactive (right) methods across all real datasets. The vertical axis is logarithmic. Lower bars indicate better time efficiency.

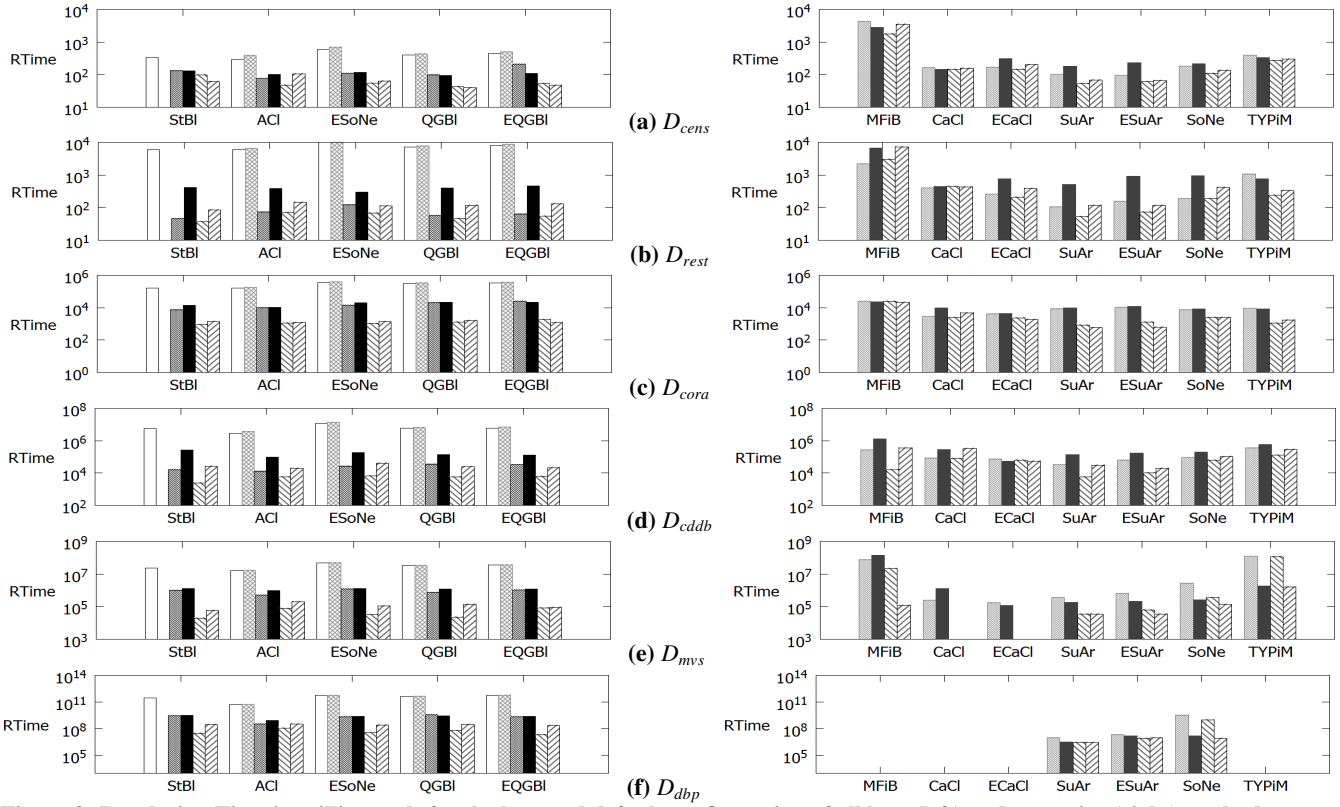

Figure 8: Resolution Time in milliseconds for the best and default configuration of all lazy (left) and proactive (right) methods across all real datasets. The vertical axis is logarithmic. Lower bars indicate better time efficiency.

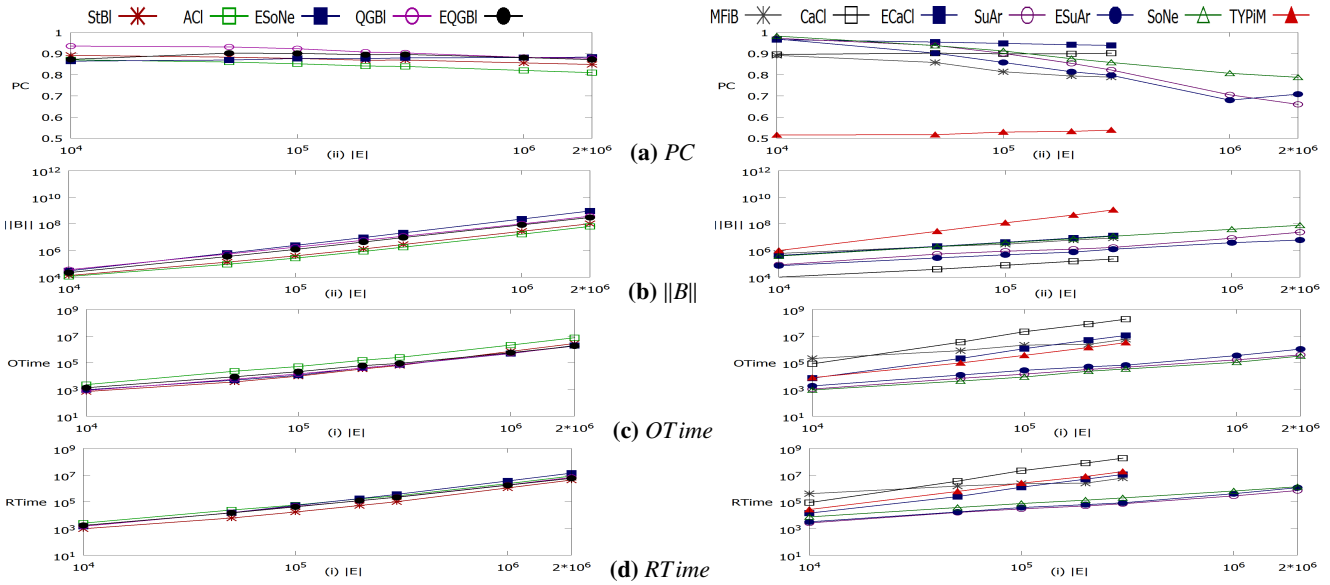

Figure 9: Scalability of the default configuration of lazy and proactive methods across the synthetic datasets with respect to (a) recall, (b) executed comparisons, (c) overhead time, and (d) resolution time. All axes are logarithmic except for the vertical ones in (a).

difference between their best and default configurations. Most lazy methods are stable for *BlBu*, sufficiently robust for *BlCl* and rather unstable for *CoCl*. The proactive methods are robust only for some of the *BlBu+BlCl* parameters, or for *CoCl*.

Second, all blocking sub-tasks are indispensable for achieving a good balance between precision and recall for both proactive and lazy methods. The former typically excel in homogeneous datasets and the latter in heterogeneous ones. MFIB and ACl show the best potential, respectively, *i.e.*, the most effective best configurations.

Third, most proactive methods exhibit very high overhead time, due to their similarity-based functionality. As a result, they do not scale to large datasets. Even for the small ones, their computational cost pays off only when the entity matching method is complex and time-consuming. In contrast, the overhead time of all lazy methods scales well to large datasets and accounts for a small portion of their resolution time, which outperforms the brute-force approach at least by an order of magnitude.

Fourth, the default configuration of lazy methods is suitable for applications emphasizing recall, as it exhibits excellent time efficiency and scalability for $PC > 0.80$ across all datasets, but the smallest one. StBl should be preferred when using a cheap entity matching method, and ACl otherwise. For applications emphasizing efficiency, the default configuration of SuAr should be preferred, as it consistently exhibits the lowest resolution time in the scalability analysis.

In the future, we plan to investigate the automatic fine-tuning of blocking methods, trying to narrow the gap in the performance of best and default configurations.

Acknowledgements. This work was partially supported by EU H2020 BigDataEurope (#644564) and The MAGNET Program InfoMedia (Office of the Chief Scientist of the Israeli Ministry of Industry, Trade & Labor).

References

[1] A. N. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, 2005.
[2] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, pages 87–96, 2006.
[3] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.
[4] P. Christen. Febrl an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *KDD*, pages 1065–1068, 2008.
[5] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.
[6] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
[7] I. P. Fellegi and A. B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.
[8] J. Fisher, P. Christen, Q. Wang, and E. Rahm. A clustering-based framework to control block sizes for entity resolution. In *KDD*, pages 279–288, 2015.
[9] A. Gal. Uncertain entity resolution. *PVLDB*, 7(13):1711–1712, 2014.
[10] A. Gal and B. Kimelfeld. Entity resolution in the big data era: Probabilistic db support to entity resolution. In *EDBT (tutorial)*, 2015.
[11] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative Data Cleaning: Language, Model and Algorithms. In *VLDB*, pages 371–380, 2001.
[12] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.
[13] G. Giannakopoulos, V. Karkaletsis, G. A. Vouros, and P. Stamatopoulos. Summarization system evaluation revisited: N-gram graphs. *TSLP*, 5(3):1–39, 2008.
[14] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. Knowl. Data Eng.*, 17(10):1347–1362, 2005.
[15] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
[16] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *SIGMOD Rec.*, 24(2):127–138, 1995.
[17] R. Isele, A. Jentzsch, and C. Bizer. Efficient multidimensional blocking for link discovery without losing recall. In *WebDB*, 2011.
[18] B. Kenig and A. Gal. Mfiblocks: An effective blocking algorithm for entity resolution. *Inf. Syst.*, 38(6):908–926, 2013.
[19] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.
[20] Y. Ma and T. Tran. Typimatch: type-specific unsupervised learning of keys and key values for heterogeneous data integration. In *WSDM*, pages 325–334, 2013.
[21] A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.
[22] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, pages 440–445, 2006.
[23] A. N. Ngomo and S. Auer. LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, pages 2312–2317, 2011.
[24] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. *PVLDB*, pages 312–323, 2015.
[25] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Eliminating redundancy in blocking-based entity resolution. In *JCDL*, pages 85–94, 2011.
[26] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Trans. Knowl. Data Eng.*, 25(12):2665–2682, 2013.
[27] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolutionto the next level. *IEEE Trans. Knowl. Data Eng.*, 26(8), 2014.
[28] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. In *EDBT*, pages 221–232, 2016.
[29] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.