Noname manuscript No.
(will be inserted by the editor)

Open Benchmark for Filtering Techniques in Entity Resolution

Franziska Schoger · Marco Fisichella · George Papadakis · Konstantinos Nikoletos · Nikolaus Augsten · Wolfgang Nejdl · Manolis Koubarakis

Received: date / Accepted: date

Abstract Entity Resolution identifies entity profiles that represent the same real-world object. Its brute-force approach suffers from a quadratic time complexity, since it considers all possible pairs of entities. To ameliorate this issue, filtering techniques reduce its computational cost to highly similar and, thus, highly likely matches. Such techniques come in two forms: (i) *blocking workflows* group together entity profiles with identical or similar signatures, and (ii) *nearest-neighbor workflows* convert all entity profiles into vectors and detect the ones closest to every query entity. As yet, the main techniques of these two types have not been juxtaposed in a systematic way and, thus, their relative performance is unknown.

To cover this gap, we perform an extensive experimental study that investigates the relative performance of the main representatives per type over numerous established datasets. Comparing techniques from the two types in a fair way is a non-trivial task, because the configuration parameters of each approach have a significant impact on its performance, but are hard to fine-tune. We consider a plethora of parameter configurations per methods, optimizing each method with respect to recall and precision in both schema-agnostic and schema-based settings. The experimental results provide novel insights into the effectiveness, the time efficiency and the scalability of the considered techniques.

This work actually extends our original experimental analysis with new methods for both types of filtering workflows and two new optimization targets, which increase the generality of our conclusions.

F. Schoger
L3S Research Center, Germany
E-mail: schoger@l3s.de

M. Fisichella
L3S Research Center, Germany
E-mail: mfisichella@l3s.de

G. Papadakis
National and Kapodistrian University of Athens, Greece
E-mail: gpapadis@di.uoa.gr

K. Nikoletos
National and Kapodistrian University of Athens, Greece
E-mail: k.nikoletos@di.uoa.gr

Nikolaus Augsten
University of Salzburg, Austria
E-mail: nikolaus.augsten@plus.ac.at

Wolfgang Nejdl
L3S Research Center, Germany
E-mail: nejdl@l3s.de

M. Koubarakis
National and Kapodistrian University of Athens, Greece
E-mail: koubarak@di.uoa.gr

1 Introduction

Entity Resolution (ER) detects so-called *duplicates* or *matches*, i.e., different entity profiles that describe the same real-world object [1]. ER lies at the core of numerous data integration tasks, such as Link Discovery, query answering and object-oriented searching [2–4].

The time complexity of ER is inherently quadratic, since it considers all possible pairs of entity profiles in the worst case. To address this, ER is applied through the *filtering-verification framework* in practice [1, 3]. Initially, *filtering* performs a coarse-grained, rapid clustering that restricts the computational cost to the most promising matches, which are called *candidate pairs*. Next, the *verification* step performs a fine-grained processing that decides for every candidate pair whether it is matching or not.

Verification, a.k.a. *matching*, typically applies a similarity function to compare the textual values of the candidate pairs [2,3,5,6]. The earlier methods operated in a rule-based manner that compared the resulting similarity scores with thresholds so as to decide whether a pair of entities is matching or not. More advanced and recent techniques rely on learning, modelling verification as a binary classification task that labels each pair as `match` or `non-match` [6]. All types of learning techniques have been applied in this frame, from supervised to active, transfer and deep learning [7]. Most of them operate *locally*, taking separate decisions for each pair of candidates, while others consider *global* evidence among all candidate pairs, feeding their decisions and matching probabilities to clustering methods that refine the output [8].

In this work, we are interested in benchmarking filtering methods, assessing their relative performance in reducing the search space of ER. Three evaluation measures are considered in this context: recall, precision and time efficiency. More specifically, the desiderata for a filtering technique are the following:

1. *High recall.* The candidate pairs should exclude a limited portion of the existing duplicates, keeping the number of false negatives as low as possible.
2. *High precision.* The candidate pairs should include a limited portion of non-matching pairs, keeping the number of false positives as low as possible.
3. *Low run-time.* Filtering's overhead should account for a limited part of the overall ER run-time.

Most filtering techniques in the literature are crafted for textual attribute values and are based on heuristics, operating in a learning-free manner that requires no labelled instances for training [3,9,10]. The reason is that labelled data are rarely available and too expensive for such a coarse-grained process [11]. For this reason, this work focuses exclusively on unsupervised filtering techniques, which are distinguished into two broad types:

1. *Blocking workflows* transform every input entity profile into a set of representative signatures that facilitates the efficient creation of clusters [3, 10, 12].
2. *Nearest neighbor (NN) workflows* transform every input entity into a sparse [13, 14] or dense numeric vector [15]. These vectors are indexed through an efficient data structure, which is queried by part or all of the entity profiles so as to yield as candidates the most similar vectors/entities.

Despite their fundamentally different operation, both types receive the same input (the entity profiles) and produce the same output (candidate pairs). Nevertheless, no prior work has investigated their relative performance. The past experimental analyses examine each type independently of the other: blocking workflows are tested in [10, 12, 16], whereas the sparse [13, 14, 17] and the dense vector-based NN methods [15] have only been evaluated with respect to run-time and approximation quality, which is not related to their ER performance (cf. Section 2). Most importantly, the few comparisons between blocking and NN workflows are not carried out in a systematic way; for example, Standard Blocking is compared to DeepBlocker in [11] as an independent approach, rather than as part of a blocking workflow, as is common in the literature [12].

Benchmarking techniques from these two types is a non-trivial task, because of the configuration parameters: they have a significant impact on the performance of each approach, yet determine them in different ways. As a result, *no* systematic fine-tuning approach applies seamlessly to both types; for example, the step-by-step configuration optimization in [12] applies to the blocking workflows, but not to the NN ones, because the first steps in the latter do not yield candidates and, thus, their performance cannot be directly measured.

To cover this gap in the literature, we present the first thorough and systematic experimental study that covers both types of filtering methods. At the core of our analysis lies a novel methodology, which fine-tunes each filtering algorithm with respect to a common performance target so as to ensure a fair comparison. More specifically, our original work maximized precision for a high recall of 0.9, examining thousands of different parameter configurations per algorithm [18]. This methodology was originally applied to seven blocking and 10 NN workflows over 10 real-world datasets under both *schema-based* and *schema-agnostic settings*; the former exclusively consider the value of the most reliable attribute in each input entity, while the latter leverage all attribute values per entity. In a nutshell, the schema-agnostic settings trade higher run-times for more robust performance and higher recall than the schema-based ones, while having a broader scope, as they support heterogeneous schemata, too. For these reasons, the schema-agnostic settings were exclusively used in the scalability analysis, which applied the same performance target to the same methods over seven synthetic datasets of increasing size (from 10^4 to $2{\cdot}10^6$ entities).

In this work, we extend our original analysis in the following ways:

– We test four additional established workflows: three blocking ones along with an NN one. We also introduce a novel NN workflow called Random Join, which trades higher efficiency for slightly lower effectiveness, while acting as a baseline for evaluating the usefulness of the more elaborate techniques. Sur-

prisingly, it outperforms many other methods, even those based on Deep Learning.
- To reinforce the generality of our methodology (and the corresponding conclusions), we consider two more performance targets for the 10 real-world and the seven synthetic datasets (of the scalability analysis): maximum precision for recall equal to 0.85 and 0.95.
- We ran all experiments on a powerful server, instead of the commodity hardware that was used in our original work. This way, we applied all methods to all datasets, covering some gaps of the NN workflows over the largest datasets, due to insufficient memory.
- Explain new content.

Overall, we make the following contributions:

- We propose a configuration optimization methodology that allows for assessing the relative performance of two fundamentally different workflows for filtering in ER: the blocking and the NN ones.
- We perform the first thorough experimental analysis on these two workflow types, testing 22 state-of-the-art workflows on 10 real-world datasets in both schema-based and schema-agnostic settings. Most of the methods are configured on every dataset, testing thousands of different configurations, with respect to three performance targets of increasing recall. The conclusions drawn from the three targets accommodate most practical ER applications.
- We present a qualitative analysis of the filtering workflows based on their *scope* and *internal functionality*.
- We perform a thorough scalability analysis that applies all 22 workflows to seven synthetic datasets of increasing size and optimizes their performance with respect to the three performance targets.
- Our work provides new insights into the relative performance and scalability of the considered techniques. We actually show that the blocking workflows and the cardinality-based sparse NN methods consistently excel in performance.
- Two of the tested NN methods, SCANN and kNN-Join, are applied to ER for the first time. SCANN is one of the most scalable techniques, and kNN-Join one of the best performing ones, while sticking out by its intuitive fine-tuning.
- All code and data used in this work are publicly available through a new, open initiative that is called Continuous Benchmark of Filtering methods for ER[1].

The remaining part of the paper is structured as follows: Section 2 discusses the related works, while Section 3 provides background knowledge on filtering and defines formally the configuration optimization task. We elaborate on the filtering methods in Section 4 and present their qualitative and quantitative analyses in Sections 5 and 6, respectively. Section 7 concludes with the main findings of our experimental analysis along with directions for future research.

[1] For more details, please refer to `https://github.com/gpapadis/ContinuousFilteringBenchmark`

2 Related Work

Numerous studies have examined the relative performance of blocking methods. The earliest ones were published in [3,10]. They focus on *schema-based settings* in conjunction with a number of user-defined parameter configurations. Also, they exclusively consider the first step of blocking workflows: block building.

The same block building methods and configurations were examined in [16], which applies them to *schema-agnostic settings* as well. In contrast to schema-based settings, the experimental results suggest that recall increases significantly, without requiring any background knowledge about the given data or its quality. They also demonstrate that the sensitivity of the model to parameter configuration has been significantly decreased.

Extending these works, the study in [12] examines the relative performance of blocking *workflows* in schema-agnostic settings. These workflows are formed by three consecutive steps: block building, block filtering and comparison cleaning. The present work goes beyond this analysis by considering workflows formed by four consecutive steps: block building, block purging, block filtering, and comparison cleaning. Each of the last three steps is optional, thus yielding seven different pipelines. Of them, only one was examined in [12].

There are two more critical differences between our work and the analysis in [12]. First, our experiments share just one dataset – excluding the scalability ones. Second, and most important, the workflow configuration is optimized in [12] through a *step-by-step* approach that relies on heuristics: initially, the block building performance is heuristically optimized, its output is fed as input to block filtering, which is also heuristically fine-tuned, and its results form the input forwarded to comparison cleaning for its fine-tuning. In contrast, we consider a *holistic* approach to configuration optimization: all steps in a blocking workflow are simultaneously fine-tuned. This approach consistently outperforms the step-by-step one as explained in [19, 20], because it is not limited to local maxima per workflow step, while testing a substantially larger set of configurations.

Finally, we go beyond all past relevant analyses [3, 10, 12, 16] in two more ways: (i) we systematically fine-tune blocking workflows in the context schema-based

settings, and (ii) we compare blocking workflows with sparse and dense NN ones. These topics were not examined in the literature before. For example, the schema-based settings examined in [3,10] leveraged human knowledge, rather than a systematic, automatic fine-tuning.

It is important to note that the sparse NN workflows essentially correspond to similarity joins. The relative performance of the main join algorithms is examined in [13,14,17] with respect to run-time. Recall and precision are not considered, because they are identical across all approaches, i.e., all methods retrieve the same pairs of entities, which are those exceeding the given similarity threshold. ER's recall and precision have not been evaluated for these pairs, given that they are not necessarily duplicates. In other words, these studies cannot be used to evaluate the performance of string similarity joins in ER. It is also important to note that none of the sparse NN methods we consider have been previously tested in experiments; the local kNN-Join lies out of the focus of [13,14,17], while the range join is combined with much lower similarity thresholds when applied to ER that those examined in [13, 14, 17].

The dense NN workflows are experimentally compared in [15]. Yet, the evaluation criteria are restricted to throughput, i.e., the number of queries executed per second, and to recall, i.e., the proportion of retrieved vectors that are indeed the closest ones according to a specific distance function (such as Euclidean). As opposed to ER recall, though, the closest vectors do not necessarily match. Nevertheless, we rely on the experimental results of [15] in order to select the top-performing dense NN methods: Cross-polytope and Hyperplane LSH, FAISS and SCANN. We also consider MinHash LSH, a popular filtering technique [9], and DeepBlocker, the most recent learning-based approach, which consistently outperforms all others [11].

3 Preliminaries

An entity profile e_i describes a real world object as a set of textual name-value pairs, i.e., $e_i = \{\langle n_j, v_j \rangle\}$ [12, 16]. Most established data formats are covered by this model, from the structured records in relational databases to the semi-structured instance descriptions in RDF data. We refer to two entities, e_i and e_j, as *duplicates* or *matches*, $e_i \equiv e_j$, if they pertain to the same real-world object.

Following [3, 5], we differentiate between two main types of ER:

1. In *Dirty ER* or *Deduplication*, the input comprises a set of entity profiles, $\mathcal{E}$, while the output consists of the detected duplicates within $\mathcal{E}$.
2. In *Clean-Clean ER* or *Record Linkage*, the input comprises two sets of entity profiles, $\mathcal{E}_1$ and $\mathcal{E}_2$, both of which lack any matching profiles. The output consists of the detected duplicates among $\mathcal{E}_1$ and $\mathcal{E}_2$.

Both tasks are addressed through a two-step filtering-verification framework. The former receives the input of ER and produces a set of candidate pairs $\mathcal{C}$, which are those more likely to be duplicates. In this way, the computational cost is reduced to $|\mathcal{C}|$ from the quadratic cost of the brute-force approach, $|\mathcal{E}| \cdot (|\mathcal{E}| - 1)/2$ or $|\mathcal{E}_1| \cdot |\mathcal{E}_2|$, respectively. The candidate pairs are then examined through an elaborate time-consuming approach during verification.

In this context, the following measures are typically used to assess filtering's **effectiveness** [5, 10, 12, 16, 21]:

1. *Pair Completeness (PC)* corresponds to *recall*, measuring the ratio of the duplicate pairs in $\mathcal{C}$ with respect to those in the groundtruth, $\mathcal{D}$: $PC(\mathcal{C}) = |\mathcal{D}(\mathcal{C})|/|\mathcal{D}|$, where $|\mathcal{D}(\mathcal{C})|$ denotes the number of duplicates in the candidate pairs.
2. *Pairs Quality (PQ)* corresponds to *precision*, estimating the ratio of pairs in $\mathcal{C}$ that correspond to real duplicates with respect to all pairs in $\mathcal{C}$: $PQ(\mathcal{C}) = |\mathcal{D}(\mathcal{C})|/|\mathcal{C}|$.

Both measures are restricted in the range of $[0, 1]$, with higher values denoting higher effectiveness. It is important to note that there is an inherent trade-off between PC and PQ: the larger C is, the higher is PC, at the expense of a lower PQ and vice versa, if the set of candidates is smaller. Finding a good balance between these two measures is the objective of filtering.

To compare fundamentally different filtering techniques on an equal footing, we formalize the following configuration optimization task:

Problem 1 (Configuration Optimization) Given two sets of entity profiles, $\mathcal{E}_1$ and $\mathcal{E}_2$, a filtering method, and a threshold τ on recall (PC), configuration optimization fine-tunes the parameters of the filtering method such that the resulting set of candidates $\mathcal{C} \sqsubseteq \mathcal{E}_1 \times \mathcal{E}_2$ maximizes PQ for $PC \geq \tau$.

This definition can be easily extended to Dirty ER. Note that we set a threshold on recall, because filtering determines the overall recall of ER, i.e., verification cannot detect duplicate pairs, which are not included in the candidates C that form its input. This is true both for Clean-Clean and Dirty ER. For this reason, we chose three PC thresholds, $\tau \in \{\mathbf{0.85}, \mathbf{0.90}, \mathbf{0.95}\}$, which allow for high recall of any end-to-end ER pipeline.

Another crucial aspect of filtering is its **time efficiency**, which should be high so as to restrict its overhead on ER. This is measured through the *run-time*

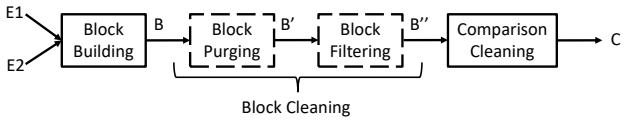

Fig. 1 The blocking workflow [22]. Dotted contours indicate optional steps.

(RT), i.e., the time between receiving the input set(s) of entity profiles and producing the output set of candidate pairs. The lower RT is, the better.

4 Filtering Methods

4.1 The two paradigms of filtering

Filtering comes in two forms: blocking and NN workflows, as shown in Figures 1 and 2, respectively.

In blocking workflows, the first step performs block building, which associates each input entity with one or more signatures and clusters together entities with similar or identical signatures. Matching candidates are entities that appear together in at least one block. Within these blocks, there are two types of candidates that are considered unnecessary, i.e., pairs that reduce precision (PQ) without increasing recall (PC).

1. *Redundant* candidates are those co-occurring in multiple blocks, because the blocks are overlapping.
2. *Superfluous* candidates correspond to pairs of non-matching entities.

To eliminate the redundant and reduce the superfluous candidates after block building, the initial blocks are restructured based on global patterns with block and comparison cleaning [12, 23]. In more detail, two coarse-grained block cleaning methods, Block Purging and Block Filtering, can be applied to the initial blocks. In both cases, a new, smaller set of blocks B' or B'', is generated. Next, one of the available, fine-grained comparison cleaning techniques can be applied to B' or B'', yielding the final set of candidates C. However, these steps are optional and can be omitted – for instance, in schema-based settings, where only few redundant and/or superfluous candidates are expected.

The NN workflows follow a fundamentally different approach. As shown in Figure 2, they organize the input set $\mathcal{E}_1$ into an index I (e.g., an inverted index) and use the other dataset as a query set. Thus, the set of candidate pairs C is derived by probing the index I for every entity profile $e_2 \in \mathcal{E}_2$ and aggregating the results. Unlike the blocking workflows, the NN ones inherently avoid redundant candidate pairs, because every entity from $\mathcal{E}_2$ is associated with the most similar indexed entities from $\mathcal{E}_1$.

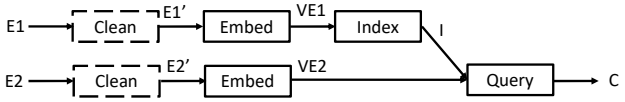

Fig. 2 The workflow of NN methods. Dotted contours indicate optional steps.

Prior to that, the stop-words may be removed from the textual attribute values and stemming may be applied to each word to transform it into its base/root form (e.g. "blocks" becomes "block") [24]. As a result of this cleaning, we obtain two collections, $\mathcal{E}_1'$ and $\mathcal{E}_2'$, which remain in textual form. Subsequently, the input entities are embedded in numeric vectors, which can be either sparse, based on bag of words models, or dense, stemming from pre-trained language models.

4.2 Blocking workflows

Block building. We consider the following algorithms, which achieve the best performance in the past experimental studies [10, 12, 16],

1) *Standard Blocking.* It extracts signatures from every entity by tokenizing the selected attribute values on whitespace. The resulting set of tokens are the signatures. A different block is created for every distinct token, involving all entities that contain it in the selected attribute value(s). Note that no configuration parameter is involved in this process.

2) *Q-Grams Blocking.* To capture matching entities with character-level errors, this algorithm uses as signatures the set of q-grams that are extracted from the tokens of Standard Blocking. Every distinct q-gram yields a block, encompassing all entities with that q-gram in any of the considered values.

3) *Extended Q-Grams Blocking.* Its goal is to improve Q-Grams Blocking by yielding blocks that are smaller, but contain candidates with more common content. To this end, it concatenates at least L q-grams, where $L = max(1, \lfloor k \cdot t \rfloor)$, k is the number of q-grams extracted from the original signature and $t \in [0, 1)$ is a threshold that reduces the number of combinations as its value increases.

4) *Suffix Arrays Blocking.* This method also captures character-level errors in the signatures of Standard Blocking by considering all their suffixes that are longer than a minimum number of characters l_{min}. Every block corresponds to such a suffix, provided that it is shared by up to b_{max} entities (i.e., very frequent suffixes are ignored).

5) *Extended Suffix Arrays Blocking.* This algorithm generalizes the previous one by defining as signatures all substrings of Standard Blocking's signatures that are longer than l_{min} and less frequent than b_{max}+1 entities.

Example. To highlight the difference between these blocking methods, consider the attribute value "Joe Biden". Standard Blocking produces two signatures: {Joe, Biden}. With $q = 3$, Q-Grams Blocking yields four signatures: {Joe, Bid, ide, den}. For T=0.9, Extended Q-Grams Blocking combines at least two q-grams from each token to form five signatures: {Joe, Bid_ide_den, Bid_ide, Bid_den, ide_den}. Using l_{min}=3 and a large enough b_{max}, Suffix Arrays Blocking yields four signatures: {Joe, Biden, iden, den}, while Extended Suffix Arrays extracts seven: {Joe, Biden, Bide, iden, Bid, ide, den}.

6) *Overlap Blocker [25].* This method creates candidate pairs directly, i.e., in a single step. To this end, it extracts a set of tokens for each entity either as q-grams or by splitting on whitespace. In the former case, the user has to determine the size of q, which is a configuration parameter. Next, for each pair of entities, the method estimates the overlap of their token sets. If it is larger than a user specified threshold, the two entities are retained as candidates.

Note that we exclude two established methods: *Attribute Clustering Blocking* [26], because it is incompatible with the schema-based settings we are considering in Section 6, and *Sorted Neighborhood* [3, 10, 16], because our experiments demonstrate that it consistently underperforms the above methods. This should be attributed to its *redundancy-neutral nature*: the number of blocks shared by two entities is not correlated with their matching likelihood. Thus, Sorted Neighborhood is incompatible with block and comparison cleaning that could reduce its superfluous pairs [9, 12].

Block cleaning. We consider two methods that can be combined if the first one precedes the second. Both rely on the reasonable assumption that the larger a block is, the less likely it is to convey matching pairs that share no other block, because their signature corresponds to a stop-word. Both raise precision to a significant extent at a limited cost in recall.

1) *Block Purging [26].* It discards the largest blocks, which contain a large portion of the input entities. The cut-off point is determined automatically, based on the relative size of the blocks. No configuration parameter is involved in this process.

2) *Block Filtering [16].* The blocks of every entity are sorted in increasing size and only r% of the top (smaller) ones is retained, where r is the filtering ratio.

Comparison cleaning. We consider two established methods, both of which remove the redundancy from the input set of blocks. They are competitive with each other and only one can be added to a blocking workflow [12, 22]:

1) *Comparison Propagation [26].* It removes all redundant pairs from any set of blocks by retrieving the set of distinct candidates per input entity from its associated blocks. This is carried out automatically, without any configuration parameter. It discards no matches, thus increasing precision at no cost in recall.

2) *Meta-blocking [27].* It goes beyond Comparison Propagation by targeting both redundant and superfluous pairs. To this end, it leverages: (i) a *weighting scheme*, which estimates the matching likelihood of every candidate pair based on the blocks shared by its constituent entities, and (ii) a *pruning algorithm*, which uses these scores to detect the candidate pairs that are more likely to be matching and, thus, need to be part of the output, i.e., the restructured blocks.

Weighting Schemes. The rationale behind them is that the more and smaller blocks two entities share (i.e., the more and less frequent their common signatures are), the more likely they are to be matching. In fact, they estimate the matching likelihood of two entities, e_i and e_j, using one or more of the following evidence:

E1) The number of blocks each entity participates in, i.e., $|B_i|$ and $|B_j|$, where B_x stands for the set of blocks containing entity e_x.
E2) Their co-occurrence frequency, i.e., the number of blocks they have in common, denoted by $|B_i \cap B_j|$.
E3) The size of these blocks, i.e., the total number of entities they contain.
E4) The cardinality of these blocks, i.e., their total number of pairs.

E2 corresponds to the common blocks scheme (CBS), which is also denoted by W1. The combination of E1 and E2 lead to the following schemes which estimate the homonymous similarities between the lists of blocks associated with each pair:

W2) Cosine=$|B_i \cap B_j|/\sqrt{|B_i| \cdot |B_j|}$=$CBS/\sqrt{|B_i| \cdot |B_j|}$.
W3) Dice=$2 \cdot |B_i \cap B_j|/(|B_i| + |B_j|)$=$2 \cdot CBS/(|B_i| + |B_j|)$.
W4) Jaccard=$|B_i \cap B_j|/|B_i \cup B_j|$=$CBS/(|B_i| + |B_j| - CBS)$.

These four weighting schemes can be *size normalized* through E3. The rationale is that the fewer entities a block contains, the more distinctive is the information it conveys and, thus, it provides a stronger evidence for the matching likelihood of a candidate pair. In other words, the larger a block is, the more likely it is to correspond to noisy information (e.g., stop-words). In this context, E1 is redefined as: $SN\text{-}B_x = \sum_{b \in B_x} 1/|b|$, where SN stands for size normalization and $|b|$ for the number of entities in block b. Similarly, the weighting schemes W1-W4 are normalized as follows:

W5) SN-CBS=$\sum_{b \in B_i \cap B_j} 1/|b|$.
W6) SN-Cosine=$SN\text{-}CBS/\sqrt{SN\text{-}B_i \cdot SN\text{-}B_j}$.
W7) SN-Dice=$2 \cdot SN\text{-}CBS/(SN\text{-}B_i + SN\text{-}B_j)$.

W8) SN-Jaccard=$SN\text{-}CBS/(SN\text{-}B_i + SN\text{-}B_j - SN\text{-}CBS)$.

The same principles apply to the *cardinality normalization* with E4: the weight of every block is divided by the number of pairs it contains, based on the idea that fewer paris indicate more distinctive information, i.e., higher matching likelihood. In this context, E1 is redefined as: $CN\text{-}B_x = \sum_{b \in B_x} 1/||b||$, where CN stands for cardinality normalization and $||b||$ for the pairs in block b. Similarly, the weighting schemes W1-W4 are normalized as follows:

W9) CN-CBS=$\sum_{b \in B_i \cap B_j} 1/||b||$.
W10) CN-Cosine=$CN\text{-}CBS/\sqrt{CN - B_i \cdot CN - B_j}$.
W11) CN-Dice=$2 \cdot CN\text{-}CBS/(CN\text{-}B_i + CN\text{-}B_j)$.
W12) CN-Jaccard=$CN\text{-}CBS/(CN\text{-}B_i + CN\text{-}B_j - CN\text{-}CBS)$.

Note that W1, W4 and W9 were originally proposed in [27] and were tested in [18]. W5, W8 and W12 were proposed in [28], where they were used as features in Supervised Meta-blocking. Ours is the first work that applies them to unsupervised (meta-)blocking together with the new features we introduced above, i.e., cosine and dice similarity along with their normalizations (W2, W3, W6, W7, W10 and W11).

Finally, our experiments involve two more weighting schemes that were defined in [27] and were examined in [18]. Using B to denote the set of all blocks, $|B|$ its size, i.e., their number, $|V|$ the total number of distinct pairs in B and $|v_i|$ the distinct pairs involving e_i, they are formally defined as:

W13) ECBS=$|B_i \cap B_j| \cdot \log \frac{|B|}{|B_j|} \cdot \log \frac{|B|}{|B_j|} = CBS \cdot \log \frac{|B|}{|B_j|} \cdot \log \frac{|B|}{|B_j|}$.
W14) EJS=$JS \cdot \log \frac{|V|}{|v_j|} \cdot \log \frac{|V|}{|V_j|}$.

Both weighting schemes leverage a formula inspired by the TF-IDF weights in Information Retrieval [24]. The former uses it to combine E2 with E1, while the latter combines Jaccard with the distinct number of pairs per entity and overall. The underlying idea in both schemes is that the scarcity of blocks or pairs increases the matching likelihood of the corresponding entities.

Pruning Algorithms. The following algorithms have been proposed in the literature and are examined in this work [27, 29]: BLAST retains those candidate pairs whose weight exceeds the average maximum weight of their constituent entities; CEP and CNP retain the top-K pairs overall and per entity, respectively (K is automatically configured according to the characteristics of the input blocks); Reciprocal CNP (RCNP) retains the pairs that are ranked in the top-K positions of both their entities; WEP keeps all pairs with a weight higher than the overall average one; WNP discards pairs with a weight lower than the average one of both constituent entities; Reciprocal WNP (RWNP) retains pairs whose weight exceeds the average of both constituent entities.

4.2.1 Baseline blocking workflows

To highlight the impact of fine-tuning, our analysis includes the following three baseline blocking workflows with default parameters, common across all datasets.

1) *Parameter-free Blocking.* It combines the three aforementioned methods with no configuration parameter: Standard Blocking, Block Purging and Comparison Propagation. In other words, it constitutes a Standard Blocking workflow with no configuration.

2) *Default Q-grams Blocking.* The experimental analysis in [12] associated each of the above block building methods with default parameters, i.e., the configuration that achieves the highest average performance among those considered during the step-wise optimization. In [18], preliminary experiments showed that the best default configuration is that of Q-Grams Blocking. This workflow uses $q = 6$ for block building, Block Filtering with ratio=0.5 for block cleaning and $WEP + ECBS$ for comparison cleaning.

3) *Default JedAI Workflow.* Another blocking workflow with default parameters, but high performance across numerous datasets was proposed in [20]. It consists of Standard Blocking, Block Purging, Block Filtering with ratio=0.8 and CNP in combination with $W4$, i.e., the Jaccard similarity of the list of blocks associated with each entity. This workflow was not benchmarked in [18].

4.3 Sparse NN methods

This type includes set-based similarity joins methods, which represent each entity by a set of *tokens* such that the similarity of two entities is derived from their token sets. The similarity between two token sets A and B is computed through one of the following three measures, which are normalized in $[0, 1]$ [13]:

1. Cosine similarity $C(A, B) = |A \cap B|/\sqrt{|A| \cdot |B|}$.
2. Dice similarity $D(A, B) = 2 \cdot |A \cap B|/(|A| + |B|)$.
3. Jaccard coefficient $J(A, B) = |A \cap B|/|A \cup B|$.

In the schema-based settings, the tokens are extracted from the value of a specific attribute, whereas the schema-agnostic settings derive the tokens from the concatenation of all attribute values. In both cases, the tokens are defined as the character q-grams [30] (as in Q-Grams Blocking) or as the words that are delimited by whitespace (as in Standard Blocking). The duplicate tokens of an entity can be ignored or de-duplicated by attaching a counter to each token [31] (e.g., $\{a, a, b\} \rightarrow \{a_1, a_2, b_1\}$).

In this context, the candidate pairs are those satisfying the similarity of two entities according to some *matching principles* [32]. We combine two well-known

principles with all the aforementioned similarity measures and tokenization schemes [33]:

1) *Range join (ε-Join) [33].* It returns as candidates all entities with a similarity higher than or equal to a user-defined threshold ε. Numerous efficient algorithms for ε-Join between two collections of token sets have been proposed over the years [13, 34–40]. These algorithms yield the exact same set of candidates, but most of them are crafted for high similarity thresholds (above 0.5), which is not the case in ER (cf. Table X in [18]). For this reason, we exclusively apply *ScanCount* [41], which is inherently crafted for low similarity thresholds. At its core lies an inverted list, whose keys correspond to all tokens in the entity collection $\mathcal{E}_1$, while their values contain the corresponding entity ids. For every entity $e_i \in \mathcal{E}_2$, a query is posed to the index to lookup the respective set of tokens, with ScanCount performing merge-counts on the resulting posting lists. Then, it returns all pairs that exceed the similarity threshold ε.

2) *k-nearest-neighbor join (kNN-Join) [33].* This algorithm associates every entity in $e_i \in \mathcal{E}_2$ with the entities from $\mathcal{E}_1$ that have one of the k highest similarity scores. This means that e_i may be paired with more than k entities if some of them have the same distance from e_i. Note that the kNN-Join is not commutative, since the order of the join partners matters. This means that different results are obtained when indexing $\mathcal{E}_1$ and querying with $\mathcal{E}_2$ than when doing the opposite. An efficient technique that leverages an inverted list on tokens that are partitioned into size stripes is the *Cone* algorithm [42], which is crafted for label sets in the context of top-k subtree similarity queries. To increase the limited scope of the original algorithm, we adapted it to leverage ScanCount.

3) *Edit-based similarity joins.* On a slightly different line of research, a well-known string similarity measure is the *edit distance* [43], which is the minimum number of character insertions, deletions, or renames that transforms one string into the other. Computing the edit distance has quadratic run-time in the length of the strings. Join techniques for the edit distance strive to avoid edit distance computations by leveraging indices and filters [44]. We use *PassJoin* [45], which is based on the pigeon-hole principle, as it is the state-of-the-art in the field [14]. Given that the order of words plays an important role in the computation of edit distance (unlike cosine, dice and Jaccard similarity), this approach is exclusively applicable to schema-based settings.

4) *Rule-based Blocker [25].* This approach defines any pair of entities, that fulfill a blocking rule as candidates. Each rule extracts the signatures representing the two entities e_i and e_j, calculates their similarity using a specific similarity measure and compares them against a user-defined threshold. The method of comparison (larger, smaller or equal) can also be chosen. If the rule is not fulfilled, they pair $< e_i, e_j >$ is treated as a candidate pair. We consider two tokenizers: the white space one, which uses the same signatures as Standard Blocking, and q-grams, which corresponds to Q-Grams blocking. These tokenizers are combined with the Jaccard and the cosine similarity. Each pair of tokenizer and similarity function is tested against a series of thresholds, which are defined in $[0, 1]$ with a varying stepsize. At thresholds which yield recalls close to the recall levels of interest, the step size was 0.001. This method is very similar to the Range Join method described further below. However there are no options for the user to optimize for efficient evaluation of blocking rules, like stop-word elimination.

4.3.1 Baseline sparse NN workflow

We consider one workflow with default configurations:

1) *Default kNN-Join.* As shown in [18], kNN-Join typically outperforms ϵ-Join, while being straightforward to configure, since it constitutes a deterministic, cardinality-based approach. Table X in [18] reports that its best performance is usually achieved when it is combined with cosine similarity, pre-processing to clean the attribute values from stop-words and perform stemming as well as a few nearest neighbors per query entity, K. Based on this evidence, this baseline method uses the aforementioned configuration, while setting K=5 and the smallest input dataset as the query set. In these settings, the multiset of character 5-grams ($C5GM$) exhibits the highest effectiveness, on average, among the available representation models. Thus, $C5GM$ is selected as the default one.

4.4 Dense NN methods

We distinguish the methods of this type into two categories, based on the way they extract the dense vector from every entity: the *endogenous* techniques rely exclusively on the textual representation of entities, whereas the *exogenous* ones leverage external knowledge in the form of pre-trained language models.

Endogenous methods. Most methods of this type rely on *Locality Sensitive Hashing* (LSH) [46,47], a cornerstone approach to approximate nearest neighbor search in high-dimensional spaces. In essence, it detects entities/vectors that are within $c \cdot R$ distance from a query vector, where $c > 1$ is a real number that represents a user-specified approximation ratio, while R is the maximum distance of any nearest neighbor vector from the

query. LSH is a popular filtering method for ER [48–51], due to its sub-linear query performance, which is coupled with a fast and small index maintenance, and its mathematical guarantee on the query accuracy. We consider three popular versions, all of which tokenize the selected attribute values on whitespace and extract the vectors from the resulting token sets:

1) *MinHash LSH* [52,53]. This technique represents each entity as a minhash of its token set, i.e., a sequence of hash values that are derived from the minimum values of random permutations. The minhashes are formed by a series of bands that comprise an equal number of rows, which lay the ground for estimating the Jaccard similarity of the original token sets. The relative size of bands and rows actually plays a decisive role in the resulting performance: few bands with many rows lead to collisions between pairs of objects with a very high Jaccard similarity, whereas many bands with few rows favor pairs of objects with very low similarity. More formally, this approach acts as a high-pass filter that approximates the probability that two objects share the same hash value with $(1/\#bands)^{(1/\#rows)}$, where $\#bands$ and $\#rows$ denote the number of bands and rows, resp.

2) *Hyperplane LSH* [54]. The vectors are assumed to lie on a unit hypersphere divided by a random hyperplane at the center, formed by a randomly sampled normal vector r. This creates two equal parts of the hypersphere with $+1$ on the one side and -1 on the other. A vector v is hashed into $h(v) = sgn(r \cdot v)$. For two vectors v_1 and v_2 with an angle α between them, the probability of collision is $Pr[h(v_1) = h(v_2)] = 1 - \frac{\alpha}{\pi}$

3) *Cross-Polytope LSH* [55]. It is a generalization of Hyperplane LSH. At their core, both methods are random spatial partitions of a d-dimensional unit sphere centered at the origin. The two hash families differ in how granular these partitions are. The cross-polytope is also known as an l1-unit ball, where all vectors on the surface of the cross-polytope have the l1-norm. The hash value is the vertex of the cross-polytope closest to the (randomly) rotated vector. Thus, a cross-polytope hash function partitions the unit sphere according to the Voronoi cells of the vertices of a randomly rotated cross-polytope. In the 1-dimensional case, the cross-polytope hash becomes the hyperplane LSH family.

4) *Random Join.* This approach is fundamentally different from the above LSH variants, but also relies on the original textual representation of entities, while being stochastic, too. To convert the input data into a dense numeric vector, it applies the first three methods of the default JedAI workflow, i.e., Standard Blocking, Block Purging and Block Filtering with its default filtering ratio ($r = 0.8$). The cleaning process (with stopword removal and stemming) may precede these methods. The resulting set of blocks defines the dimensionality of the dense vector that represents the input data. Every dimension d_i corresponds to a blocking signature t_i and is weighted according to its inverse frequency, i.e., $w(t_i) = 1/|b_i|$, where $|b_i|$ denotes the number of entities in block b_i (as in the Meta-blocking weighting schemes). Thus, the smaller a block is, the higher is its weight.

Dense vectors are also used inside the blocks. More specifically, every block is associated with two vectors: one with the entities from $\mathcal{E}_1$ and one with those from $\mathcal{E}_2$. In these vectors, entity $e_i \in \mathcal{E}_y, y \in \{1, 2\}$ is weighted with the function $w(e_i) = 1/\sum_{b \in B_i} ||b||$, where B_i stands for the set of blocks containing e_i and $||b||$ for the number of entity pairs in block b. That is, the fewer candidate pairs are contained in the blocks involving e_i, the higher is the corresponding weight w_i.

Random Join enerates the candidate pairs using these vectors according to two user-defined parameters: (i) the query entity collection $\mathcal{E}_q$, and (ii) the maximum number of candidates per query entity. For every entity in $\mathcal{E}_q$, one of its associated blocks/dimensions d_q is randomly selected with a probability that is proportional to its weight, i.e., the rarer a signature is among both input entity collections, the more likely it is to be selected. Next, in the selected block/dimension d_q, one of the entity/dimensions of the vector corresponding to $\mathcal{E}_i, i \neq q$, is randomly selected with a probability that is proportional to its weight. That is, the fewer pairs share this entity/dimension, the more likely it is to be selected. The resulting candidate pair is ignored, if it is redundant, i.e., it has been randomly selected before. In case no non-redundant pair is selected after 10 iterations, the processing moves to the next query entity to avoid a stalemate.

Exogenous methods. We consider three frameworks that rely on pre-trained language models:

1) *FAISS [56].* To use this approach, we first convert the textual representation of each entity into a 300-dimensional vector using pre-trained fastText embeddings. After indexing the vectors of the one input entity collection $\mathcal{E}_x$, it associates every query vector q from the other collection $\mathcal{E}_y$ with the k entries from $\mathcal{E}_x$ with the smallest distance from q. Two approximate methods are provided: (i) a hierarchical, navigable small world graph method, and (ii) a cell probing method with Voroni cells that can be optionally combined with product quantization. In line with [56], the experiments in [18] verified that both of them underperform the Flat index, which offers an exact solution. As a result, we exclude the approximate methods as well as the range/similarity search, which consistently underperforms the kNN search.

2) *SCANN [57].* Similar to the previous technique, this approach leverages fastText embeddings, while implementing a versatile framework with very high throughput [15]. This framework supports two main similarity measures, dot product and Euclidean distance, as well as two types of scoring: brute-force, which performs exact computations, and asymmetric hashing, which performs approximate computations, trading higher efficiency for slightly lower accuracy. In all cases, SCANN leverages partitioning, i.e., it trains its index over the indexed vectors so as to split them into disjoint sets. Thus, every query is answered by examining the scores only inside the most relevant partitions.

3) *DeepBlocker [11].* This is the state-of-the-art filtering method that leverages deep learning, as it consistently outperforms all others, such as AutoBlock [50] and DeepER [51]. It relies on fastText for the creation of the embedding vector per entity and on FAISS for indexing and querying. Unlike the previous approaches of this type, it performs learning in order to train a tuple embedding module. Its goal is to aggregate the individual embedding vectors that are associated with each entity (i.e., the vectors per token and/or value) into a single representative vector. Note that the labelled instances are automatically generated through self-supervised learning, thus minimizing their cost (i.e., they require no human intervention). DeepBlocker conveys several different modules. Among them, the Autoencoder constitutes the most effective under the schema-based settings, while ranking second in the schema-agnostic ones [11]. Given that it lies in very close distance of the top-performing Hybrid module, while exhibiting much higher time efficiency, we exclusively combine DeepBlocker with the Autoencoder module.

4.4.1 Baseline dense NN workflow

We use one workflow with default configurations:

1) *Default DeepBlocker.* The experiments in [18] show that DeepBlocker consistently outperforms all other dense vector-based NN methods in terms of effectiveness (cf. Figure 10). In most cases, DeepBlocker works best when cleaning the attribute values with stemming and stopword removal, when using the smallest input dataset as the query set and when using a small number of candidates per query. We set K=5 so that it is directly comparable with its spare NN counterpart, i.e., the Default kNN-Join.

4.5 Overview

Given the large number of methods per type, it is worth discussing and clarifying the relations between them.

Table 1 The scope per type of filtering methods. SB and SA stand for schema-based and schema-agnostic settings, resp.

Scope		Blocking	Sparse NN		Dense NN
			edit-based	set-based	
Syntactic Representation	SB	✓	✓	✓	✓
	SA	✓	-	✓	✓
Semantic Representation	SB	-	-	-	✓
	SA	-	-	-	✓

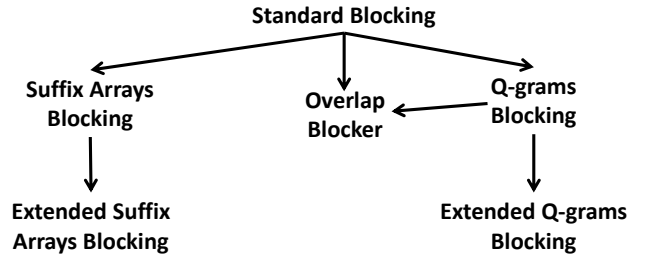

Fig. 3 The genealogy tree of blocking building techniques.

Table 2 The scope per type of filtering methods. SB and SA stand for schema-based and schema-agnostic settings, resp.

Scope		Blocking	Sparse NN		Dense NN
			edit-based	set-based	
Syntactic Representation	SB	✓	✓	✓	✓
	SA	✓	-	✓	✓
Semantic Representation	SB	-	-	-	✓
	SA	-	-	-	✓

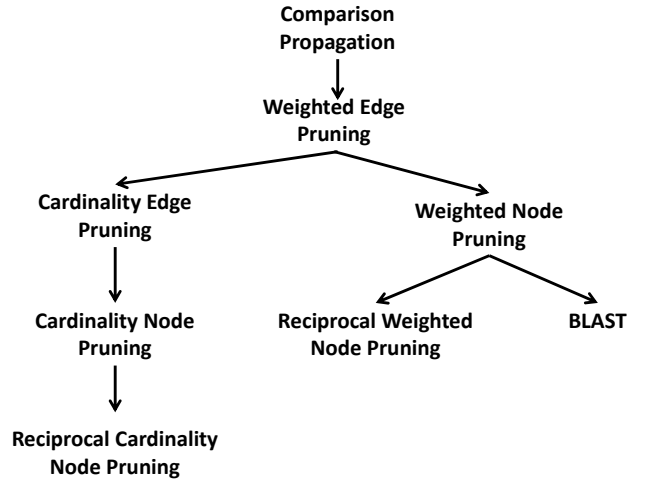

Fig. 4 The genealogy tree of comparison cleaning techniques.

Starting with block building, Figure 3 presents the genealogy tree of its techniques, with every edge pointing from the original one to its improvement. The cornerstone approach is Standard Blocking, which uses as signatures the tokens resulting from whitespace tokenization. Three methods alter them so as to accommodate typographical errors, trading higher recall for lower precision: (i) Q-Grams Blocking, which transforms the signatures of Standard Blocking into character n-grams; (ii) Suffix Arrays Blocking, which transforms them into suffixes longer than a minimum length; (iii) Extended Suffix Arrays Blocking, which is more noise-

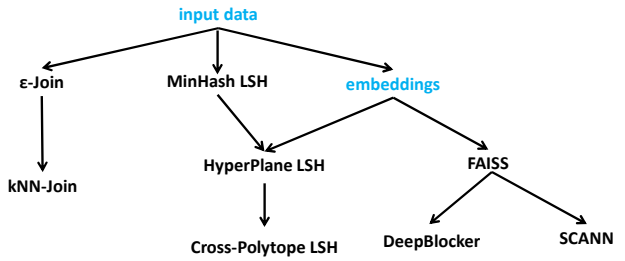

Fig. 5 The genealogy tree of sparse and dense NN methods, on the left and right respectively. Every edge points from an original approach to an extension that alters the type of thresholds or the way the input data is handled.

tolerant than all other methods, as it considers all substrings longer than a minimum length. In contrast, Extended Q-Grams Blocking emphasizes precision at the cost of slightly lower recall, combining multiple signatures of Q-Grams Blocking into a new one.

Overlap Blocker uses the signatures of Standard or Q-Grams Blocking, but differs from the rest of block building techniques in that it incorporates all subsequent workflow steps. A boolean, user-defined parameter turns block cleaning on and off. Yet, unlike the adaptive pruning of Block Purging and Block Filtering, which depends on data characteristics, Overlap Blocking performs a static cleaning: it excludes a predefined list of stop-words from the signatures of each input entity. Comparison cleaning is alway applied through a combination of the CBS weighting scheme with the WEP pruning algorithms. However, instead of using an adaptive threshold for candidate pairs based on the characteristics of the blocks at hand (as WEP does), the threshold is a-priori determined by the user.

The genealogy tree of comparison cleaning techniques appears in Figure 4, where every edge points from the original technique to the one alters the pruning threshold. The cornerstone approach is Comparison Cleaning, which removes all redundant pairs, increasing precision at no cost in recall. All other methods trade slightly lower recall for significantly higher precision. WEP is the simplest one, since it uses a *global similarity threshold*, discarding all pairs with a lower weight. This is replaced by a *local* similarity threshold in the right branch of the family tree, whereas the left branch includes pruning algorithms that replace it with a cardinality threshold. This sets the maximum number of retained pairs and is applied globally (CEP) or locally, i.e., per entity (CNP, Reciprocal CNP).

Note that the top-k set similarity joins [58,59] compute the k entity pairs between $\mathcal{E}_1$ and $\mathcal{E}_2$ with the highest similarities among all possible pairs. This means that they perform a global join that returns the k top-weighted pairs. This is equivalent to ε-Join, if the k^{th} has a similarity equal to ε. Instead, the kNN-Join performs a local join that returns at least k pairs per element $e_i \in \mathcal{E}_2$.

Note that FAISS and SCANN also use 300-dimensional fastText embeddings. In fact, they are equivalent to the simple average tuple embedding module of DeepBlocker.

Every edge points from the original technique to the one that improves it, i.e., it uses transforms the blocking keys into a more noise-tolerant format (in the former case), or alters the threshold or the algorithm that is used for pruning candidate pairs (in the latter case).

Note that neither Overlap nor Rule-based Blocker are described in a dedicated publication. We retrieved their description from their implementation and the documentation of their API.

Baseline per type to demonstrate the utility of parameter fine-tuning.

5 Qualitative Analysis

Taxonomies. We coin two novel taxonomies that facilitate the understanding and use of filtering methods.

Scope. This criterion distinguishes the entity representation at the core of filtering methods into two types:

1. *Syntactic or symbolic representations* leverage the co-occurrences of tokens or character n-grams, which are derived from the actual text in an entity profile.
2. *Semantic representations* transform the textual values into embedding vectors that are derived from word-, character- or (Sentence)BERT-based models. The former, which include Word2Vec [] and GloVe [], usually suffer from out-of-vocabulary cases in ER tasks, due to their domain-specific terminology. On the other hand, the (Sentence)BERT-based typically require labelled instances for their fine-tuning, while involving complex, time-consuming neural models. To address these issues, we exclusively consider the unsupervised, pre-trained embeddings of fastText [60], which are widely used in the literature in ER tasks, due to their high effectiveness [11,61–64].

Combining these two types with the two schema settings results in the four scope fields shown in Table 2.

These distinctions are essential for two reasons:

(i) Unlike the semantic representations, which are incomprehensible to non-experts, the syntactic ones yield models that are interpretable, in the sense that it is straightforward to justify candidate pairs.

(ii) The overhead of semantic representations, when transforming textual values into embeddings, is signifi-

Table 3 Functionality per NN method.

Operation	Threshold	
	Similarity	Cardinality
Deterministic	ε-Join	kNN-Join FAISS SCANN
Stochastic	MH-, HP-, CP-LSH	DeepBlocker

cant, while requiring external resources, too. These resources are typically loaded into main memory, raising the space complexity of filtering. In contrast, the syntactic representation methods are directly applicable to the input text without any run-time or space overhead.

In this context, the dense NN methods offer the greatest versatility, covering all four scope fields. MinHash LSH relies on syntactic representations, whose dimensions are determined by character k-grams, known as k-shingles, which are weighted according to the term frequency [53]. The rest of the dense NN methods use the semantic representations offered by fastText.

In contrast, the edit-based sparse NN methods have the most narrow scope, as they are compatible only with schema-based syntactic representations. They cannot be applied to the schema-agnostic ones for two reasons: (i) the arbitrary order of words. The attribute values are concatenated into a long text in no particular order and, thus, two consecutive words might originate from different attributes, having no semantic relation. This violates the core assumption of edit-based joins, because they are sensitive to the order of tokens and characters. For example, the distance of "John Smith" from "Smith John" is the same (10 edits) as that from the random string "xYXS1.fklt". (ii) the length difference of strings that should be similar may vary to a great extent. This has a major impact on edit distance, because missing characters must be inserted. For example, "John" is more similar to "Mike" (4 edits) than to "John Smith" (6 edits). Hence, the edit-based joins are only suitable for comparing short textual values from a specific attribute, e.g., to detect typos.

Finally, the blocking and the set-based sparse NN methods cover both settings syntactic similarities, as they operate directly on the input data, while being agnostic to the word order in attribute values.

Internal functionality. The second taxonomy relates to the internal functionality of filtering methods. The blocking workflows are already divided into *lazy* and *proactive* ones based on their block building phase: the former create blocks without any restrictions, whereas the latter refine their initial blocks, e.g., by imposing a limit on their maximum size – see [12] for more details.

In this work, we define a taxonomy for the NN workflows, which consists of the two dimensions in Table 3:

1. The *type of operation* can be stochastic or deterministic, depending on the use or lack of randomness.
2. The *type of threshold* can be similarity-based, specifying the minimum similarity score for a pair of candidates, or cardinality-based, determining the maximum number of candidates per query entity.

These distinctions are essential for two reasons:

(i) In contrast to deterministic methods, which yield stable performance in every run, stochastic methods produce slightly different results in every run. This is particularly important in the context of Problem 1, which sets specific limits on recall. To account for this, we set the performance of stochastic methods as the average one after 10 repetitions.

(ii) Cardinality-based methods can be easily and a-priori configured, since they primarily depend on the number of input entities. Similarity-based methods, on the other hand, depend on the characteristics of a dataset, in particular, the distribution of similarities.

Configuration space. To maximize filtering performance, it is necessary to fine-tune configuration parameters. To this end, we apply grid search, combining each method with a wide range of parameter values. Tables 4, 5 and 6 provide a listing of the domains we considered per parameter and method.

We begin with the configuration space for the blocking workflows in Table 4. Lazy blocking techniques have two common parameters: whether or not Block Purging was applied and the ratio r for Block Filtering. Grid search tests all values for r from 1 until 0 with a step of -0.025, i.e., 40 configurations in total. However, as soon as recall drops below the target threshold τ, grid search is terminated, resulting in a reduced number of examined configurations. The reason is that block cleaning bounds the maximum recall of comparison cleaning. Thus, configurations violating τ will not produce results that satisfy Problem 1.

Comparison cleaning is common to all blocking workflows. Two are the possible configurations: the parameter-free Comparison Propagation (CP) or one of the 98 Meta-blocking configurations resulting from the 14 weighting schemes and the seven pruning algorithms.

The Standard Blocking workflow has the fewest configurations, as it only involves the common parameters. For the other block building methods, we consider the configurations examined in [12]. Noted that Suffix Arrays Blocking and its extension are proactive workflows, thus incorporating no block cleaning method.

Regarding the Overlap Blocker, it constitutes a proactive method that incorporates a single comparison cleaning technique (WEP with W_1/CBS weights). Yet, instead of using an automatically configured threshold, as all other meta-blocking techniques, its overlap thresh-

Table 4 The configuration space per blocking workflow.

	Parameter	Domain	Configurations
Common for lazy	Block Purging (BP) Block Filtering ratio (BFr)	{ -, ✓} [0.025, 1.00] with a step of 0.025	2 40
Common for all	Weighting Scheme (WS) Pruning Algorithm (PA)	{W_1-W_{14}} {BLAST, CEP, CNP, RCNP, RWNP, WEP, WNP} or CP	14 7 + 1
(a) lazy methods			
Standard Blocking	–	–	7,920
Q-Grams Blocking	q	$[2, 6]$ with a step of 1	39,600
Extended Q-Grams Blocking	q t	$[2, 6]$ with a step of 1 $[0.8, 1.0)$ with a step of 0.05	158,400
(b) proactive methods			
(Ex.) Suffix Arrays Blocking	l_{min} b_{max}	$[2, 6]$ with a step of 1 $[2, 100]$ with a step of 1	39,600
Overlap Blocker	Stop-word removal Tokenization Overlap Threshold	{ -, ✓} Whitespace or Q-Grams with $q \in [2, 6]$ with a step of 1 [1, 100] with a step of 1	2 6 100

old is determined by the user. We actually consider all values in $[1, 100]$ with a step of 1. Two more parameters are used by Overlap Blocker: stop-word removal (in the place of block cleaning) and tokenization, for which we consider six options: whitespace tokenization, as in Standard Blocking, or q-grams tokenization, as in Q-Grams Blocking. Overall, the Overlap Blocker involves up to 1,200 different configurations, but in practice, slightly more than half were tested per dataset, as the overlap threshold is consistently lower than 63.

The Rule-based Blocker has similar parameters than ε-Join. However, cleaning is not possible within the package. We consider only two similarity measures (cosine and jaccard) and two representation models (whitespace tokenization and character n-grams with $n = 3$). The minimum threshold for which PC reaches the target recall was determined with a step size of 0.001. To find this threshold, the definition range was searched with a varying stepsize.

The parameters for the sparse NN workflows can be found in Table 5. The common parameters are cleaning (i.e. removing stop words and stemming), the similarity measure, and the representation model. We consider all options discussed in Section 4.3 for the last two parameters. These include three similarity measures - cosine, Dice and Jaccard - and 10 representation models: whitespace tokenization (T1G) or character n-grams (CnG) with $n \in \{2, 3, 4, 5\}$; for each representation model, we use the set or the multiset of its items (the latter is denoted by appending an M to its name).

As an additional parameter, ε-Join is combined with up to 100 similarity thresholds. The grid search is initiated with the largest one and terminates when PC drops below the target recall. In kNN-Join, there are at most 100 cardinality thresholds. Here, the grid search starts with the smallest one and terminates as soon as PC exceeds the target recall. RVS is another key parameter for kNN-Join, determining whether we should reverse the indexing and the query entity collection or not; if RVS is true, we index $\mathcal{E}_2$ and query with $\mathcal{E}_1$. Despite the fact that kNN-Join theoretically has double the number of configurations compared to ε-Join, its cardinality threshold does not exceed 26 in practice (see Table X in [18]), and thus the maximum number of its configurations is significantly reduced.

A list of the parameters of dense NN workflows can be found in Table 6. The common parameter is whether cleaning is applied or not. As we noted previously, additional parameters for MinHash LSH are the number of bands and rows. They are chosen as powers of two such that the product of both is 2^n with $n \in \{7, 8, 9\}$. For the k-shingles in MinHash LSH, we consider four values for k, namely $[2, 5]$. In the case of Hyperplane and CrossPolytope LSH, we configure two parameters: (i) the number of hash tables ($\#tables$), i.e., the number of cross-polytopes and the number of hyperplanes, and (ii) the number of hash functions ($\#hashes$). During testing, we used the ranges reported in Table 6, since values outside the given intervals significantly increased the query time for a marginal increase in precision. The number of probes for multi-probe was automatically set to achieve the target recall using the approach in [65][do we really want to ignore them completely in the parameter count? I didn't count the number of candidates, because they didn't fullfill the recall requirement, but otherwise there is no difference to the settings I looked at. Additionally I had to manually finetune the number of probes for the chosen settings, so that they are stable]. The last cp dimension is specified only for CrossPolytope LSH, and should be between 1 and the smallest power of two larger than the dimension of the embeddings vector (here 512) [66].

Table 5 The configuration space per sparse NN method.

	Parameter	Domain	Config.
Common	Cleaning (CL)	{ -, ✓}	2
	Sim. Measure (SM)	{Cosine, Dice, Jaccard}	3
	Rep. Model (RM)	{T1G, T1GM, C2G, C2GM, C3G, C3GM, C4G, C4GM, C5G, C5GM}	10
ε-Join	Similarity threshold (t)	[0.00, 1.00] with a step of 0.01	6,000
kNN-Join	Cand. per query (K)	[1, 100] with a step of 1	12,000
	Reverse Datasets (RVS)	{ -, ✓}	
Rule-based Blocker	Cleaning	–	
	Sim. Measure	{Cosine, Jaccard}	
	Rep. Model	Whitespace or Q-Grams with $q = 3$	
	Similarity threshold	$[0, 1]$ with varying step size and a minimum step of 0.001	

Table 6 The configuration space per dense NN method.As the caption of the table is above the table, I would also put the (a) and (b) captions above. For me it would be less confusing then

	Parameter	Domain	Configurations
Common	Cleaning (CL)	{ -, ✓}	2
MH-LSH	$\#bands$, $\#rows$	$\#bands \in P(2), \#rows \in P(2) : \#bands \times \#rows \in \{128, 256, 512\}$	168
	k	$[2, 5]$ with a step of 1	
HP- & CP-LSH	$\#tables$	$2^n : n \in \{0, 9\}$	200 (HP), 400 (CP)
	$\#hashes$	[2, 20] with a step of 2 (HP); [1, 2] (CP)	
	last cp dimension	$2^n : n \in \{0, 9\}$	

(a) Threshold-based algorithms

Common	Rev. Datasets (RVS)	{ -, ✓}	2
	K	[1, 100] with step=1, [105, 1000] with step=5, [1010, 5000] with step=10	5,000
FAISS			2,720
SCANN	index	{ AH, BF }	10,880
	similarity	{ DP, LP^2 }	
DB			2,720

(b) Cardinality-based algorithms

Table 7 Real datasets for Clean-Clean ER. $|D|$ and $|E_x|$ denote the number of duplicates and entities in dataset E_x, respectively.

	D_{c1}	D_{c2}	D_{c3}	D_{c4}	D_{c5}	D_{c6}	D_{c7}	D_{c8}	D_{c9}	D_{c10}
E_1	Rest. 1	Abt	Amazon	DBLP	IMDb	IMDb	TMDb	Walmart	DBLP	IMDb
E_2	Rest. 2	Buy	GB	ACM	TMDb	TVDB	TVDB	Amazon	GS	DBpedia
$\|E_1\|$	339	1,076	1,354	2,616	5,118	5,118	6,056	2,554	2,516	27,615
$\|E_2\|$	2,256	1,076	3,039	2,294	6,056	7,810	7,810	22,074	61,353	23,182
$\|D\|$	89	1,076	1,104	2,224	1,968	1,072	1,095	853	2,308	22,863
$\|E_1\| \times \|E_2\|$	$7.65{\cdot}10^5$	$1.16{\cdot}10^6$	$4.11{\cdot}10^6$	$6.00{\cdot}10^6$	$3.10{\cdot}10^7$	$4.00{\cdot}10^7$	$4.73{\cdot}10^7$	$5.64{\cdot}10^7$	$1.54{\cdot}10^8$	$6.40{\cdot}10^8$
Best Attr.	Name	Name	Title	Title	Title	Name	Name	Title	Title	Title

The cardinality-based dense NN-methods share two parameters with kNN-Join (Table 5): RVS, and K, the cardinality threshold. As in kNN-Join, we consider all values within $[1, 100]$ with a step of 1 for the latter. Due to the fact that this is not sufficient in some datasets, we additionally consider all values between $[105, 1000]$ and $[1010, 5000]$ with steps of 5 and 10 respectively. Starting from the lowest value, the grid search ends whenever PC reaches the target recall.

In addition to the common parameters, FAISS has three further parameters, the distance function, whether or not the vectors should be normalized and the type of index. According to our experiments, the normalized embedding vectors have a better performance. In that case the two distance options, Inner Product and Euclidean distance are equivalent. In addition the Flat index should always be used.

SCANN has two additional parameters: the type of index (asymmetric-hashing (AH) or brute-force (BF)) and the similarity measure (dot product or Euclidean distance). Neither of these options is clearly superior, see Table **??**. This should be the table with the best configurations. Is it missing for a reason?

To conclude, DeepBlocker adds a tuple embedding model to the common parameters. In our experiment, we used both top-performance modules, namely AutoEncoder and Hybrid. The latter, however, raises out-of-memory exceptions in most cases, and is orders of magnitude slower than the former, as documented in [11]. Therefore, we only consider AutoEncoder in the following.

Take-away message. All the above filtering techniques involve three or more configuration parameters, which must be fine-tuned to reach the desired level of recall, while maximizing precision. This is a complex task, because typically several thousands of different settings have to be tested. Helpful in this task are parameters that are common among techniques of the same type, and, therefore, experience with one approach may facilitate the fine-tuning of another. This is the case with the block cleaning techniques, which are shared by all lazy blocking methods; for example, Block Purging is typically unnecessary for all these methods in schema-based settings [18].

Unfortunately, this does not apply to the configuration of comparison cleaning in blocking workflows, which is heavily data-driven. The best pruning algorithm and weighting scheme varies widely among the considered methods when applied to the same dataset under the same schema settings.

In contrast, some parameters of the NN workflows are intuitive, i.e., easy to configure. This is particularly true for the number of candidates per entity, which is the main parameter of cardinality-based NN methods. These methods provide the highest level of usability, especially when their operation type is deterministic. Among those, only kNN-Join uses syntactic representations, allowing interpretable decisions to be made in a similar fashion to the blocking workflows. The other methods with deterministic functionality and intuitive parameters are FAISS and SCANN, but they operate on semantic representations. DeepBlocker also utilizes a cardinality threshold, yet its tuple embedding model uses neural networks, which are randonly initialized and trained on random artificial data, rendering its functionality stochastic and less robust.

The deterministic similarity-based methods are ε-Join and Rule-based Blocker, whereas the three LSH methods are stochastic by definition. Hyperplane and Cross-Polytope LSH utilize random spatial partitions of a d-dimensional unit sphere, while MinHash LSH permutes token sets randomly. Thus, even though ε-Join and Rule-based Blocker have a larger configuration space, they are more usable.

6 Quantitative Analysis

Datasets. We use two sets of datasets that cover both main types of ER. For Clean-Clean ER, we use 10 real-world datasets that are popular in the literature [11, 12, 67, 68]. Their technical characteristics are reported in Table 7. D_1, which was first used in OAEI 2010 [69], contains restaurant descriptions. D_2 encompasses duplicate products from the online retailers Abt.com and Buy.com [68]. D_3 matches product descriptions from Amazon.com and the Google Base data API (GB) [68]. D_4 entails bibliographic data from DBLP and ACM [68]. D_5, D_6 and D_7 involve descriptions of television shows from TheTVDB.com (TVDB) and of movies from IMDb and themoviedb.org (TMDb) [70]. D_8 matches product descriptions from Walmart and Amazon [67]. D_9 involves bibliographic data from publications in DBLP and Google Scholar (GS) [68]. D_{10} interlinks movie descriptions from IMDb and DBpedia [20] – note that it includes a different snapshot of IMDb than D_5 and D_6.

For Dirty ER, we use seven synthetic datasets whose size increases from 10 thousand to 2 million entities. They have been widely used in the literature [12, 16, 71], as they are ideal for investigating the scalability of filtering techniques. They have been generated by Febrl [72] using the guidelines specified in [10]: first, duplicate-free entities describing persons (i.e., their names, addresses etc) were created based on frequency tables of real-world data. Then, duplicates of these entities were randomly generated according to real-world error characteristics and modifications. The resulting datasets contain 40% duplicate entities with up to 9 duplicates per entity, no more than 3 modifications per attribute, and up to 10 modifications per entity. Table 8 reports their technical characteristics.

Setup. All experiments were performed on a server with an Intel Xeon Gold 6238R @ 2.20GHz with 256GB of RAM, running Ubuntu 18.04.6 LTS. For most time measurements, we performed 10 repetitions and report the average value. These measurements exclude the time required to load the input data in main memory.

For most methods, we used existing, popular implementations. For blocking and sparse NN workflows, we employed JedAI's latest version, 3.2.1 [73]. The same library was used for Random Join. All experiments were run on Java 15. For Overlap and Rule-based Blocker, we used `py-entitymatching` (i.e., Magellan), version 0.4.0. For HP- and CP-LSH, we used the Python wrapper of FALCONN [66], version 1.3.1. For FAISS, we used version 1.7.2 of the Python wrapper provided by Facebook Research [74]. For SCANN, we used version 1.2.5 of the Python implementation provided by Google Research [75]. For DeepBlocker, we used the implementation provided by the authors [76]. FAISS, SCANN and DeepBlocker can exploit GPU optimizations, but all methods were run on a single CPU to ensure a fair comparison.

Schema settings. As explained above, the *schema-agnostic settings* apply filtering to all attribute values of each entity, regardless of their attribute names. In this way,

Table 8 The synthetic, Dirty ER datasets. $|D|$ and $|E|$ stand for the number of duplicates and entities, respectively.

	$\mathbf{D_{10K}}$	$\mathbf{D_{50K}}$	$\mathbf{D_{100K}}$	$\mathbf{D_{200K}}$	$\mathbf{D_{300K}}$	$\mathbf{D_{1M}}$	$\mathbf{D_{2M}}$
$\|E\|$	10,000	50,000	100,000	200,000	300,000	1,000,000	2,000,000
$\|D\|$	8,705	43,071	85,497	172,403	257,034	857,538	1,716,102
$\|E\| \times (\|E\|\text{-}1)/2$	$5.00{\cdot}10^7$	$1.25{\cdot}10^9$	$5.00{\cdot}10^9$	$2.00{\cdot}10^{10}$	$4.50{\cdot}10^{10}$	$5.00{\cdot}10^{11}$	$2.00{\cdot}10^{12}$

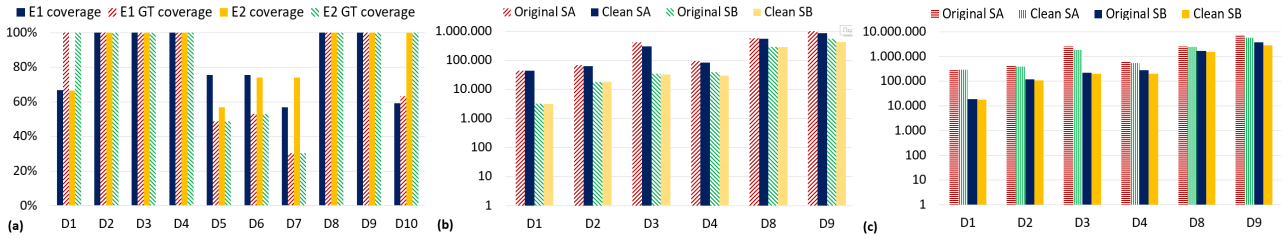

Fig. 6 (a) The coverage of the best attribute per each dataset, (b) the vocabulary size in schema-agnostic and schema-based settings, and (c) the overall character length in the textual content of the datasets for both schema settings.

they have a broad scope that covers all datasets. In contrast, the *schema-based settings* apply filtering to the values of a specific attribute, which usually appears in most entities, while being quite distinctive. As a result, these settings are inapplicable in datasets with a large portion of missing values and high levels of noise.

To explore the feasibility of the schema-based settings in the datasets of Table 7, we define two measures:

1. the *coverage* of attribute a, which is the portion of entities that contain a non-empty value for a, and
2. the *groundtruth coverage*of attribute a, which expresses the portion of duplicate profiles that have at least one non-empty value for a.

For each dataset, we estimate these measures for the attributes with the highest frequency and *distinctiveness*, i.e., the highest portion of different values among the given entities (e.g., an attribute like year for publications or movies has very low distinctiveness in contrast to their titles). The selected attributes are reported in the last row of Table 7.

Figure 6(a) reports these measures for the selected attribute per dataset. Half the datasets exhibit practically perfect scores for both measures: D_2, D_3, D_4, D_8 and D_9. Similarly, perfect score is achieved for the groundtruth coverage in D_1, too, even though the selected attribute appears in just 2/3 of the input entities. Low scores are observed in the remaining datasets. More specifically, in D_5, D_6 and D_7, the overall coverage fluctuates between 55% and 75%, dropping to 30%-53% for duplicates. Given that these scores are the highest among all available attributes, we can deduce that no filtering technique can satisfy any of the three PC thresholds specified in Section 3. As a result, we exclude the schema-based settings of D_5-D_7 from our analysis. The same applies to D_{10}, even though the inadequate coverage pertains only to one of its constituent datasets.

Note that the low coverage for distinctive attributes like "Name" and "Title" does not mean that there are entities missing the corresponding values. Their values are typically misplaced, associated with a different attribute, e.g., due to extraction errors [11,67]. *The schema-agnostic settings inherently tackle this form of noise, unlike the schema-based ones.*

For the datasets with both settings, we compare their computational cost with respect to two measures:

1. the *vocabulary size*, i.e., the total number of distinct tokens, and
2. the *overall character length*, i.e., the total number of characters.

These measures are reported in Figures 6(b) and (c), respectively, along with their values after *cleaning*, i.e., after removing the stop-words and stemming all tokens, as required by the workflow in Figure 2. We used the `nltk` package [77] for this purpose.

On average, the schema-based settings reduce the vocabulary size and the character length by 2/3, because the schema-agnostic ones typically include 3-4 name-value pairs. The more attributes and the more name-value pairs per entity a dataset includes, the larger is the difference between the two settings. Cleaning further reduces the vocabulary size by 11.9% and the character length by 13.5%, on average. Hence, compared to the schema-agnostic settings, *the schema-based ones trade a much lower scope for a significantly higher time efficiency, especially when combined with cleaning.*

6.1 Experimental Results

To assess the relative performance of the filtering techniques, we rely on the four evaluation measures defined in Section 3: PC (recall), PQ (precision). $|\mathcal{C}|$ (number of candidates), and RT (run-time). For brevity, PC

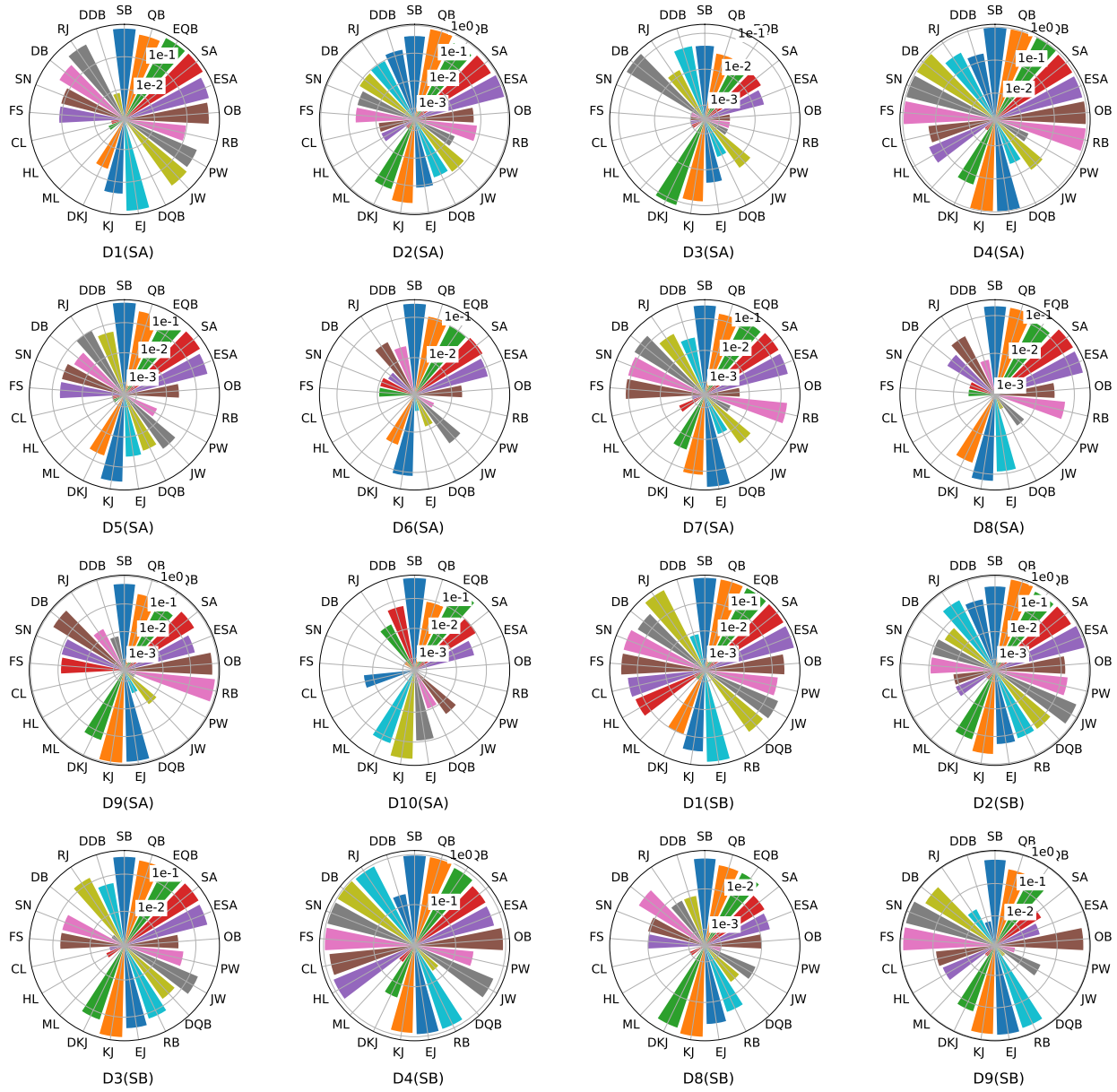

Fig. 7 *PQ* per method across all datasets in Table 7 under schema-agnostic (SA) and schema-based (SB) settings for $\tau = 0.85$.

and $|\mathcal{C}|$ per dataset, schema-settings and method are reported in the extended online version of our work []. The reason is that *PC* is essentially determined by the three recall targets (i.e., $\tau \in \{0.85, 0.90, 0.95\}$), with very few expectations that primarily correspond to baseline techniques with default configurations across all datasets (see Section 6.2 for more details); given that *PC* coincides with τ in the vast majority of cases, $|\mathcal{C}|$ can be deduced from *PQ*.

In this context, Figures 7, 8 and 9 report *PQ* per dataset, schema-settings and workflow for recall target $\tau = 0.85$, 0.90 and 0.95, respectively. To save space, we replace the full method names with the abbreviations in Table 9. Comparing the performance of the same workflows over the same datasets and settings across the three figures, we observe that as the target recall increases, as we move from Figure 7 to Figure 8 and 9, the corresponding precision decreases. This is expected, as more candidates are required in order to detect more duplicate pairs, thus raising recall.

6.1.1 Blocking vs NN workflows

Schema-agnostic settings. Starting with the relative performance of the three workflow types, we observe that the dense NN workflows are not the top performers in any datasets. Only for D_4, they are consistently very

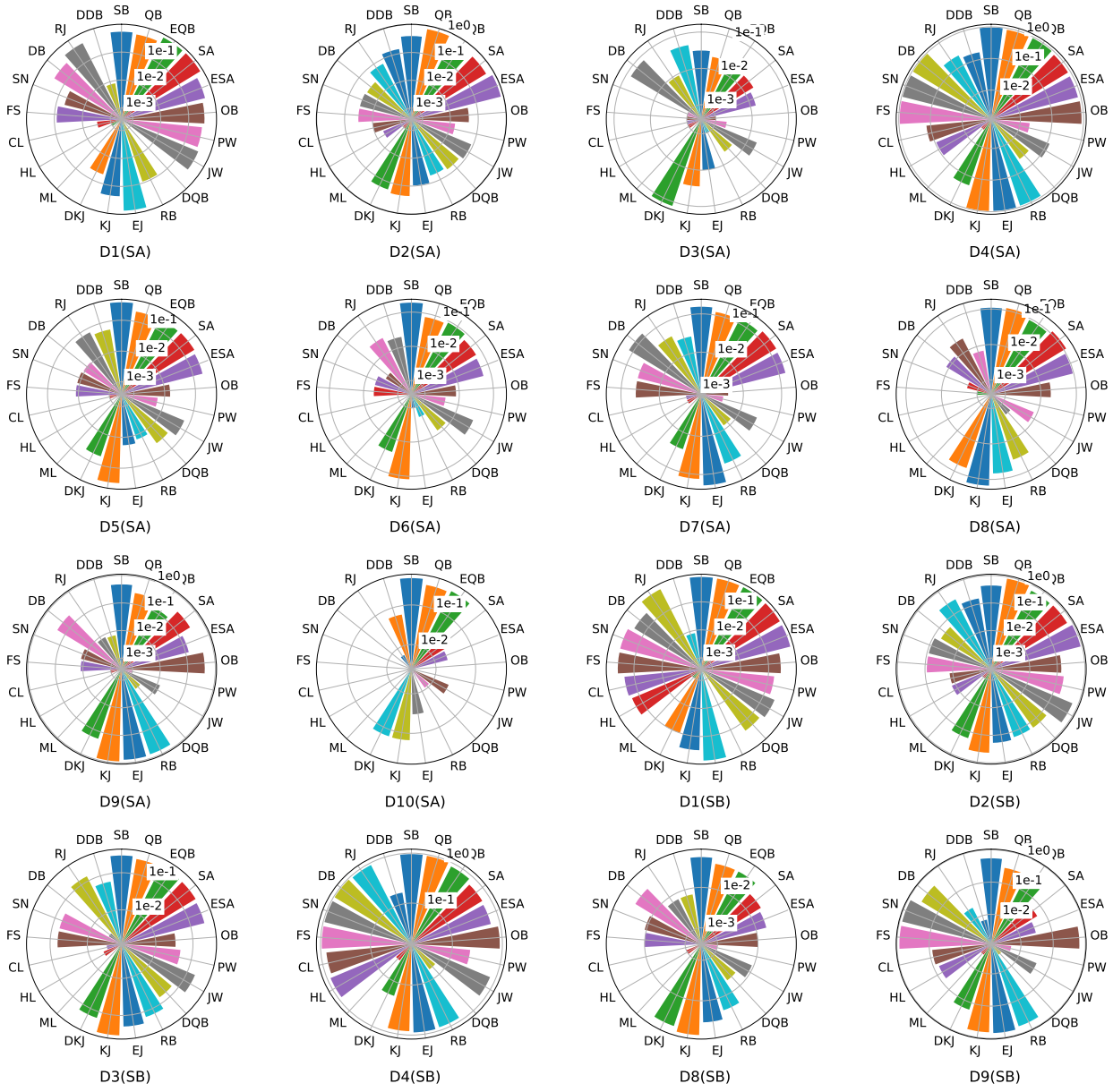

Fig. 8 *PQ* per method across all datasets in Table 7 under schema-agnostic (SA) and schema-based (SB) settings for $\tau = 0.90$.

close to sparse NN tau=0.85, which exhibit the highest precision, even by just 0.2% for $\tau = 0.90$ and $\tau = 0.95$. This applies, however, to the blocking workflows, as well, because D_4 contains clean bibliographic data, with all workflow types achieving almost perfect precision, i.e., $gg0.95$, in most cases – the only exception are the blocking workflows for $\tau = 0.95$, where their maximum precision drops to 0.9. Such a high performance renders matching dispensable, as the outcomes of blocking in this dataset can be considered as the output of the entire ER pipeline, regardless of the target recall. This pattern, however, does not appear in any other dataset. The second highest precision across the three target recalls is just 0.88, corresponding to D_9, which also includes relative clean bibliographic datasets.

For the remaining datasets, we observe that for six of them, the best performing workflow type is consistent across all τ. These include D_1, D_2, D_6, D_{10}, where blocking workflows are the top performers, with the sparse NN approaches in the second place. The former outperform the latter by at least 4%, with an average of 31%, 46% and 56% for $\tau = 0.85$, $\tau = 0.90$ and $\tau = 0.95$, respectively. This means that the lead of blocking workflows increases with the increase of the recall level, thus indicating that they are more suitable for applications requiring very high recall.

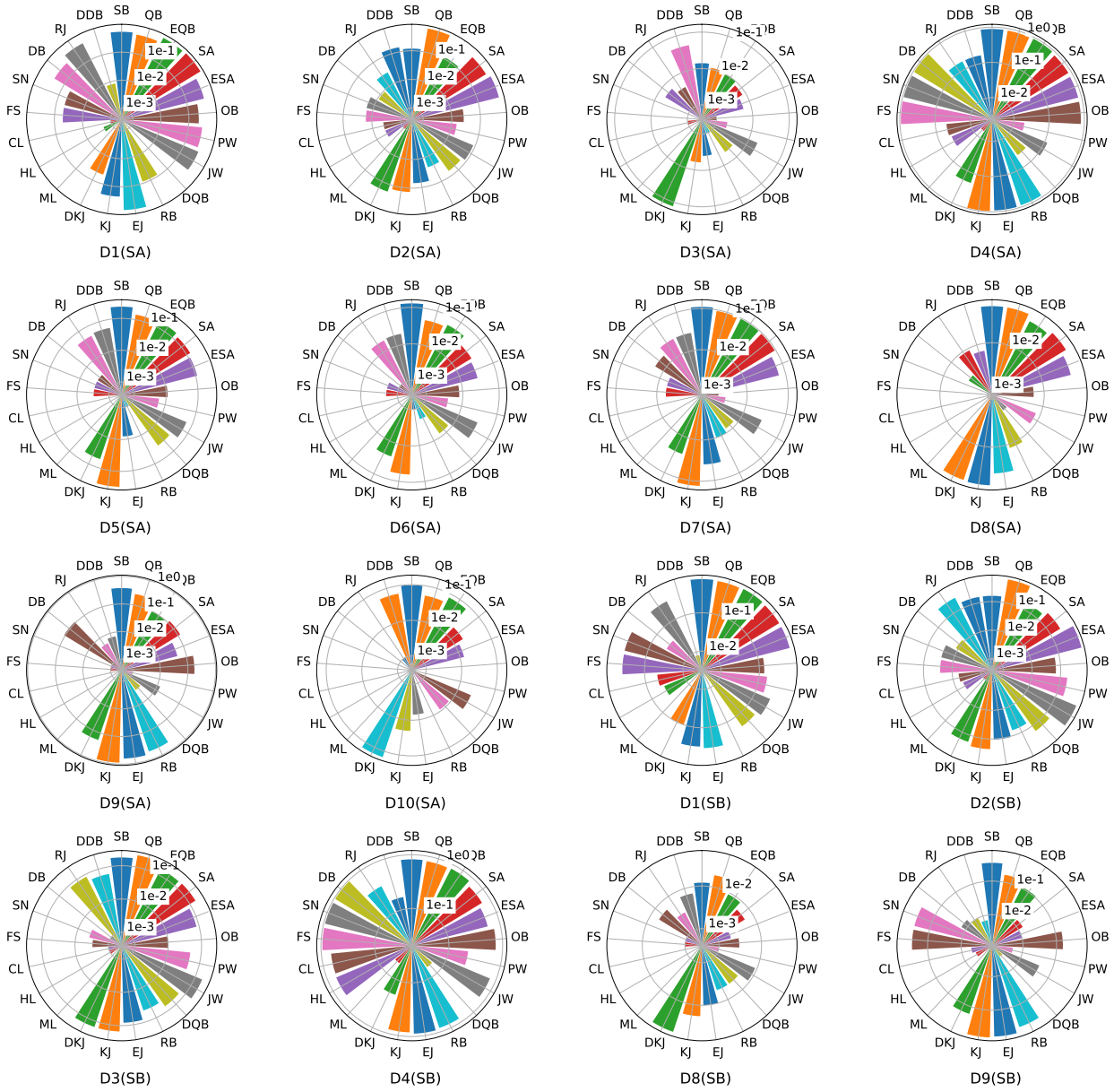

Fig. 9 *PQ* per method across all datasets in Table 7 under schema-agnostic (SA) and schema-based (SB) settings for $\tau = 0.95$.

The remaining two datasets with consistent patterns are D_7 and D_9, where the sparse NN workflows are the top performers, leaving the blocking ones in the second place. Their precision is higher by at least 15%, increasing proportionally with the increase of τ – from 24% for $\tau = 0.85$ to 33% and 37% for $\tau = 0.90$ and $\tau = 0.95$, respectively, on average. The reason is that both datasets contain relatively clean, long textual values that are ideal for detecting the most likely matches per record, as KJ does. D_4 could also be considered as a dataset dominated by sparse NN-based workflows, but, as explained above, the difference between the three types is rather minor.

For the remaining three datasets, there is a strong competition between the blocking and the sparse NN workflows. In D_3, the latter start with the highest by far precision, which however drops faster than that of the blocking workflows, as τ increases. As a result, blocking takes the lead for $\tau = 0.90$ and $\tau = 0.95$, due to its more robust performance. In absolute numbers, though, the blocking workflows outperform the sparse NN ones by less than 1%. The opposite is true for D_8, where the blocking workflow is the top performer for $\tau = 0.85$, but is superseded by the sparse NN methods for the next recall thresholds. The reason is that the configuration of KJ is the same for both $\tau = 0.85$ and $\tau = 0.90$, retriev-

Table 9 Name abbreviations of the considered methods.

Acronym	Method name
SB	Standard Blocking
QB	Q-grams Blocking
EQB	Extended Q-grams Blocking
SA	Suffix Arrays Blocking
ESA	Extended Suffix Arrays Blocking
OB	Overlap Blocker
PW	Parameter-free workflow
JW	JedAI's default blocking workflow [20]
DQB	Default Q-grams Blocking [12]
EJ	Range(ϵ)-Join
KJ	k-nearest-neighbor(kNN)-Join
RB	Rule-based Blocker
DKJ	Default kNN-Join
ML	MinHash(MH)-LSH
HL	Hyperplane(HP)-LSH
CL	Cross-Polytope(CP)-LSH
FS	FAISS
SN	SCANN
DB	DeepBlocker
RJ	Random Join
DDB	Default DeepBlocker

ing a single candidate for the smallest input dataset, which corresponds to the lowest number of candidate pairs. In other words, the sparse NN workflows cannot reduce the candidates when moving from $\tau = 0.90$ to $\tau = 0.85$, despite the lower target recall. In contrast, blocking workflows are more flexible in pruning candidate pairs, independently of the size of the input, thus achieving higher precision for the lowest τ. Finally, for D_5, the blocking workflows exhibit the highest precision for $\tau = 0.85$ and $\tau = 0.90$, but drop to the second place for $\tau = 0.95$. This is because their precision drops as the recall threshold increases (i.e., they generate more candidates in order to detect the necessary duplicates), while both measures remain the same for the sparse NN methods: for KJ, it suffices to retrieve a single candidate per record in the smallest input dataset to exceed 0.95 in terms of recall.

Overall, *we can conclude that the relative maximum performance of the three workflow types is relatively stable across the three recall thresholds. Blocking workflows are the preferred option, except for datasets with long textual values with relatively low levels of noise, such as D_4, D_7 and D_9. In such cases, the sparse NN methods yield much fewer candidates and, thus, higher precision.*

Schema-based settings. Similar patterns are observed for the first three datasets in the case of schema-based settings. More specifically, the blocking workflows are top performers for the two smallest datasets, D_1 and D_2, just like in the schema-agnostic settings. Moreover, there is a strong competition between the blocking and the sparse NN workflows in D_3, following exactly the same pattern as with the schema-agnostic settings: the latter outperform the former for $\tau = 0.85$, the two types are very close for $\tau = 0.90$, but blocking workflows are more effective for $\tau = 0.95$.

For D_4, the dense NN workflows achieve exactly the same precision as with their schema-agnostic settings ($PQ = 0.95$ across all τ). In contrast, the other two workflow types decrease their precision as the target recall raises, under-performing their schema-agnostic configuration in all cases – the only exception is the blocking workflows for $\tau = 0.85$, where it matches the precision of the dense NN ones. As a result, the dense NN workflows are consistently the top performers for this dataset, being as effective as their schema-agnostic sparse NN counterparts.

For D_8, the sparse NN workflows exhibit the highest precision for $\tau = 0.85$ and $\tau = 0.90$, but the blocking ones take the lead for $\tau = 0.95$. This is because the blocking workflows are more flexible in reducing the number of candidates through the large variety of meta-blocking's pruning algorithms and weighting schemes. As a result, their $|C|$ just doubles from $\tau = 0.85$ to $\tau = 0.95$. In contrast, KJ increases the number of candidates it returns per entity of the smallest constituent dataset (Walmart) as the recall threshold increases: from 4 for τ=0.85 to 6 and 17 for τ=0.90 and τ=0.95, respectively.

Finally, in D_9, the dense NN workflows significantly outperform the other two types for $\tau = 0.85$ and $\tau = 0.90$. For $\tau = 0.95$, though, they underperform the sparse NN workflows. The reason is that SCANN and FAISS exceed the two lower recall thresholds by yielding a single candidate ($k = 1$) per entity of the smallest constituent dataset (DBLP). To satisfy the highest τ, though, they raise the number of candidates to three ($k = 3$). In contrast, their sparse NN counterpart, KJ exceeds all recall thresholds with $k = 1$, because it retains all ties in the resulting set of candidates.

To sum up, *the blocking workflows remain the best option for the schema-based settings, too,* consistently exhibiting the lowest distance from the top. On average, their best precision is 10%, 14% and 8% lower than the overall maximum one per dataset for $\tau = 0.85$, $\tau = 0.90$ and $\tau = 0.95$, respectively. This difference is much higher for the sparse and dense NN workflows: 14%/24%/28% and 39%/45%/60%, respectively.

Schema-based vs Schema-agnostic Settings. It is worth comparing the two schema settings in terms of precision and run-time. Theoretically, the lower vocabulary size and character length of the schema-based settings, as indicated in Figures 6, should raise PQ, as the search space is reduced to the values of a single attribute without hampering the detection of duplicates, as it combines high coverage with high distinctiveness. In prac-

tice, though, this is not the case in most datasets. The maximum precision is practically the same in both settings for D_1, D_2 and D_4, regardless of the recall threshold. In D_8, the schema-based precision is consistently lower than the schema-agnostic one by 2/3, whereas in D_9, the precision of the schema-agnostic settings is higher by 5% (for $\tau = 0.85$ and $\tau = 0.90$) to 26% (for $\tau = 0.95$). Only in D_3 do the schema-based settings offer a significant advantage in effectiveness: their PQ is higher by 78%, 88% and 92% (a whole order of magnitude) for τ =0.85, 0.90 and 0.95, respectively.

These patterns suggest that *in most datasets, the schema-agnostic settings provide additional content that reinforces the evidence shared by duplicate entities, without increasing the number of false positives or even by reducing it. Only in datasets with a single highly reliable attribute and many generic ones do the schema-based settings increase precision to a considerable extent.* This is the case with D_3, where "title" provides a concrete description of products, whereas "manufacturer" and "price" are quite generic, increasing the ambiguity of candidate pairs.

6.2 Best workflows per type

Schema-agnostic settings. Among the blocking workflows, we observe that six datasets exhibit the best performance withr the same algorithm across all recall levels. These are D_1, where SA is always the top performer, as well as D_3, D_5, D_6, D_7 and D_{10}, where SB consistently achieves the maximum precision. The rest of the datasets exhibit consistent performance for $\tau = 0.85$ and 0.90, but a different top performer appears for $\tau = 0.95$. In D_2, ESA is replaced by QB, in D_4, TB by EQB and in D_8, SA by TB. The only exception is D_9, where OB is the top performer for the lower recall threshold, but is replaced by TB for the next thresholds. On average, the best performance is achieved by TB, because it achieves the lowest average distance from the maximum precision of this type across all recall thresholds. This distance amounts to 9% for $\tau = 0.85$ and 0.90 and to 14% for $\tau = 0.95$, whereas the distance of the second best algorithm, which differs among the thresholds, is consistently higher than 28%.

Note that all blocking methods are the top performers in at least one dataset. The only exceptions are the baseline workflows, i.e., PW, JW and DQB. PW actually exhibits the lowest precision across all blocking workflows in practically all datasets – it performs a shallow cleaning that achieves very high recall at the cost of a large set of candidates that is dominated by false positives. Surprisingly, JW and DQB outperform OB in at least half the datasets, even though its run-time is consistently higher by orders of magnitude.

It is also worth noting that the default workflows do not surpass the recall threshold in all cases. For $\tau = 0.90$, this is true only for JW over D_3, while for $\tau = 0.95$, this applies to D_2, D_3 and D_6 for both JW and DQB and to D_{10} for DQB. The only other blocking workflows that violate the recall threshold are SA and ESA for $\tau = 0.95$ over D_{10}. The reason is that their maximum block size does not exceed 100 entities, which are insufficient in the context of a large noisy dataset.

Among the sparse NN methods, we observe that KJ consistently achieves the highest precision in most of the datasets, regardless of the recall level. EJ is consistently the top performer in D_1, while for $\tau = 0.85$ and $\tau = 0.90$, it also outperforms KJ in D_7. For the same recall thresholds, the difference between the two methods is minor, i.e., $< 1\%$, over D_4. RB achieves the highest precision in none of the datasets, despite is very high run-tme. We can conclude, therefore, that KJ is the most effective algorithm of this type. Its configuration is rather easy, especially for lower levels of recall, as indicated by the performance of its default version, which yields insufficient recall just once for $\tau = 0.85$ and three times for $\tau = 0.90$; for $\tau = 0.95$, its recall exceeds 0.95 in just four datasets, because of the low cardinality threshold, which remains the same across all recall levels.

Among the dense NN methods, DB is the top performer for D_4, D_7 and D_9 across all τ, because these datasets contain relatively clean long textual values. In fact, the title of the matching publications in D_4 and D_9 is almost the same in both constituent data sources). In all other datasets, RJ achieves the maximum precision by far, while being by far the fastest approach of this type, due to its stochastic operation and the lack of vectorization. As a result, this is the best choice for this type of workflows.

Compared to their sparse NN counterparts, the dense NN workflows are much harder to configure. This is demonstrated by the poor performance of their default workflow, DDB, which uses the same cardinality threshold as DKJ, but does not exceed the recall threshold in half the datasets, regardless of τ. These are D_2, D_3, D_5, D_6 and D_{10} (as well as D_8 for $\tau = 0.95$).

Schema-based settings. The relative performance of the blocking workflows exhibits more diverse patterns than the schema-agnostic settings. No method achieves the highest precision for a specific dataset across all recall thresholds, except for ESA, which dominates D_2. Most methods achieve the top performance at least once. The only exceptions are the baseline workflows PW, JW and

DQB, due to their default configuration, and SA. Together with ESA, these are the only algorithms that violate the recall threshold. This applies to PW over D_2 and D_4 across all thresholds, JW over D_2, D_3 and D_4 across all thresholds, to SA and ESA over D_9 for $\tau = 0.90$ and 0.95 as well as D_8 for $\tau = 0.95$, while the number of DQB's violations increases in proportion to the recall threshold: D_1 for τ=0.85, D_1 and D_2 for τ=0.90 and all datasets but D_4 and D_9 for τ=0.95. These threshold violations along with the four datasets with insufficient coverage in Figure 6(a), suggest that *without fine-tuning, the schema-based settings fall short of recall in half the cases, regardless of the target recall.*

The overall best blocking workflow can be determined as the one with the lowest average distance from maximum precision of this type. For the two lowest recall thresholds, there is a strong competition between TB and QB: the former is the top performer for τ=0.85 with 11.1% vs 16.6%, but the situation is reversed for τ=0.90, where QB takes a narrow lead with 11.7% vs 12.3%. The same algorithms outperform all others for τ=0.95, but QB takes a clear lead with 10.1% vs 23.1%.

The sparse NN workflows exhibit quite stable performance across all recall thresholds. RB never achieves the maximum precision, whereas EJ and KJ are consistently the top performers over D_1,D_4 and D_3,D_4,D_8, respectively. Only for D_9, there is a competition between EJ and KJ, with the former outperforming the latter for τ=0.85, and vice versa for the other two recall thresholds. It is worth noting that DKJ yields insufficient recall only for D_8, regardless of τ, and D_3 for τ=0.95, due to its low cardinality threshold ($k = 5$).

Regarding the dense NN workflows, we observe that the LSH variants consistently generate an excessively large portion of candidate pairs, which are dominated by false positives. As a result, they never achieve the maximum precision in any of the datasets. All other methods are the top performers in at least one case, but the performance of RJ is significantly lower than the schema-agnostic settings. It actually fails to satisfy the recall limit in D_2 and D_4 across all thresholds as well as in D_3 and D_9 for τ=0.95. The reason is that the blocking methods it incorporates discard valuable information that is shared by the duplicate entities. DQB also violates the recall thresholds in may cases, namely in D_2 and D_3 under all settings as well as in D_8 for τ=0.95, due to its low cardinality threshold ($k = 5$).

Among the remaining methods, FS and SN exhibit practically identical performance in most cases. They dominate all other methods over D_2 and D_9 across all recall thresholds. They also excel in D_1, but underperform RJ for τ=0.85 and 0.90. DB also outperforms all other methods in two datasets, D_4 and D_8, regardless

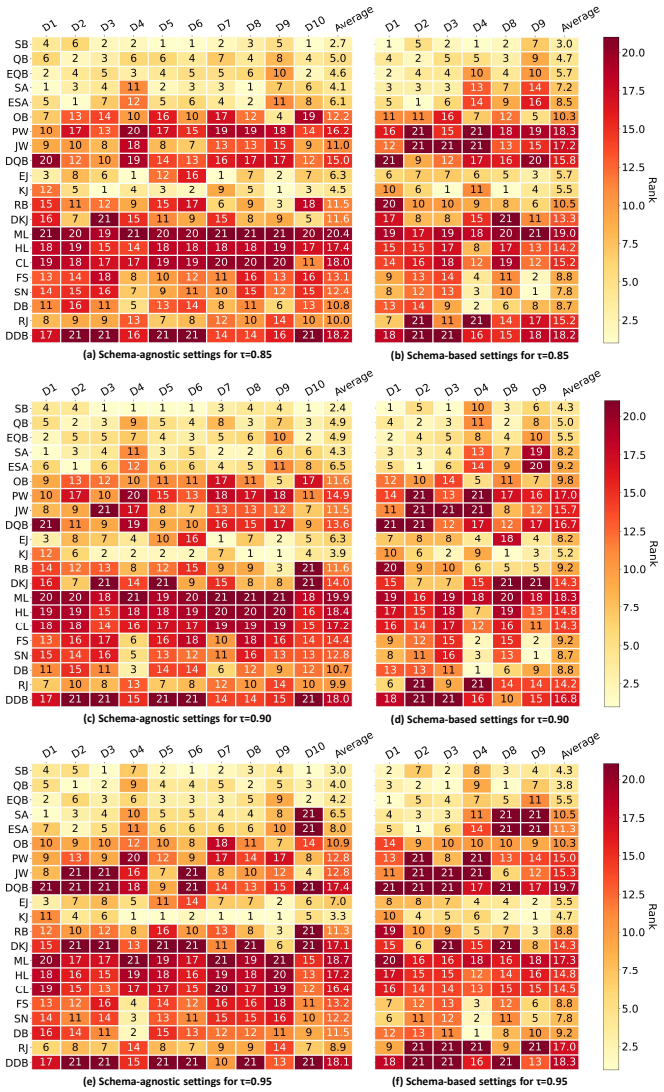

Fig. 10 The ranking positions per workflow, dataset and recall threshold. Lower is better.

of the recall thresholds, while excelling in D_3, where it underperforms RJ for τ=0.90. Among these three methods, the lowest average distance from the maximum precision of this type is achieved by DB for τ=0.85 (19%), FS for τ=0.90 (27%) and SN for τ=0.95 (16%).

6.2.1 Ranking positions

The above patterns about the relative performance of filtering workflows are verified by Figure 10, which demonstrates the ranking position per workflow with respect to precision (PQ) across all schema-settings, datasets and recall thresholds. Note that in every dataset, the methods that fail to satisfy the recall threshold share the last ranking position (21).

The overall best workflow is TB, whose average ranking position is by far the lowest in almost all cases. The only exception pertains to the schema-based settings

with τ =0.95, where it ranks second, with QB having a narrow lead (3.8 vs 4.3). QB generally exhibits low average ranking positions, taking the overall second position in the remaining schema-based settings. It is also the second best blocking workflow and the third overall in the schema-agnostic settings for $\tau = 0.95$, while for τ =0.85 and 0.90, it is only surpassed by SA and EQB (in that order). The worst performance is consistently achieved by the blocking workflows with default configuration, namely PW, JW and DQB (their relative ranking position depends on the recall threshold and the schema settings, but JW is consistently the best among them in schema-agnostic settings). This confirms the significant benefits of parameter fine-tuning.

These patterns suggest that *attribute value tokens offer the best granularity for blocking signatures in the schema-agnostic settings*. Even though some candidates might be missed by typographical errors, they typically share multiple other tokens, due to the schema-agnostic settings. Using substrings of tokens (i.e., q-grams and suffixes) as signatures increases significantly the number of candidate pairs, without any significant benefit in recall. These pairs are significantly reduced by the block and comparison cleaning, but to a lesser extent than those of SB, yielding lower precision.

Among the sparse NN workflows, KJ is consistently the best one, as it ranks third in all cases, except for the schema-agnostic settings with $\tau = 0.90$ and 0.95, where it raises to the second position. The second best workflow of this type is EJ, whose overall ranking position fluctuates between 4 (in the schema-based settings for τ =0.85 and 0.95) and 7 (in the schema-agnostic settings for τ =0.85). This suggests that *the cardinality thresholds are significantly more effective in reducing the search space of ER than the similarity ones*. The reason is that the latter apply a global condition, unlike the former, which operate locally, selecting the best candidates per query entity. Note also that the large distance between KJ and its fixed configuration version, DKJ, in favor of the former, verifies again the benefits of parameter fine-tuning.

Finally, the relative performance of the dense NN workflows is determined by the schema settings. In the schema-agnostic ones, RJ consistently ranks 8^{th} among all workflows, followed by DB, which is always the second best workflow of this type. Their difference for τ =0.85 and 0.90 is low, with DB ranking 9^{th} overall. In the schema-based settings, though, SN takes the lead, ranking 6^{th} overall for τ =0.95 and 7^{th} in the other two cases. DB and FS compete for the second and third place of this type, with the former having the upper hand for the two smaller recall thresholds.

We also observe that among the dense NN methods, the similarity-based ones consistently achieve the lowest by far precision among all fine-tuned techniques. CL typically outperforms the other two LSH variants, i.e., ML and HL, but underperforms the baseline methods PW, JW and DKJ in most of the cases, especially over the largest datasets. The reason is that *the similarity-based methods achieve high recall only by producing an excessively large number of candidate pairs*.

Significantly better performance is achieved by the cardinality-based NN methods. The ranking position of DB is lower than SN and FS on average and in most datasets under the schema-agnostic settings. This means that *the learning-based tuple embedding module raises significantly the precision of NN methods*.

6.2.2 Distances of Duplicates

We now investigate the relative performance of the cardinality-based filtering methods that rely on syntactic representations, i.e., KJ, and those leveraging semantic representations, i.e., FS, SN and DB. To this end, Figure 11 depicts the ranking positions between a query entity and its indexed duplicate across all datasets under the schema-agnostic settings, when using every entity profile from $\mathcal{E}_2$ as a query, whose set of candidates is ranked in decreasing similarity (or increasing distance). For a pair of duplicates $< e_i, e_j >$, if e_j is used as a query, the higher the ranking position of e_i the better is the effectiveness (i.e., higher precision), with $x = 0$ indicating that e_i is ranked first. The horizontal axis corresponds to the ranking position of the matches, while the vertical one corresponds to the number of duplicate pairs per ranking position (note its logarithmic scale). Figure 12 depicts the same diagrams over all datasets with schema-agnostic settings when reversing the role of the input datasets (i.e., indexing $\mathcal{E}_2$ and querying with $\mathcal{E}_1$), while the schema-based settings appear in Figure 13.

For KJ, we used the configuration that performs best on average, across all all datasets, thus lying at the core of DKJ: every entity is represented as multiset of character five-grams, while the query results are ranked in decreasing cosine similarity. For the semantic representations, we use pre-trained 300-dimensional fastText embeddings in combination with Euclidean distance, as computed by the brute-force approach. These settings are representative of all relevant methods dense NN methods, i.e., FS, SN and DB. Note that the resulting diagrams are independent of the recall threshold.

We observe that in the vast majority of cases, the syntactic representations have a higher concentration of duplicates on the top ranking position, which is typically $x = 0$. Seemingly, the syntactic representation

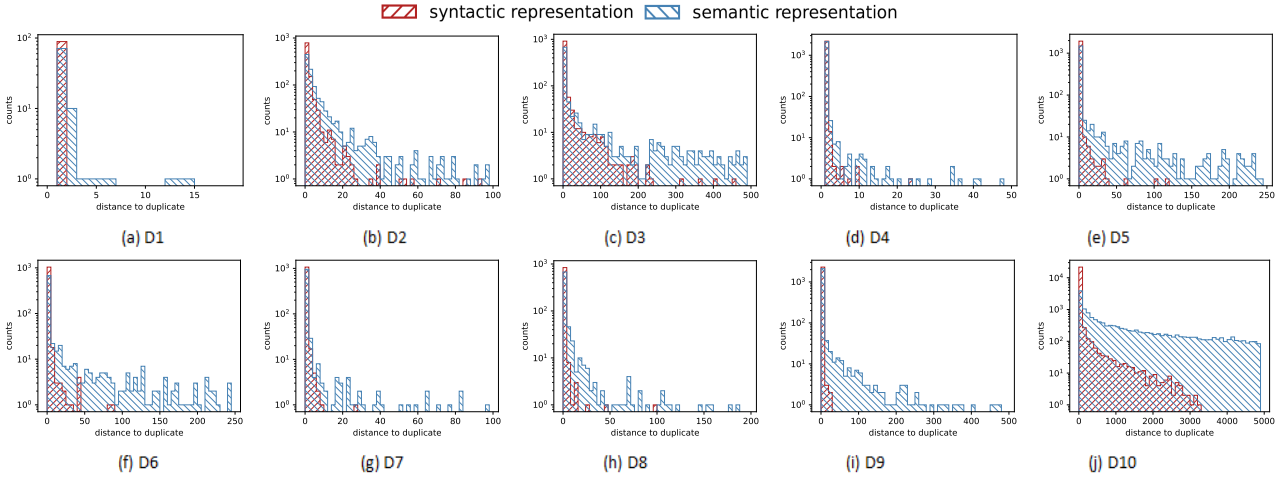

Fig. 11 The distribution of distances between duplicate pairs across all datasets with schema-agnostic settings when indexing $\mathcal{E}_1$ and querying with $\mathcal{E}_2$.

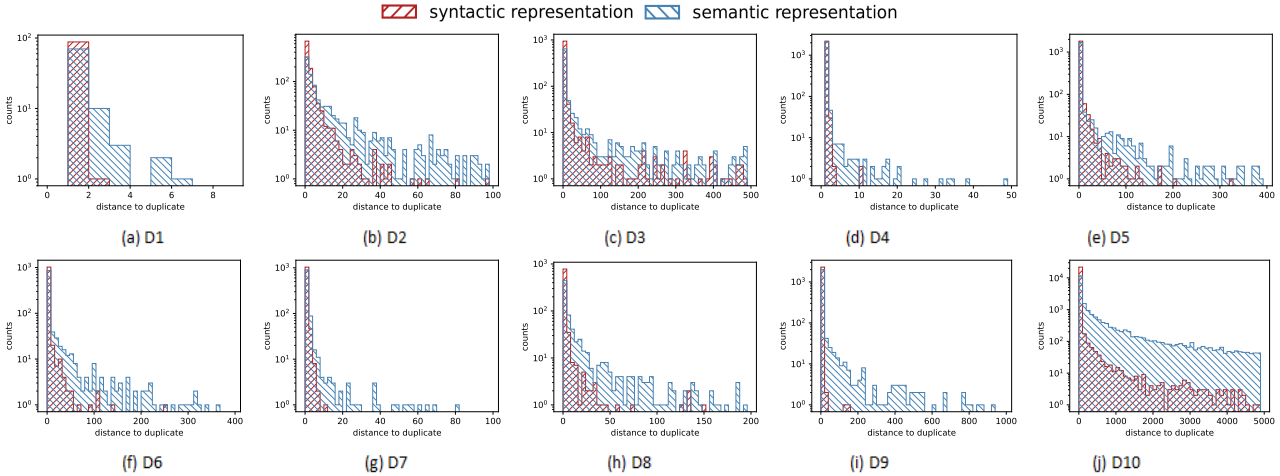

Fig. 12 The distribution of distances between duplicate pairs across all datasets with schema-agnostic settings when reversing the datasets, i.e., when indexing $\mathcal{E}_2$ and querying with $\mathcal{E}_1$.

exceeds the semantic ones to a minor extent, but in practice, their difference is quite large, when considering the logarithmic scale of the vertical axis. As a result, fastText distributes a large number of candidtes to higher ranking positions, thus dominating all bars to the right of the leftmost one (typically $x > 0$). This pattern, which justifies the superiority of KJ over FS, SN and DB, should be attributed to the domain-specific terminology of ER datasets, which is underrepresented in the training corpora of pre-trained embeddings.

There are only a handful of exceptions to this pattern. Under the schema-agnostic settings, KJ exhibits very similar precision with DB, FS and SN in D_{c1}, D_{c4} and D_{c7} for $\tau = 0.85$ and 0.90, because the portion of duplicates that are assigned to $x = 0$ by each workflow suffices for reaching the target recall. The superiority of KJ becomes more clear for $\tau = 0.95$, except for D_{c4}, where all workflows consistently exhibit almost perfect precision. The same applies to the schema-based settings of D_{c1} and D_{c4}.

Overall, we can conclude that *the syntactic representations typically provide more accurate evidence for filtering than semantic ones, due to the domain-specific terminology in ER datasets.*

6.3 Scalability Analysis

We now examine the relative time efficiency of all filtering techniques as the size of the input data increases from 10^4 (D_{10K}) to $2{\cdot}10^6$ (D_{2M}) entities, using the seven synthetic datasets in Table 8. Note that the different programming languages do not allow for comparing them on an equal basis. For this reason, we follow the approach of ANN Benchmark [15], which compares implementations rather than algorithms. This is because even slight changes (e.g., a different data structure) in the implementation of the same algorithm in the same language might lead to significantly different run-times.

Figures 14, 15 and 16 report the experimental results for the blocking, the sparse NN and the dense NN

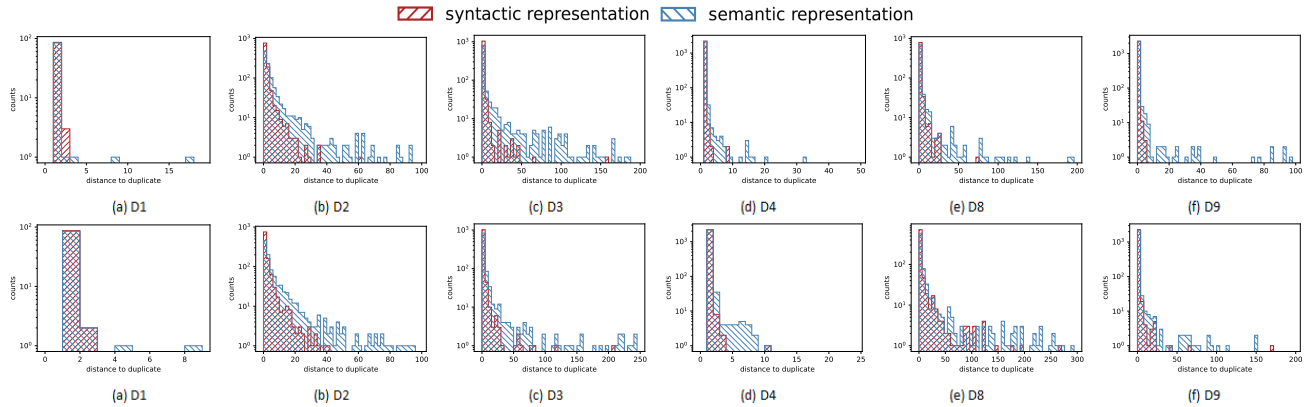

Fig. 13 The distribution of distances between duplicate pairs across all datasets with schema-based settings when indexing $\mathcal{E}_1$ and querying with $\mathcal{E}_2$ (upper line) and when reversing the datasets, i.e., indexing $\mathcal{E}_2$ and querying with $\mathcal{E}_1$ (lower line).

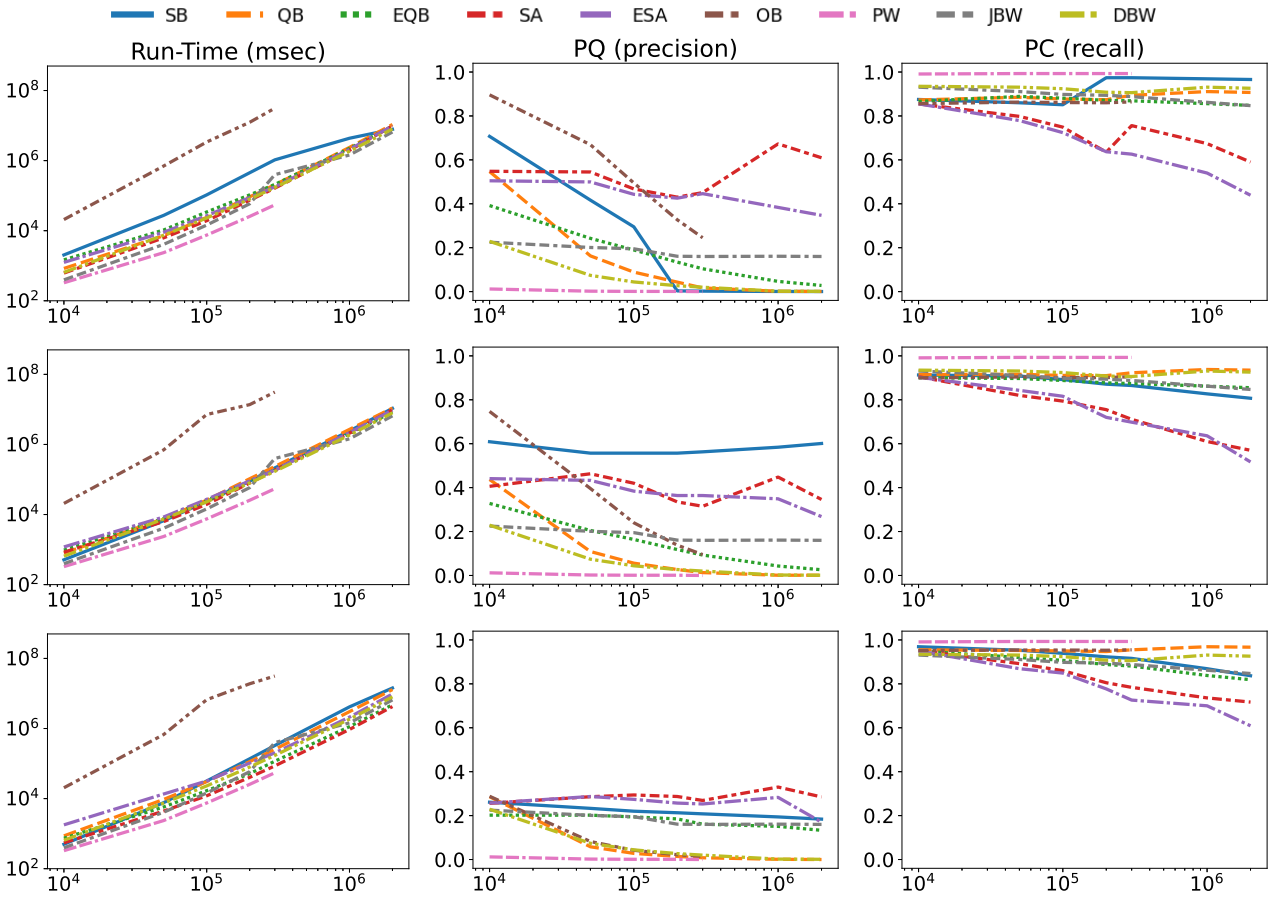

Fig. 14 Scalability analysis of blocking workflows over all datasets in Table 8 for τ=0.85, 0.90 and 0.95 in the first, second and third row, respectively. Logarithmic scale is used in all horizontal axes and the vertical ones in the leftmost diagrams.

workflows, respectively. In every figure, the diagrams in the first row correspond to $\tau = 0.85$, those in the second one to $\tau = 0.90$ and those in the third one to $\tau = 0.95$. Note that the scale of the vertical axis is logarithmic, with the maximum value ($5{\cdot}10^8$ msec) corresponding to 140 hrs. Every filtering technique was fine-tuned on the smallest dataset (D_{10K}) with respect to Problem 1 and the same configuration was applied to all seven datasets. We exclusively consider schema-agnostic settings, due to their robustness to recall.

Starting with the blocking workflows in Figure 14, we observe that PW is the fastest one, due to its simple comparison cleaning, which merely applies Comparison Propagation to eliminate the redundant candidate

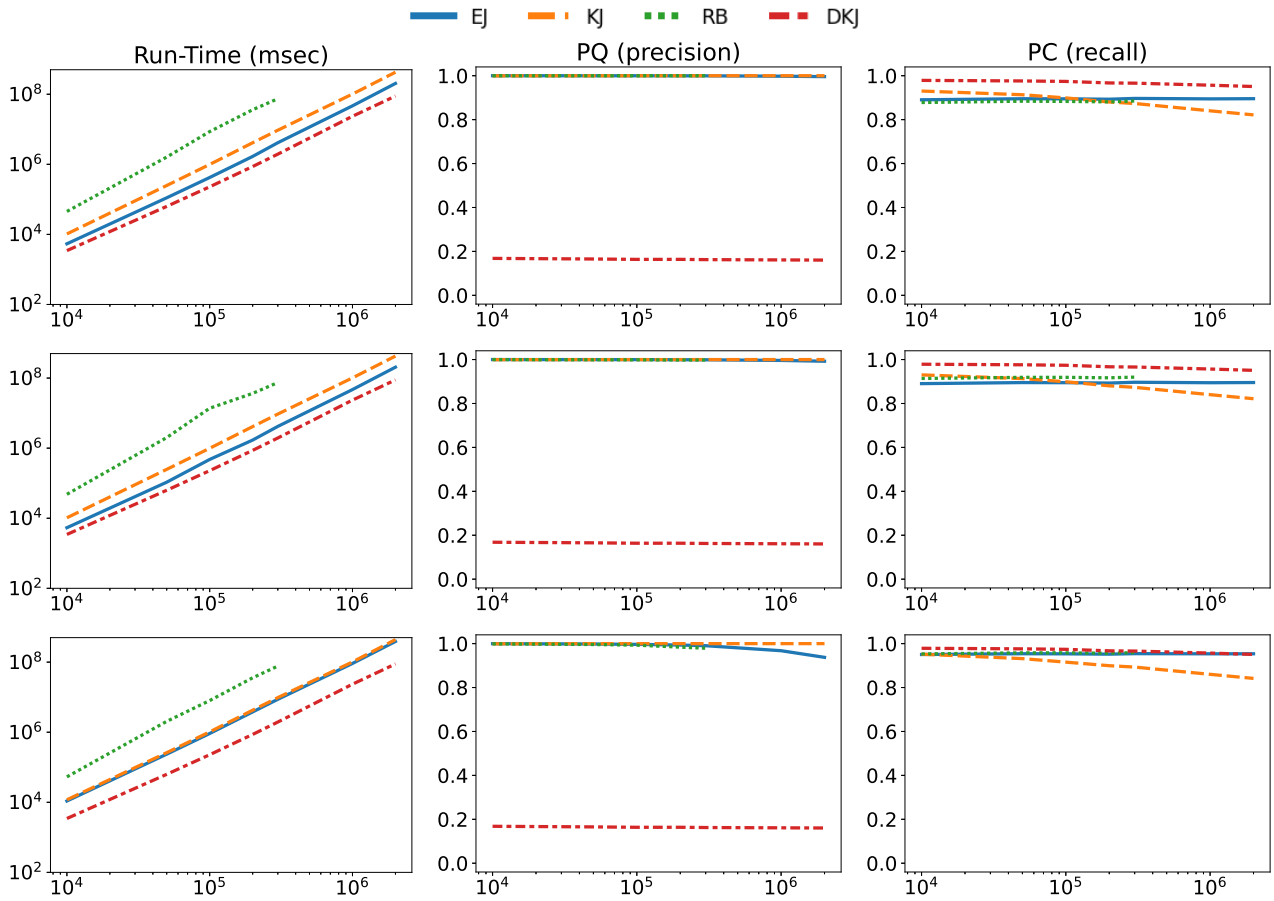

Fig. 15 Scalability analysis of the sparse NN workflows over all datasets in Table 8 for τ=0.85, 0.90 and 0.95 in the first, second and third row, respectively. Logarithmic scale is used in all horizontal axes and the vertical ones in the leftmost diagrams.

pairs. As a result, it processes D_{300K} in less than 1 minute, but it does not scale to the two largest datasets, due to the very large number of candidate pairs it generates. All other workflows are coupled with a Meta-blocking approach that assigns a weight to every candidate pair and prunes the lowest-weighted ones in an effort to reduce the superfluous pairs, too. In this way, they trade higher run-times for higher scalability.

The only exception is OB, whose implementation is rather time-consuming, requiring ∼9 hours to process D_{300K}. Its run-time actually scales super-quadratically, increasing by ∼1,500 times from D_{10K} to D_{300K}. As a result, it cannot process D_{1M} (and D_{2M}) in less than 140 hours. The reason is that its PC consistently satisfies the corresponding recall threshold at the cost of a continuously decreasing precision, i.e., the number of candidate pairs stored in main memory increases superlinearly with increase in the dataset size. This should be attributed to the simplest weighting scheme that is used by OB, namely the number of common blocks (E1).

The differences between the rest of the blocking workflows are minor: they all require between 1.2 and 4 hours to process D_{2M}, regardless of the recall threshold. They all scale superlinearly, but suq-quadratically, increasing their run-time by ∼4·10^3 to ∼29·10^3 times – these extremes correspond to SB for τ=0.85 and 0.95, respectively. It is worth noting that they violate the recall thresholds in most cases in an effort to retain why precision. This is particularly true for SA and ESA, which in all cases, exceed the respective τ only for the smallest dataset. The methods with default configurations, JBW and DBW, satisfy only τ=0.85 across all datasets. The same applies to SB and EQB, while QB constitutes the sole exception, as it consistently exceeds the recall threholds across all datasets under all settings.

Regarding the sparse NN methods, Figure 15 shows that they satisfy the recall thresholds across all datasets. The only exception is KJ, whose PC drops gradually by 11.5% overall in every case. As a result, it violates the recall thresholds from D_{1M} on for $\tau = 0.85$, from D_{200K} on for $\tau = 0.90$ and already from D_{50K} on for $\tau = 0.95$. All methods also exhibit practically perfect precision across all datasets and recall thresholds, which indicates

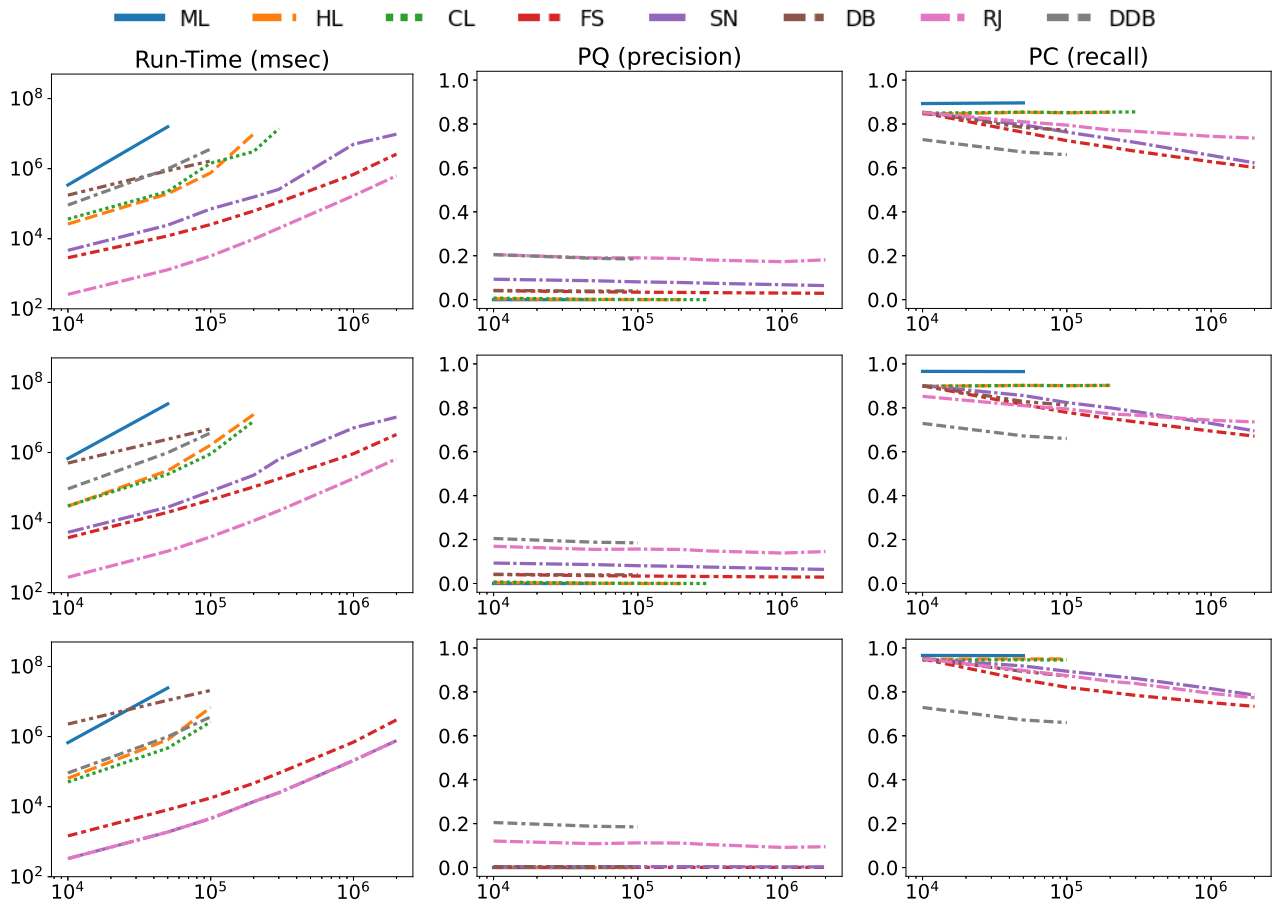

Fig. 16 Scalability analysis of the dense NN workflows over all datasets in Table 8 for τ=0.85, 0.90 and 0.95 in the first, second and third row, respectively. Logarithmic scale is used in all horizontal axes and the vertical ones in the leftmost diagrams.

that they perform the most accurate pruning among all filtering techniques. Only DKJ achieves low precision, with its PQ stable to ∼0.16 in all cases. However, the high effectiveness of these methods comes at the cost of very low time efficiency. KJ and EJ scale quadratically, as their overall run-time increases by ∼40,000 and ∼37,000 times from D_{10K} to D_{2M}, requiring ∼120 and ∼108 hours, respectively, to process the largest dataset (for $\tau = 0.85$ and 0.90, though, EJ requires just ∼56 hours over D_{2M}). The reason is that their best configurations rely on character bigrams under all recall thresholds, thus yielding an excessive number of candidates. In contrast, DKJ is much faster, because of its large q-grams ($q = 5$), which generate few candidates per entity: it processes D_{2M} within <25 hours, scaling subquadratically, i.e., its run-time increases by ∼26,000 times from D_{10K} to D_{2M}. The slowest workflow, by far, is RB, which scales only up to D_{300K}, where it requires ∼20 hours. On average, it is 8 times slower than KJ, thus exceeding our run-time limit for D_{1M} and D_{2M}.

Among the dense NN methods, Figure 16 shows that ML scales up to D_{50K}, while CL and HL scale up to D_{300K}, D_{200K} and D_{100K} for τ=0.85, 0.90 and 0.95, respectively. The reason is that their large number of candidates does not fit into the available main memory. Similarly, DB and DDB scale up to D_{100K}, because of their quadratic space complexity (i.e., they requires ∼300GB of RAM to create a 200K×200K float array for D_{200K}). FS and SN exhibit much higher scalability, due to the approximate indexes they employ (IVF and AH, respectively). They both process D_{2M} within a couple of hours, regardless of the recall threshold. However, the fastest by far approach, not only in this category but in general, is RJ, which processes D_{2M} in just 13 minutes under all recall settings. This should be attributed to its stochastic functionality, which is able to detect highly likely matches without time consuming weighting schemes.

Regarding effectiveness, we observe that the LSH variants are the only techniques of this category that satisfy the recall threshold across all datasets they can process. This stems from the excessively large number of candidate pairs they generate, which prevents them from scaling to the largest datasets. The remain-

ing techniques satisfy the recall threshold only in the smallest dataset. In fact, they all combine very low precision with a steadily decreasing recall. The most effective approach is RJ for τ=0.85 and 0.90: its recall is consistently higher than FS, SN and DB across all datasets, dropping to 0.75 and 0.76, respectively, while its precision is higher than the second best approach, SN, by 2-3 times for τ=0.85 and by 6-7 times for τ=0.90. For τ=0.95, the recall of RJ is lower by ∼2.4%, on average, than SN, which consistently achieves the highest PC. Yet, RJ's precision is at least 27 times higher than SN across all datasets. This means that RJ dominates all other workflows in this category in all respects, i.e., both with respect to time efficiency and effectiveness.

Overall, we can conclude that the blocking workflows leveraging Meta-blocking are quite fast, but quite unstable, requiring fine-tuning per dataset and recall threshold. The only exception is QB, which remains maintains high recall at the cost of very low precision. The best dense NN approach is RJ, but its very high efficiency comes at the cost of an effectiveness that is inversely proportional to size of the input data and the recall threshold. The sparse NN methods EJ and KJ exhibit the highest robustness and effectiveness at the cost of the highest run-time. Therefore, techniques for enhancing the scalability of these two sparse NN methods are required.

6.4 Run-time Analysis

Not only they consistently achieve the lowest run-time, but , at the cost of significantly higher run-time. In fact, the blocking workflows offer actually the fastest functionality under all circumstances

We now examine the breakdown of the run-time for every filtering method. The breakdown for all filtering methods, including the baseline ones, appears in Figure 19 for the schema-agnostic settings of the datasets D_5-D_7 and D_{10}, which lack the schema-based settings. For the remaining datasets, the breakdown of the schema-agnostic and the schema-based settings is presented in Figures 17 and 18, respectively. For the cardinality-based, the indexing and querying time corresponds to the best configuration (see RVS in Table ?? for kNNJ and Table ?? for FAISS, SCANN and DeepBlocker). Note that every line corresponds to a different dataset, with a different diagram per type of filtering methods, but these diagrams share the same scale so that the comparison between all methods is straightforward. For this reason, the same scale per dataset is used by both schema-agnostic and schema-based settings in Figures 17 and 18.

Starting with the blocking workflows, their overall run-time consists of the run-times of the four steps in Figure 1:

1. the block building time (t_b),
2. the block purging time (t_p),
3. the block filtering time (t_f), and
4. the comparison cleaning time (t_c).

We observe that the block cleaning methods are quite fast, due to their coarse-grained functionality, corresponding to a tiny portion of the overall run-time. On average, the block purging time accounts for less than 0.9% for all methods for both schema-agnostic and schema-based settings. The only exception is the fastest blocking workflow, PBW, where its contribution to RT raises to 3.2%, on average. Block Filtering involves a more fine-grained functionality that operates at the level of individual entities. As a result, it accounts for a larger portion of RT, which however remains lower than 2% in all cases. The only exceptions are the schema-based settings of SBW and DQBW, where its cost raises to 17.9% and 3.8%, respectively, due to the absence of Block Purging (i.e., Block Filtering processes all the initial blocks).

Regarding the relative cost of block building and comparison cleaning, we distinguish the blocking workflows in two main groups: one with the methods performing fast signature extraction (i.e., SBW, QBW and DQBW), and one with the methods incorporating an elaborate signature extraction (i.e., EQBW and ESABW). For the former, the cost of comparison cleaning outweighs the cost of block building: on average, it accounts for 68.8% and 58.9% of the overall RT for SBW and QBW/DQBW, respectively. In contrast, the cost of block building is much higher than comparison cleaning for EQBW and ESABW, accounting for 65.2% and 58.3%, respectively, on average. This should be attributed not only to the more complex process of signature extraction, but also to the lower number of candidate pairs in the resulting set of blocks. To this category also belongs SBW, whose block building takes up 76.4% of the overall run-time. However, this should be attributed to the very low cost of its comparison cleaning, which simply applies Comparison Propagation (i.e., it discards redundant candidate pairs, without assigning scores to remove superfluous ones, too). In the middle of these two categories lies SABW, where the cost of block building and comparison cleaning similar (46.6% and 53.4%, respectively, on average). The reason is that its block building phase is relatively complex, but produces a very small set of candidate pairs. As a result, it is often one of the fastest blocking workflows.

The overall run-time for the rest of the filtering methods is divided into:

1. the pre-processing time (t_r), which includes the cost of stop-word removal, stemming and the transformation of attribute values into embedding vectors (if applicable).
2. the indexing time (t_i), and
3. the querying time (t_q).

As regards the sparse NN methods, we observe that the cost of indexing is consistently the lowest one among the three steps. On average, across all datasets and schema settings, t_i accounts for 3.6%, 5.6% and 9.9% of the overall run-time of ε-, kNN- and DkNNJ, respectively. The higher portion for DkNN stems from the complex representation it uses, i.e., the multiset of five-grams. The second most time-consuming step is pre-processing, which on average, accounts for 31.4%, 22.7% and 52.18% of the total RT for ε-, kNN- and DkNN, respectively. The discrepancy between the first two methods and the last one is caused by the default configurations of the baseline method, which always includes cleaning. In contrast, the fine-tuned methods apply cleaning in 2/3 and 1/2 of the cases, respectively (see Table **??**). The contribution of t_r to RT would be much higher for both methods, if they consistently applied cleaning. Note also that the breakdown of DkNN demonstrates that stop-word removal and stemming have a high computational cost. Finally, the rest of RT corresponds to the querying phase, which occupies 65.0%, 72.3% and 37.9% of the overall run-time, on average, for ε-, kNN- and DkNN, respectively. Note that the portion of t_q is higher for kNNJ than for ε-Join, because its querying phase is more costly, due to the sorting of candidate pairs. This portion fluctuates considerably for both methods (σ =0.299 in both cases), as it depends on the threshold they use. The variance is significantly reduced for DkNN (σ =0.186), because it uses the same cardinality-threshold in all cases. The low value of this threshold is another reason for the dominance of the pre-processing time in the case of DkNN.

Finally, for the dense NN methods we observe that the pre-processing time dominates their run-time to a significant extent. The reason is that this step now includes the cost of creating the semantic representations of entities based on the pre-trained fastText embeddings. Its portion ranges from 66% (SCANN) to 86.4% (FAISS) and 91.0% (DeepBlocker and DDB). The portion is lower for SCANN, due to the time-consuming partitioning it performs in both the indexing and the query phase. In contrast, it is higher for FAISS, due to the very low cost of the other two phases. For the same reason, t_r accounts for a larger portion of RT in the schema-based settings, where both indexing and querying are much faster. However, in the case of DeepBlocker and DDB, the high portion of t_r should be attributed to the Autoencoder, which learns the tuple embedding module, i.e., it transforms values into fastText embeddings, it creates an artificial dataset and then learns a neural model. As expected, the indexing phase is the fastest step for all methods, accounting for <0.5% (FAISS, DeepBlocker and DDB) to 11.9% (SCANN) of the overall run-time. The rest of RT corresponds to the querying phase, whose portion ranges from 9.0% (DeepBlocker and DDB) to 22.7% (CP-LSH). Note that the only exception to these patterns is MH-LSH, which does not involve the cost of semantic representations. As a result, its t_r is reduced to 14.9% of RT, with the querying phase dominating its run-time (69.7%), due to the very large number of candidate pairs it generates.

6.5 Configuration Analysis

In this section, we present the detailed configuration of every filtering method that corresponds to its performance in Table **??**.

Table **??** reports the configuration of blocking workflows across all datasets and schema settings. BP stands for the use of Block Purging or not (it is a parameter-free approach), BFr indicates the filtering ratio used by Block Filtering, PA denotes the pruning algorithm that is used by the Meta-blocking step and WS stands for the corresponding weighting scheme. Recall that the the domain of each parameter appears in Table 4.

The configuration of sparse NN methods is listed in Table **??**. CL stands for the use of pre-processing for cleaning an attribute value, SM for the similarity measure, RM for the representation model that is used, t for the similarity threshold and K for the number of nearest neighbors, while RVS indicates whether $\mathcal{E}_2$ is indexed and $\mathcal{E}_1$ is used as the query set, instead of the opposite. The domain of each parameter appears in Table 5.

Finally, the configuration of the dense NN methods is reported in Table **??**. Note that CL denotes the use of pre-processing for cleaning an attribute value, t the corresponding Jaccard similarity threshold, k the size of the k-shingles that are used as representation model and RVS whether the indexed and the query dataset should be reversed or not. The domain of each parameter appears in Table 6.

Based on the fine-tuning experiments that were performed over D_{c1}-D_{c10}, we can draw several useful conclusions regarding the configuration of the filtering techniques.

For the blocking workflows, the following rules of thumb can be used when fine-tuning them:

• Block Purging should never be used in schema-based settings. Even in schema-agnostic settings, it should be avoided, due to its aggresive pruning.

• Block Filtering offers a more suitable alternative for coarse-grained block cleaning. It should always be part of a blocking workflow, albeit in combination with a relative high ratio (i.e., $BFr \geq 0.5$ in most cases).

• Among the comparison cleaning techniques, RCNP and BLAST constitute the best choice in most cases. Typically, they are combined with the weighting schemes χ^2 and ARCS.

• For Q-Grams Blocking, large values for q should be used regardless of the schema settings, i.e., $q = 6$ in most cases. The same applies to Extended Q-Grams Blocking, except for the schema-based settings, where q should be set to 3. At the same time, the threshold parameter should be set to $t = 0.9$.

• Similarly, large values for l_{min} should be typically used for Suffix Arrays Blocking and Extended Suffix Arrays Blocking. For both algorithms, the maximum block size b_{max} should be set in proportion to the number of input entities.

For sparse NN methods, the following guidelines should be used when fine-tuning them:

• The cosine similarity is the best measure in the vast majority of cases for both ϵ- and kNN-Joins.

• Stop word removal and stemming should be typically applied, at least as a means of reducing the search space.

• In the case of kNN-Join, the largest dataset should be indexed, and the smallest one should be used for querying. In this way, very small values for k (i.e., $k < 5$) suffice for achieving high recall and precision.

• For ϵ-Joins, a low similarity threshold should be used in the case of schema-agnostic settings (0.35 on average) and a higher one for schema-based settings (0.55).

• Both joins should be combined with character n-grams for tokenization. The size of n should be 2 or 3 for schema-based settings and 4 or 5 for schema-agnostic ones. In the latter case, multi-sets should be used instead of bags.

For dense NN methods, the following advice could be used when fine-tuning them:

• Stop word removal and stemming should be typically applied to the input data.

• For MinHash LSH, the size of k-shingles should be set to 2, while the number of bands and rows should be set to 32 and 16, respectively.

• For Cross-polytope and Hyperplane LSH, the number of tables and hashes should be set in proportion to the size of the input data. The number of probes can be automatically configured in order to achieve the desired level of recall.

• For FAISS, SCANN and DeepBlocker, the number of candidates can be reduced by indexing the largest dataset and querying it with the smallest one. The number of candidates per query entity, k, should be proportional to the number of input entities, with the schema-based settings calling for smaller values than the schema-agnostic ones.

• FAISS should be combined with a Flat index, while its embedding vectors should always be normalized and combined with the Euclidean distance.

• Similarly, SCANN should be combined with a brute-force index over Euclidean distances.

• DeepBlocker should be coupled with AutoEncoder, which lowers the memory requirements and accelerates its run-time.

7 Conclusions

Our experimental results lead to the following conclusions:

1) Fine-tuning vs default parameters. For all types of methods, optimizing the internal parameters with respect to a performance goal significantly raises the performance of filtering. This problem is poorly addressed in the literature [9], and the few proposed tuning methods require the involvement of experts [78,79]. *More emphasis should be placed on a-priori fine-tuning the filtering methods through an automatic, data-driven approach that requires no labelled set.*

2) Schema-based vs schema-agnostic settings. The former significantly improve the time efficiency at the cost of unstable effectiveness, while the latter offer robust effectiveness, as they inherently address heterogeneous schemata as well as misplaced and missing values that are common in ER [11,67]. Even when the schema-based settings exhibit high recall, their maximum precision outperforms the schema-agnostic settings in just three cases (D_1–D_3). The schema-agnostic settings also exceed the target recall even in combination with default configurations (the baseline blocking workflows), unlike the schema-based settings, where all baseline methods fall short of the target recall at least once. For these reasons, *the schema-agnostic settings are preferable over the schema-based ones.*

3) Similarity vs cardinality thresholds. Poor performance is typically achieved by all similarity-based NN methods (cf. Table 3). The LSH variants achieve high recall only by producing an excessively large number of candidates: MH-, CP- and HP-LSH reduce the candidate pairs of the brute-force approach by 48%, 91% and 89%, respectively, on average, across all datasets

in Table 7. This might seem high (a whole order of magnitude for CP- and HP-LSH), but is consistently inferior to the cardinality-based NN methods. This applies even to ε-Join, which is the best similarity-based approach, reducing the candidate pairs of the brute-force approach by 99% (i.e., multiple orders of magnitude): it underperforms kNN-Join in 9 out of 16 cases. Most importantly, the number of candidates produced by similarity-based methods depends quadratically on the total size of the input. For the cardinality-based methods, it depends linearly on the size of the query dataset, which is usually the smallest one, i.e., $|C| = k \cdot min(|E_1|, |E_2|)$; in almost all cases, $k \ll 100$ for all cardinality-based methods, especially kNN-Join. Therefore, *cardinality thresholds are preferable over similarity thresholds.*

4) Syntactic vs semantic representations. The blocking workflows and the sparse NN methods assume that the pairs of duplicates share textual content; the rarer this content is, the more likely are two entities to be matching. In contrast, most dense NN methods assume that the duplicate entities share syntactically different, but semantically similar content that can be captured by pre-trained character-level embeddings. The latter assumption is true for the matching step of ER [11], but our experiments advocate that filtering violates this assumption: semantic-based representations outperform the syntactic ones only in two cases (D_{b4} and D_{b9}). Comparing kNN-Join with cardinality-based dense NN methods, we observe that the former consistently uses a lower threshold (see Tables **??** and **??**). This means that the semantic representations introduce more false positives than the syntactic ones, due to the out-of-vocabulary, domain-specific terms in ER datasets – see Section B in the Appendix for an in depth analysis. Hence, the *syntactic representations are preferable over semantic ones.*

5) Most effective filtering method. The only method that combines a cardinality threshold with a syntactic representation is kNN-Join. Although the Standard Blocking workflow (SBW) performs better in the schema-agnostic settings, kNN-Join offers two qualitative advantages: (i) Unlike SBW, the number of candidates is linear in the input size. (ii) kNN-Join is easy to configure as shown by the high performance of the DkNN-Join baseline: even though its recall fluctuates in $[0.8, 0.9]$ for three datasets, it outperforms PBW, the default configuration of SBW, in almost all other cases.

6) Most scalable filtering method. Among the considered techniques, only the blocking workflows, FAISS and SCANN scale to all synthetic, Dirty ER datasets within a reasonable time (<4 hrs). As the number of input entities increases from 10^4 (D_{10K}) to $2{\cdot}10^6$ (D_{2M}), the run-times of all techniques scales superlinearly (>200 times), but subquadratically (<40,000 times). For the blocking workflows, the increase actually ranges from ∼8,000 times (EQBW) to ∼20,000 times (SBW). This increase is just ∼1,600 times for SCANN and ∼700 times for FAISS. As a result, FAISS is by far the fastest and most scalable filtering technique when processing large datasets, due to its approximate indexing scheme, leaving SCANN in the second place.

In the future, we will enrich the *Continuous Benchmark of Filtering methods for ER* with new datasets and will update the rankings per dataset with new filtering methods. We will also explore filtering techniques that consider not only textual information but also geographic, numeric etc.

References

1. L. Getoor and A. Machanavajjhala, "Entity resolution: Theory, practice & open challenges," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 2018–2019, 2012.
2. X. L. Dong and D. Srivastava, *Big Data Integration*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.
3. P. Christen, *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*, ser. Data-Centric Systems and Applications. Springer, 2012.
4. A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 1–16, 2007.
5. V. Christophides, V. Efthymiou, and K. Stefanidis, *Entity Resolution in the Web of Data*, ser. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2015.
6. G. Papadakis, E. Ioannou, E. Thanos, and T. Palpanas, *The Four Generations of Entity Resolution*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2021.
7. N. Barlaug and J. A. Gulla, "Neural networks for entity matching: A survey," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 3, pp. 52:1–52:37, 2021.
8. O. Hassanzadeh, F. Chiang, R. J. Miller, and H. C. Lee, "Framework for evaluating clustering algorithms in duplicate detection," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 1282–1293, 2009.
9. G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas, "Blocking and filtering techniques for entity resolution: A survey," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 31:1–31:42, 2020.
10. P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 9, pp. 1537–1555, 2012.
11. S. Thirumuruganathan, H. Li, N. Tang, M. Ouzzani, Y. Govind, D. Paulsen, G. Fung, and A. Doan, "Deep learning for blocking in entity matching: A design space exploration," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2459–2472, 2021.
12. G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas, "Comparative analysis of approximate blocking techniques for entity resolution," *Proc. VLDB Endow.*, vol. 9, no. 9, pp. 684–695, 2016.

13. W. Mann, N. Augsten, and P. Bouros, "An empirical evaluation of set similarity join techniques," *Proc. VLDB Endow.*, vol. 9, no. 9, pp. 636–647, 2016.
14. Y. Jiang, G. Li, J. Feng, and W. Li, "String similarity joins: An experimental evaluation," *Proc. VLDB Endow.*, vol. 7, no. 8, pp. 625–636, 2014.
15. M. Aumüller, E. Bernhardsson, and A. J. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Inf. Syst.*, vol. 87, 2020.
16. G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika, "Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data," *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 312–323, 2015.
17. F. Fier, N. Augsten, P. Bouros, U. Leser, and J. Freytag, "Set similarity joins on mapreduce: An experimental survey," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1110–1122, 2018.
18. G. Papadakis, M. Fisichella, F. Schoger, G. Mandilaras, N. Augsten, and W. Nejdl, "How to reduce the search space of entity resolution: with blocking or nearest neighbor search?" *CoRR (to appear in ICDE 2023)*, vol. abs/2202.12521, 2022.
19. G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis, "Domain- and structure-agnostic end-to-end entity resolution with jedai," *SIGMOD Rec.*, vol. 48, no. 4, pp. 30–36, 2019.
20. G. Papadakis, G. M. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis, "Three-dimensional entity resolution with jedai," *Inf. Syst.*, vol. 93, p. 101565, 2020.
21. M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios, "TAILOR: A record linkage tool box," in *ICDE*, 2002, pp. 17–28.
22. S. Galhotra, D. Firmani, B. Saha, and D. Srivastava, "BEER: blocking for effective entity resolution," in *SIGMOD*, 2021, pp. 2711–2715.
23. ——, "Efficient and effective ER with progressive blocking," *VLDB J.*, vol. 30, no. 4, pp. 537–557, 2021.
24. C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.
25. P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra, "Magellan: Toward building entity matching management systems," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1197–1208, 2016.
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.*, vol. 25, no. 12, pp. 2665–2682, 2013.
27. G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl, "Meta-blocking: Taking entity resolution to the next level," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 8, pp. 1946–1960, 2014.
28. L. Gagliardelli, G. Papadakis, G. Simonini, S. Bergamaschi, and T. Palpanas, "Generalized supervised meta-blocking," *Proc. VLDB Endow.*, vol. 15, no. 9, pp. 1902–1910, 2022.
29. G. Simonini, S. Bergamaschi, and H. V. Jagadish, "BLAST: a loosely schema-aware meta-blocking approach for entity resolution," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1173–1184, 2016.
30. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001, pp. 491–500.
31. N. Augsten and M. H. Böhlen, *Similarity Joins in Relational Database Systems*. Morgan & Claypool Publishers, 2013.
32. N. Augsten, "A roadmap towards declarative similarity queries," in *EDBT*, 2018, pp. 509–512.
33. Y. N. Silva, W. G. Aref, P. Larson, S. Pearson, and M. H. Ali, "Similarity queries: their conceptual evaluation, transformations, and processing," *VLDB J.*, vol. 22, no. 3, pp. 395–420, 2013.
34. R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007, pp. 131–140.
35. S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006, p. 5.
36. P. Bouros, S. Ge, and N. Mamoulis, "Spatio-textual similarity joins," *Proc. VLDB Endow.*, vol. 6, no. 1, pp. 1–12, 2012.
37. D. Deng, G. Li, H. Wen, and J. Feng, "An efficient partition based method for exact set similarity joins," *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 360–371, 2015.
38. D. Deng, Y. Tao, and G. Li, "Overlap set similarity joins with theoretical guarantees," in *SIGMOD*, 2018, pp. 905–920.
39. E. Zhu, D. Deng, F. Nargesian, and R. J. Miller, "JOSIE: overlap set similarity search for finding joinable tables in data lakes," in *SIGMOD*, 2019, pp. 847–864.
40. C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient similarity joins for near-duplicate detection," *ACM Trans. Database Syst.*, vol. 36, no. 3, pp. 15:1–15:41, 2011.
41. C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.
42. D. Kocher and N. Augsten, "A scalable index for top-k subtree similarity queries," in *SIGMOD*, 2019, pp. 1624–1641.
43. G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.
44. M. Yu, G. Li, D. Deng, and J. Feng, "String similarity search and join: a survey," *Frontiers Comput. Sci.*, vol. 10, no. 3, pp. 399–417, 2016.
45. G. Li, D. Deng, J. Wang, and J. Feng, "PASS-JOIN: A partition-based method for similarity joins," *Proc. VLDB Endow.*, vol. 5, no. 3, pp. 253–264, 2011.
46. P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998, p. 604–613.
47. M. Fisichella, A. Ceroni, F. Deng, and W. Nejdl, "Predicting pair similarities for near-duplicate detection in high dimensional spaces," in *DEXA*, 2014, pp. 59–73.
48. D. Karapiperis, D. Vatsalan, V. S. Verykios, and P. Christen, "Efficient record linkage using a compact hamming space," in *EDBT*, 2016, pp. 209–220.
49. H. Kim and D. Lee, "HARRA: fast iterative hashed record linkage for large-scale data collections," in *EDBT*, 2010, pp. 525–536.
50. W. Zhang, H. Wei, B. Sisman, X. L. Dong, C. Faloutsos, and D. Page, "Autoblock: A hands-off blocking framework for entity matching," in *WSDM*, 2020, pp. 744–752.
51. M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang, "Distributed representations of tuples for entity resolution," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1454–1467, 2018.
52. A. Z. Broder, "On the resemblance and containment of documents," in *SEQUENCES*, 1997, pp. 21–29.

53. J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive data sets*. Cambridge university press, 2020.
54. M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002, pp. 380–388.
55. B. N. et al., "Multiprobe-lsh," https://github.com/gopalmenon/Multi-Probe-LSH, 2018.
56. J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Trans. Big Data*, vol. 7, no. 3, pp. 535–547, 2021.
57. R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *ICML*, 2020, pp. 3887–3896.
58. C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *ICDE*, 2009, pp. 916–927.
59. Z. Yang, B. Zheng, G. Li, X. Zhao, X. Zhou, and C. S. Jensen, "Adaptive top-k overlap set similarity joins," in *ICDE*, 2020, pp. 1081–1092.
60. P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, 2017.
61. S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra, "Deep learning for entity matching: A design space exploration," in *SIGMOD*, 2018, pp. 19–34.
62. C. Fu, X. Han, J. He, and L. Sun, "Hierarchical matching network for heterogeneous entity resolution," in *IJCAI*, 2020, pp. 3665–3671.
63. Z. Yao, C. Li, T. Dong, X. Lv, J. Yu, L. Hou, J. Li, Y. Zhang, and Z. Dai, "Interpretable and low-resource entity matching via decoupling feature learning from decision making," in *ACL/IJCNLP*, 2021, pp. 2770–2781.
64. D. Zhang, Y. Nie, S. Wu, Y. Shen, and K. Tan, "Multi-context attention for entity matching," in *WWW*, 2020, pp. 2634–2640.
65. FALCONN-LIB, "Fine-tuning Multi-proble LSH," https://github.com/FALCONN-LIB/FALCONN/blob/master/src/examples/glove/glove.py, 2017, [Online; accessed April 25, 2022].
66. A. Andoni, P. Indyk, T. Laarhoven, I. P. Razenshteyn, and L. Schmidt, "Practical and optimal LSH for angular distance," in *NIPS*, 2015, pp. 1225–1233, available at https://github.com/FALCONN-LIB/FALCONN.
67. S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra, "Deep learning for entity matching: A design space exploration," in *SIGMOD*. ACM, 2018, pp. 19–34.
68. H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 484–493, 2010.
69. J. Euzenat, A. Ferrara, C. Meilicke, J. Pane, F. Scharffe, P. Shvaiko, H. Stuckenschmidt, O. Sváb-Zamazal, V. Svátek, and C. T. dos Santos, "Results of the ontology alignment evaluation initiative 2010," in *Proceedings of the 5th International Workshop on Ontology Matching (OM-2010), Shanghai, China, November 7, 2010*, vol. 689, 2010.
70. D. Obraczka, J. Schuchart, and E. Rahm, "Embedding-assisted entity resolution for knowledge graphs," in *ESWC*, vol. 2873, 2021.
71. B. Kenig and A. Gal, "Mfiblocks: An effective blocking algorithm for entity resolution," *Inf. Syst.*, vol. 38, no. 6, pp. 908–926, 2013.
72. P. Christen, "Febrl -: an open source data cleaning, deduplication and record linkage system with a graphical user interface," in *KDD*, 2008, pp. 1065–1068.
73. JedAI, "Java gEneric DAta Integration (JedAI) Toolkit," https://github.com/scify/JedAIToolkit, 2017, [Online; accessed April 25, 2022].
74. F. Research, "Faiss: a library for efficient similarity search," https://github.com/facebookresearch/faiss.
75. G. Research, "Scann," https://github.com/google-research/google-research/tree/master/scann.
76. QCRI, "Deepblocker," https://github.com/qcri/DeepBlocker.
77. S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.
78. H. Li, P. Konda, P. S. G. C., A. Doan, B. Snyder, Y. Park, G. Krishnan, R. Deep, and V. Raghavendra, "Matchcatcher: A debugger for blocking in entity matching," in *EDBT*, 2018, pp. 193–204.
79. R. Maskat, N. W. Paton, and S. M. Embury, "Pay-as-you-go configuration of entity resolution," *Trans. Large Scale Data Knowl. Centered Syst.*, vol. 29, pp. 40–65, 2016.

Appendix

A. Detailed Performance

Table **??** presents the detailed performance of all filtering techniques over all datasets in Table 7 under schema-agnostic and schema-based settings. Table **??**(a) reports the recall (PC), Table **??**(b) the precision (PQ), Table **??**(c) the overall run-time (RT) and Table **??**(d) the actual number of candidate pairs they generate per dataset. In Tables **??**(b) and (d), the best performance per type of algorithms, dataset and schema settings is underlined, the overall best performance is highlighted in bold, while the cases corresponding to insufficient recall are marked in red.

We notice that the precision of all methods is highly correlated. In fact, the Pearson correlation coefficient between any pair of methods exceeds 0.5. This means that the performance of filtering depends heavily on the dataset characteristics. For example, datasets like D_{a3} yield very low precision for all methods, because their duplicate entities share only generic/noisy content that appears in many non-matching profiles, too (e.g., stop words). In contrast, datasets like D_{a4} entail duplicates with very distinguishing content in common, yielding an almost perfect performance in most cases.

Regarding time efficiency, we observe that the blocking workflows excel in run-time. Most of them require less than a second to process all datasets up to D_{a8}, few seconds for D_{a9} and less than a minute for D_{a10}. In most cases, the fastest workflow is PBW, due to its simple comparison cleaning, which merely applies Comparison Propagation to eliminate the redundant candidate pairs. All other workflows are always coupled with a Meta-blocking approach that assigns a weight to every candidate pair and prunes the lowest-weighted ones in an effort to reduce the superfluous pairs, too (see the configurations of Table **??** for more details). *Comparison cleaning actually dominates the run-time of blocking workflows*, with RT being proportional to the number of candidate pairs resulting from block cleaning. For this reason, EQBW and ESABW are typically slower than QBW and SABW, res

Among the sparse NN methods, kNNJ is much faster than ϵ-Join in half the datasets: D_{a5}-D_{a9}. This is counter-intuitive, given that the former approach involves a more complex functionality, sorting the candidate pairs per query entity. However, as shown in Table **??**, k=1 for these datasets (2 in D_{a8}), thus minimizing the overhead of sorting. This also explains why DkNN, which uses $k = 5$, is slower than both other techniques in D_{a5}-D_{a7} (for D_{a8}-D_{a10}, DkNN is faster than kNNJ, because the latter uses shorter q-grams that generate more candidate pairs). Overall, *the run-time of sparse NN methods is dominated by the querying time*, which consistently accounts for more than half of RT. In contrast, indexing time accounts for less than 10% in practically all cases, with the rest corresponding to cleaning.

Regarding the dense NN-methods, MH-LSH is by far the slowest similarity-based one for D_{a5} on, even though it does not apply cleaning (i.e., stop-word removal and stemming) in most cases (see Table **??**). This should be attributed to the very large number of candidates it generates during the querying phase, which prevents it from scaling to D_{a10}. The lowest run-time corresponds to CP-LSH, because it exhibits the highest PQ among the three methods. Among the cardinality-based NN methods, FAISS is consistently the fastest one. It is significantly faster than SCANN in all cases, even though they have an almost identical effectiveness. The reason is that FAISS saves the overhead of data partitioning. Most importantly, both versions of *DeepBlocker are slower than the other NN methods by a whole order of magnitude in most cases.* This is caused by the cost of automatically creating a labelleled dataset and using it for training the tuple embedding module – the number of candidates per query entity plays a minor role, which can be inferred from the relative run-time of DDB and DeepBlocker and the configurations in Table **??**. Therefore, we can conclude that *DeepBlocker emphasizes effectiveness at the cost of very low time efficiency.* Note also that the Hybrid tuple embedding module is a whole order of magnitude slower than the Autoencoder [11].

The same patterns apply in the schema-based settings. However, RT has significantly improved in most cases, especially for the largest datasets, due to the lower vocabulary size and character length. As a result, all methods are capable of processing all datasets in less than 1 second (few seconds for D_{b9}). DeepBlocker and DDB typically remain a whole order of magnitude slower than all other methods, due to the inelastic computational cost of creating a labelled dataset and using it to train the neural-based tuple embedding module.

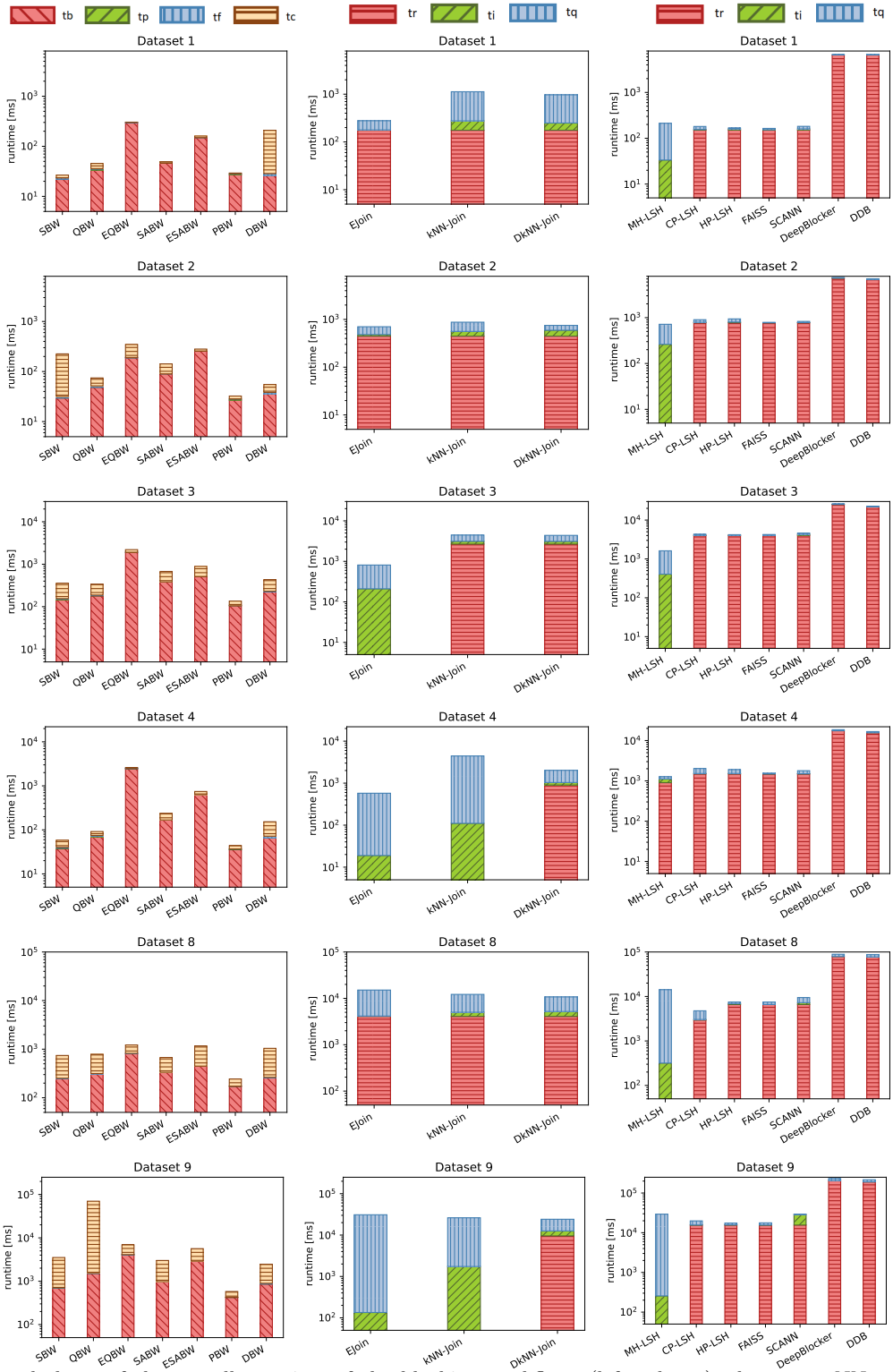

Fig. 17 The break-down of the overall run-time of the blocking workflows (left column), the sparse NN methods (middle column) and the dense NN methods (right column) for the schema-agnostic settings of D_1-D_4 and D_8-D_9.

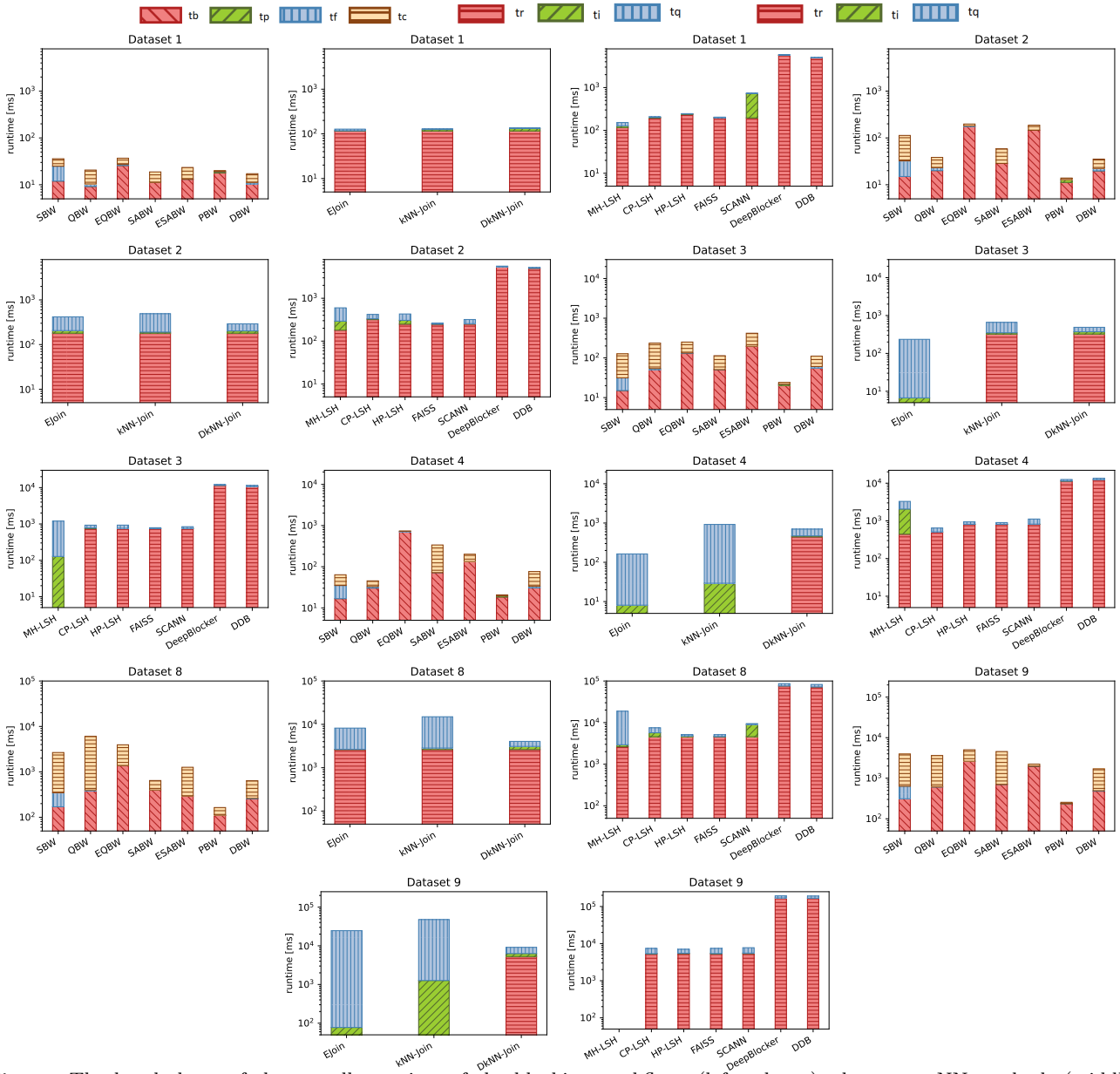

Fig. 18 The break-down of the overall run-time of the blocking workflows (left column), the sparse NN methods (middle column) and the dense NN methods (right column) for the schema-based settings of D_1-D_4 and D_8-D_9.

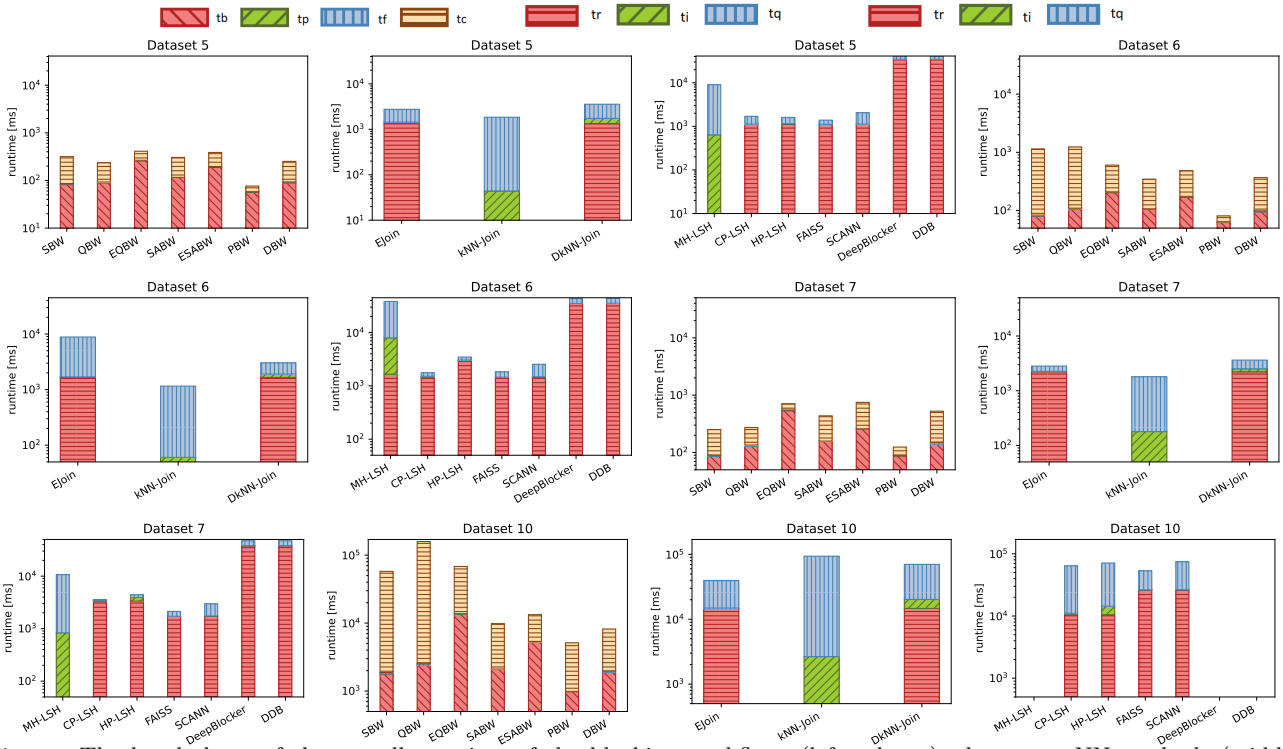

Fig. 19 The break-down of the overall run-time of the blocking workflows (left column), the sparse NN methods (middle column) and the dense NN methods (right column) for the schema-agnostic settings of D_5-D_7 and D_{10}.