Progressive Entity Matching: A Design Space Exploration

JAKUB MACIEJEWSKI, National and Kapodistrian University of Athens, Greece
KONSTANTINOS NIKOLETOS, National and Kapodistrian University of Athens, Greece
GEORGE PAPADAKIS, National and Kapodistrian University of Athens, Greece
YANNIS VELEGRAKIS, University of Trento & Utrecht University, The Netherlands

Entity Resolution (ER) is typically implemented as a batch task that processes all available data before identifying duplicate records. However, applications with time or computational constraints, e.g., those running in the cloud, require a progressive approach that produces results in a pay-as-you-go fashion. Numerous algorithms have been proposed for Progressive ER in the literature. In this work, we propose a novel framework for Progressive Entity Matching that organizes relevant techniques into four consecutive steps: (i) filtering, which reduces the search space to the most likely candidate matches, (ii) weighting, which associates every pair of candidate matches with a similarity score, (iii) scheduling, which prioritizes the execution of the candidate matches so that the real duplicates precede the non-matching pairs, and (iv) matching, which applies a complex, matching function to the pairs in the order defined by the previous step. We associate each step with existing and novel techniques, illustrating that our framework overall generates a superset of the main existing works in the field. We select the most representative combinations resulting from our framework and fine-tune them over 10 established datasets for Record Linkage and 8 for Deduplication, with our results indicating that our taxonomy yields a wide range of high performing progressive techniques both in terms of effectiveness and time efficiency.

CCS Concepts: • **Information systems → Entity resolution**.

Additional Key Words and Phrases: Progressive Entity Resolution, Nearest Neighbor Search, Blocking, Sorted Neighborhood

ACM Reference Format:
Jakub Maciejewski, Konstantinos Nikoletos, George Papadakis, and Yannis Velegrakis. 2025. Progressive Entity Matching: A Design Space Exploration. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 65 (February 2025), 25 pages. https://doi.org/10.1145/3709715

1 INTRODUCTION

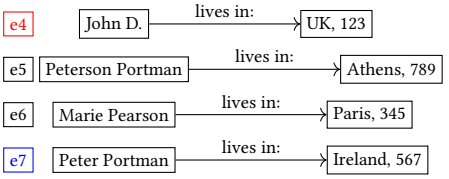
Entity Resolution (ER), often also referred to as Record Linkage (a.k.a. Clean-clean ER) or Deduplication (a.k.a. Dirty ER), is a fundamental task in data management [24]. It deals with the challenge of identifying and linking data structures, typically referred to as *entity profiles*, that represent the same real-world object [7] The linked structures are referred to as duplicates. Detecting the duplicates is crucial for boosting the performance of a wide range of data management tasks, from recommendation to question answering.

Authors' addresses: Jakub Maciejewski, National and Kapodistrian University of Athens, Greece, sdi1700080@di.uoa.gr; Konstantinos Nikoletos, National and Kapodistrian University of Athens, Greece, k.nikoletos@di.uoa.gr; George Papadakis, National and Kapodistrian University of Athens, Greece, gpapadis@di.uoa.gr; Yannis Velegrakis, University of Trento & Utrecht University, The Netherlands, i.velegrakis@uu.nl.

ACM 2836-6573/2025/2-ART65
https://doi.org/10.1145/3709715

id	First name	Last name	City	Zip Code
e1	John	Doe	London	123
e2	Peter	Portman	Dublin	567
e3	Donald	Hirsh	New Jersey	486

(a) Data source D_1



(b) Data source D_2

(e1, e4), (e2, e7), (e2, e5), ..., (e3, e6)

(c) Ordered set of candidate pairs

Fig. 1. Two clean data sources and the candidate pairs generated by a Progressive Entity Matching approach.

An ER application in order to be successful, needs to overcome a series of challenges [3, 7, 13]. The first is *Volume*. ER algorithms should scale to thousands or even millions of entity profiles. This isn't straightforward, due to the the inherently quadratic nature of ER, which is why it is addressed through filtering [32]. The second is *Variety*. ER should seamlessly apply to data sources that vary in format, schema and structure. Yet, most ER approaches are crafted either for structured, e.g., relational data or semi-structured, e.g., RDF [29]. The third is *Velocity*. Most ER solutions operate in an offline mode, i.e., they produce results only after having processed all the input data. Unfortunately, there are cases where there are restrictions and cost in computational or temporal resources. An example is the cloud environment. In these cases, partial results are required within a specific time frame [9–11, 45].

To address these challenges, we focus on *Schema-agnostic Progressive Entity Matching*. Velocity is addressed by the progressive functionality, which yields results before processing all input data through a pay-as-you-go functionality. Volume is addressed by Filtering, which restricts the computational cost to the most similar entity profiles, disregarding those dissimilar. Variety is addressed by the schema-agnostic functionality, which represents every entity profile through a concatenation of all attribute values, regardless the respective attribute names.

Consider Figure 1 that illustrates a Record Linkage case, i.e., a case of two data sources D_1 and D_2, each one being duplicate-free, meaning that it contains no matching entity profiles, but there are duplicates across the sources. For instance, e_1 matches to e_4 and e_2 to e_7. The two sources need to be merged, and to do so, the duplicates need to be detected. Furthermore, the two data sources exhibit high variety, varying both in format (structured vs semi-structured) and in schema. Typically, an ER process consists of three steps [3, 13]: Filtering, Entity Matching and Clustering. Yet, due to the small size of this example, Filtering is omitted, while Clustering lies out of the scope of our work. Therefore, we exclusively focus on Entity Matching, where a typical batch algorithm considers all pairs of entities, returning $e_1 \equiv e_4$ and $e_2 \equiv e_7$ after processing the entire data sources, i.e., after examining 12 pairs. In contrast, Progressive Entity Matching defines a processing order, as in Figure 1(c), so as to promote the most likely matches and detect most of them even if the processing is terminated prematurely.

Most existing progressive methods are independent of matching decisions, defining a *static* processing order [34, 45, 54]. Yet, they have some limitations: (i) They disregard recent advances in Filtering: they exclusively consider (meta-)blocking as a pre-processing step for addressing volume, but recent studies show that nearest neighbor search achieves a much higher performance [25, 35, 50]. (ii) They disregard recent advances in natural language processing and the dominance of the semantic entity representations that stem from pre-trained language models in both ER tasks, i.e., blocking and matching. Instead, they exclusively rely on the syntactic representation of entity profiles [25, 49, 50, 52, 56]. (iii) There is no generic framework that unifies the traditional, syntactic- and blocking-based approaches with the latest, semantic-based ones that leverage nearest neighbor search.

To address these shortcomings, we propose a novel framework for Progressive Entity Matching that is generic enough to cover any type of entity representation and of Filtering methods. Our goal is not to combine the outcome of different techniques (e.g., the syntactic- and the semantic-based), but to propose a set of generic, integrated steps that offer a series of options to researchers and practitioners, facilitating the construction of progressive pipelines that leverage diverse methods in a seamless way. To the best of our knowledge, no other framework offers this in a unifying way.

Our design space consists of four consecutive steps: (i) Filtering restricts the computational cost to the most similar pairs of candidates. (ii) Weighting assigns a similar score to each candidate pair that is proportional to the matching likelihood of the entity profiles comprising it. (iii) Scheduling leverages the similarity scores in order to define the optimal processing order that gives precedence to the duplicate pairs over the non-matching ones. (iv) Matching analytically examines each candidate pair to decide whether its entity profiles are matching or not.

We explain how the filtering step of our framework incorporates all the existing state-of-the-art approaches, from blocking- and sorting-based to those leveraging nearest neighbor search. For each approach, we discuss the corresponding weighting functions for syntactic and semantic representations. For scheduling, we propose four main algorithms based on the concept of the similarity graph [29], which includes a node for each input entity with an edge connecting each candidate pair. We stress that the existing Progressive Entity Matching approaches cover only a small portion of those generated by our framework. We also stress that the matching process lies outside the scope of this work, since it is an orthogonal issue [45, 54], with numerous recent state-of-the-art solutions based on Deep Learning [8, 49].

To test the performance of our framework, we perform a grid search to fine-tune a wide range of end-to-end pipelines over a set of 10 well-established real-world datasets for Record Linkage and of 8 for Deduplication. The experimental results indicate the best combination of filtering and scheduling algorithms in terms of effectiveness. We compare the best configuration of each filtering approach with the state-of-the-art from the literature, i.e., DeepBlocker [50] and Sparkly [35]. The former leverages semantic representations and the latter syntactic ones. This is the first time the two algorithms are applied on Progressive Entity Matching. Our results indicate that DeepBlocker consistently underperforms our semantic-based nearest-neighbor approach both with respect to effectiveness and to time efficiency. Sparkly is much faster than our syntactic-based nearest neighbor approach (due to its Spark-based parallelization), but at the same time, it is significantly less accurate. We also consider an additional state-of-the-art technique as baseline method: I-PES [11].

Overall, we make the following contributions:

(1) We introduce a generic framework for Progressive Entity Matching consisting of four steps, that organize the existing approaches, and giving rise to some new. Among the latter are the first progressive methods leveraging nearest-neighbor search.
(2) We analytically explain the configuration space of each progressive method generated by our framework. Some of the new methods are the first to apply pre-trained language models to Progressive Entity Matching.
(3) We perform an extensive experimental analysis of all Progressive Entity Matching approaches and configurations (using grid search) over 18 well-established datasets. Our experiments give valuable insights into the relative performance of the filtering and scheduling algorithms and demonstrate the superiority of our approaches over the state-of-the-art in the literature.

The implementation of our approaches is available online at: https://github.com/JacobMaciejewski/PER-Design-Space-Exploration.

2 RELATED WORK

Progressive Entity Matching methods can be distinguished in two main categories, the *static* and the *dynamic* [29]. The former generate a processing order that is independent of the matching decisions, while the latter update the processing order based on the latest matching decision(s). Our work focuses on the former for three reasons. First, they offer a more realistic setting, where scheduling is performed once, without the need of an oracle for the matching. The dynamic methods assume an oracle with 100% matching accuracy in every turn. Second, the static methods are more efficient than the dynamic, because the dynamic rearrange the candidate pairs after each matching decision. This raises the computational cost significantly. Last, but not least, most existing Progressive Entity Matching methods yield a static processing order [45, 54].

Our design space organizes the static methods proposed in [45, 54] into a unified framework that facilitates their extension and comparison. It also integrates the latest works in blocking and nearest neighbor search [28, 56], which correspond to the Filtering step and partially to the Weighting step of our design space. State-of-the-art approaches like Sparkly [35] and DeepBlocker [50] can also be integrated into our framework, however, they are used as baseline methods in our experimental analysis.

Static approaches over dynamic data have recently been studied [11] by applying Progressive Entity Matching to streaming data that is not available upfront, but arrives at varying rates. The goal is to identify duplicates soon after their arrival, while scaling to large volumes. The specific framework gives rise to three different schema-agnostic algorithms, of which the entity-centric I-PES consistently exhibits the highest performance, and is, consequently, experimentally compared to our approaches in Section 7.7.

Unlike the static progressive, the dynamic progressive rely on a perfect matching algorithm in order to iteratively rearrange the processing order of the candidate pairs. The Dynamic Progressive Sorted Neighborhood [34] organizes the sorted entities into a two-dimensional array such that after detecting a match in $A(i, j)$, the processing moves on to check $A(i + 1, j)$ / and $A(i, j + 1)$. This is a dynamic extension of our sorting-based workflows. The *pBlocking* [10] is another method that initially generates a set of blocks, that is then iteratively refined through block cleaning based on the ratio of duplicate and non-duplicate profiles as it is determined after a limited amount of matching decisions in every round. Comparison cleaning based on meta-blocking is also applied. This approach, which has been demonstrated in the *BEER system* [9], is a dynamic extension of the blocking workflows of our design space.

Disk-based dynamic methods [48] have been used in cases of extremely large datasets that cannot be entirely loaded into memory. They aim to avoid high I/O overhead by optimally scheduling the transition of data between main memory and the hard disk, while searching for the most promising pairs and defining their processing order. They leverages a cost benefit analysis to split data into partitions that are iteratively scheduled for processing. In Query-driven ER [46], the ER is performed on query results rather than entire datasets. A data lake is queried and the produced results are progressively returned after deduplication. This idea has been implemented in the BrewER system [57].

It should be stressed that the blocking workflows of our design space are based on the pipeline presented elsewhere [33], but go beyond it by including Block Filtering [25] and many more weighting schemes (ref. to W1-W14 in Section 5.3). Furthermore, the specific work [33] focuses exclusively on batch ER, missing the Scheduling phase which renders a blocking workflow suitable for Progressive Entity Matching. To the best of our knowledge, no prior work examines the performance of progressive workflows that combine a diverse set of blocking and weighting techniques with novel scheduling algorithms through a thorough experimental analysis.

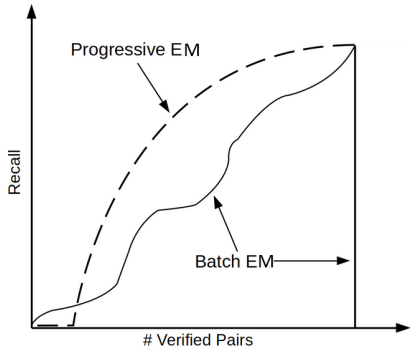


Fig. 2. Batch vs Progressive Entity Matching.

Due to the Scheduling, Active Learning based ER [12, 15, 43], AL-based for short, is similar to Progressive Entity Matching because it also assigns scores to the candidate pairs generated by Filtering. However, instead of trying to place the matching pairs before the non-matching, AL-based ER aims to minimize the number of labeled pairs that are required for training a supervised classifier for Entity Matching. To this end, it operates iteratively, selecting in each iteration the most critical unlabeled pairs, whose labeling will reinforce the distinction between the positive and the negative classes (e.g., the most uncertain pairs as determined through a committee of classifiers) [21, 29]. In other words, AL-based ER promotes pairs with a matching likelihood close to 0.5, unlike Progressive Entity Matching, which promotes pairs with a matching likelihood close to 1.0. Using the former in the place of the latter (or vice versa) would result in poor performance. In the future, it is worth exploring more elaborate techniques for combining AL-based ER with Progressive Entity Matching.

3 PROBLEM STATEMENT

Entity Resolution comes in different forms depending on the number of input data sources [3, 5, 7, 13, 41]. The main are:

• *Deduplication*, often referred to as *Dirty ER*, receives as input a single data source D that contains duplicates. The goal is to partition D into sets of duplicate entity profiles such that every set corresponds to a different real-world object.

• *Record Linkage*, often referred to as *Clean-Clean ER*, receives as input two data sources D_1 and D_2, with each one containing no dublicates in itself. Its goal is to identify the duplicate profiles across the two sources.

• *Multi-source ER* generalizes Record Linkage to more than two duplicate-free, but overlapping data sources. The goal is to cluster together the duplicate profiles from the different sources. This task can also be treated as a series of Record Linkage tasks or as a single Deduplication task, where the profiles of the identified linkages are merged into one after each step.

In all cases, the solution to ER typically consists of the two consecutive steps forming the *Filtering-Verification* framework [24, 32, 50]. First, Filtering reduces the search space to the most similar entity profiles, thus excluding the obvious non-matches. This is an approximate process that curtails the originally quadratic time complexity at the cost of missing a limited portion of the duplicate profiles. Its outcome comprises a set of candidate pairs, which is then processed by Verification: typically, a complex, time-consuming function is applied to each pair $< e_i, e_j >$, yielding either a binary decision ("match" or "non-match") or a numeric score proportional to the matching likelihood of e_i and e_j.

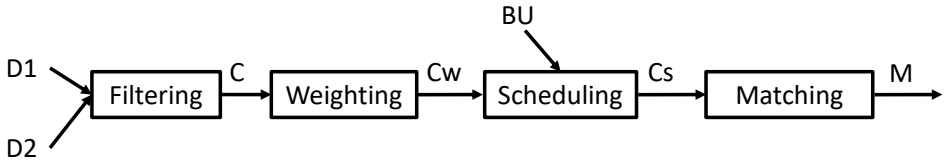


Fig. 3. The Progressive Entity Matching workflow.

A major drawback of the Filtering-Verification framework is its *batch* functionality, which yields results only after processing the entire input [29]. This is not compatible with ER applications having strict performance requirements with respect to run-time and/or computational costs [25, 35, 50]. For example, consider applying ER to large corporate data lakes that run on third-party cloud infrastructures, which charge according to the computational resources that are consumed (e.g., the AWS Lambda functions). In these settings, the preferred solution is a *progressive* functionality that produces results in a pay-as-you-go manner [29]. This requires that the matching pairs take precedence over the non-matching such that the more processing time is available or the more verifications are performed, the more duplicates are detected.

More formally, assuming that batch Entity Matching needs to verify N candidate pairs in order to process a data source D, a progressive approach should satisfy the following requirements [34, 45, 54]: (i) *Higher early effectiveness.* If a batch and a Progressive Entity Matching approach verify N' candidate pairs such that $N' \ll N$, the latter detects many more matching pairs than the former. (ii) *Same eventual effectiveness.* Upon verifying N pairs, the Progressive Entity Matching approach yields the same duplicates as the batch one. The two requirements are highlighted in Figure 2. The horizontal axis corresponds to the verified pairs, while the vertical one corresponds to recall. We define the area under the curve as **progressive recall@N**, where N is the budget of the maximum verified pairs. It takes values in $[0, 1]$, with higher values indicating higher effectiveness. In other words, the higher the progressive recall is, the more and earlier are the existing duplicates detected.

In this context, our goal can be formally described as follows:

Problem 1. *Given two data sources, D_1 and D_2, along with a budget of N verifications, **Progressive Entity Matching** produces a set of candidate pairs ordered such that progressive recall@N is maximized, while the run-time is minimized.*

Note that the above definition can be easily adapted to Deduplication and Multi-source ER as well as to a budget defined in terms of maximum run-time. Note also that the run-time excludes the Verification time [45, 54] – it only considers the time that intervenes between receiving the input data sources and returning the ordered set of candidate pairs. A similar assumption holds for progressive recall [45, 54]: it is independent of Verification performance, assuming an oracle function that always decides with 100% accuracy whether two entity profiles are matching or not.

4 DESIGN SPACE OF PROGRESSIVE ENTITY MATCHING

We now propose an architecture template for Progressive Entity Matching solution consisting of four modules applied in the following order, also illustrated in Figure 3:

(1) *Filtering* receives as input two clean data sources, D_1 and D_2, (or a single dirty one) and returns as output the set of candidate pairs C, which includes the most likely matches.
(2) *Weighting* receives as input the set of candidates pairs C defined by filtering. Its output comprises a set C_w with the same pairs, where each one is assigned to a positive weight that is proportional to the matching likelihood of its entities.
(3) *Scheduling* receives as input the user-defined budget of *BU* verifications along with the weighted candidate pairs C_w and defines their processing order in such a way that the most likely

matches are examined first. Therefore, its output comprises C_s, which is a permutation of the given set C_w.

(4) *Matching* receives C_s as input and iteratively outputs to the next candidate pair to be verified. In other words, it simply applies a matching function to the pairs in C_s according to the processing order defined by Scheduling. The set of verified pairs is denoted by M.

For each module, we discuss a diverse approaches. First for Filtering and Weighting (Section 5), and then for Scheduling (Section 6). An exception is the Matching step, which is common to both batch and progressive solutions, with a bulk of recent works combining language models with deep learning to achieve high accuracy [8, 30, 50]. This is why Matching is out of the scope of this work. The same applies to the clustering step, which is necessary for an end-to-end ER pipeline. The integration of the state-of-the-art clustering techniques for Record Linkage [27] and Deduplication [14, 51] in our framework is left for future work.

5 FILTERING & WEIGHTING

Due to the quadratic time complexity of ER, Filtering is necessary for curtailing the search space by discarding the apparent non-matches [3, 35, 50]. In other words, Filtering retains only the most likely matches, as determined by their high similarity, through a quick and approximate process of low time complexity, which can be accomplished in one of the following ways:

(1) **NN workflows.** The input entity profiles are embedded into dense, high-dimensional vectors through pre-trained language models. In Record Linkage, one of the two data sources is indexed, while the vectorized profiles of the other query the index to retrieve their semantically k nearest neighbors.

(2) **Join workflows.** Unlike the semantic focus of NN workflows, the join workflows leverage syntactic similarities: they follow the same approach of indexing and querying, but convert the input entity profiles into sparse, multi-dimensional numeric vectors. Every dimension in these vectors corresponds to a different character or token n-gram, with a weight proportional to its frequency in the values of the respective profile.

(3) **Blocking workflows**. Signatures, called blocking keys, are extracted from the attribute values of each profile. Each signature s creates a separate block b_s containing all entities associated with s. The resulting blocks are refined based on the assumption that the larger a block is, the less likely it is to contain distinctive information. The refined blocks are then converted into a graph, where block sharing is translated into node adjacency. The edges are weighted by metrics quantifying the block co-occurrence of the corresponding entity profiles.

(4) **Sorting-based Workflows.** They rely on the same signatures as blocking workflows, but define as candidates the pairs with similar (not identical) blocking keys. The entity profiles are alphabetically ordered, according to the signatures extracted from their attribute values. This results in a sorted list of entities, which is iteratively processed using sliding windows of increasing size. The matching likelihood of two profiles is proportional to their co-occurrence frequency in these windows.

Following [45], all approaches operate in a schema-agnostic manner that disregards all attribute names, but considers all attribute values. None of them involves a learning-based functionality that requires a labelled dataset. Instead, they all rely on heuristics, trading high time efficiency at the cost of lower effectiveness. Below, we elaborate on their functionality along with the corresponding weighting functions and the configuration parameters.

Input: $\mathcal{D}_1$, $\mathcal{D}_2$:The data sources to be matched, k: the maximum number of candidates per query entity, LM: the Language Model that vectorizes each entity, Sim: the similarity function between two embedding vectors. IS: the indexing scheme.
Output: C: the resulting set of candidate pairs

```
foreach entity e_i ∈ D_1 do
    E(e_i) ← LM.Embed(e_i);
    I.index(E(e_i));
end
C ← {} ;                                          // Set of candidate pairs
foreach entity e_j ∈ D_2 do
    E(e_j) ← LM.Embed(e_j);
    C_j ← I.getNN(Sim, E(e_j), k);
    C ← C ∪ C_i;
end
return C;
```

Algorithm 1: Outline of the NN and Join workflows.

5.1 NN Workflows

The functionality of NN workflows is outlined in Algorithm 1. The input comprises the two data sources to be matched along with the maximum number of candidates per query entity, corresponding to its most probable matches. For k, we consider three representative values (i.e., $k \in \{1, 5, 10\}$). The input should also specify the pre-trained language model LM that converts the given entities into embedding vectors. Following [56], we consider 10 established LMs, disregarding AlBERT [18] and XLNet [55], due to their consistently high run-time and low effectiveness:

- the main static ones, which always associate every token or character n-gram with the same precalculated embedding vector: Word2Vec [22, 23], FastText [2] and Glove [36].
- the main BERT-based ones, which offer a per-token contextual vectorization: BERT [6], DistilBERT [42], and RoBERTa [19].
- the main SentenceBERT-based ones, which associate the entire schema-agnostic representation of every entity with a single context-aware embedding vector: S-MiniLM [53], S-MPNet [47], S-GTR-T5 [37] and S-DistilRoBERTa [19].

Another configuration parameter is the similarity function $Sim : \mathbb{R} \rightarrow [0, 1]$ between two multidimensional embedding vectors $v, w \in \mathbb{R}^n$. We consider two established options:
(1) The Euclidean one, which considers the magnitude and direction of the two vectors, but is sensitive to their dimensionality [56]:

$$s_{\text{euclidean}}(\mathbf{v}, \mathbf{w}) = \frac{1.0}{1.0 + d_{\text{euclidean}}(\mathbf{v}, \mathbf{w})} = \frac{1.0}{1.0 + \sqrt{\sum_{i=1}^{n}(v_i - w_i)^2}}$$

(2) The cosine similarity, which exclusively considers the angle between the two input vectors:

$$s_{\text{cosine}}(\mathbf{v}, \mathbf{w}) = 1.0 - d_{\text{cosine}}(\mathbf{v}, \mathbf{w}) = \frac{\sum_{i=1}^{n} v_i \cdot w_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \cdot \sqrt{\sum_{i=1}^{n} w_i^2}}$$

The final configuration parameter is the *indexing scheme*, which exclusively applies to Record Linkage, designating which of the input data sources will be indexed – leaving the other one as the query set. Three are the possible options: (1) indexing the smallest source, (2) indexing the largest one, or (3) both.

In Algorithm 1, we index the first data source D_1 and query with the second one, D_2. More specifically, every entity in D_1 is converted to the embedding vector of the given LM, after concatenating all its attribute values in a sentence without any special tokens [56] (Line 2). The resulting vector is indexed by a state-of-the-art tool for nearest neighbor search (Line 3). Based on [1], we use FAISS [16] for this purpose, as it constitutes one of the fastest and most effective libraries for indexing high dimensional embedding vectors and retrieving the nearest neighbors per query.

Next, all entity profiles in D_2 are vectorized one by one, using the same LM (Lines 6-7). Every embedding vector is posed as a query to the index I, returning the k most similar entities from D_1 (Line 8). These candidates are added to the set of candidate pairs C, which is returned as output (Lines 9-11). Note that internally, C associates every candidate pair with its similarity score as determined by the given similarity function, Sim.

The time complexity of Algorithm 1 depends on the time complexity of each query to the index I, i.e., $O(|D_2| \cdot |q_I(D_1)|)$, where $|q_I(D_1)|$ is the time complexity of a single entity on I, after having indexed D_1 entities. Note that $|q_I(D_1)|$ is constant and $|q_I(D_1)| \ll |D_1|$, due to FAISS' internal functionality, which partitions the indexed vectors in such a way that every query is restricted to a few partitions (rather than the entire indexed data source). Theoretically, the time complexity of vectorizing all input entity profiles is linear and, thus, lower than the cost of querying. Similarly, the space complexity of Algorithm 1 is determined by the the cost of storing the vectors of D_1 entities in memory, i.e., $O(|D_1|)$.

NN workflows have not been applied to Progressive Entity Matching before.

5.2 Join Workflows

Approaches of this type follow Algorithm 1, involving two phases:

(1) the *Indexing phase* in Lines 1-4, where one of the input datasets is transformed to a structure suitable for the fast detection of nearest neighbors, and
(2) the *Querying phase* in Lines 5-10, where entities from the other input dataset query the indexing structure to detect their nearest neighbors.

The only difference between Join and NN workflows lies in the vectorization approach: unlike the dense embedding vectors of the latter, which map entities to the semantic space of language models, the Join workflows leverage sparse multi-dimensional vectors directly extracted from the attribute values of the input entities. More specifically, two functions are combined to this end:

(1) The *tokenization function* converts the concatenated attribute values of each entity into a set of character or token n-grams: $n \in \{3, 4, 5\}$ in the former case and $n \in \{1, 2\}$ in the latter one.
(2) The *feature scoring function* associates every dimension in the sparse vector with a numerical score. We consider 3 options:
    (a) Boolean scores (BS) indicate the presence or absence of an n-gram in the attribute values of the given entity.
    (b) Term-frequency scores (TF) indicate how many times each n-gram appears in the attribute values of the given entity. Note that the frequency of each n-gram is normalized by the highest frequency within the given entity.
    (c) TF-IDF scores (TF-IDF) extends TF to encompass the n-gram's importance across the entire input dataset through the logarithmically scaled inverse fraction of the number of profiles that contain the n-gram.

In the following, we consider all combinations of tokenization and feature scoring functions. Note that instead of FAISS, we use a standard inverted index for quickly retrieving the candidates per query entity, due to the sparse vectors used by Join workflows. Note that Join workflows have not been applied to Progressive Entity Matching before.

Input: $\mathcal{D}_1$, $\mathcal{D}_2$:The data sources to be matched, WS: the weighting scheme
Output: $\mathcal{G} = \mathcal{V} \times \mathcal{E}$: the similarity graph

1 $\mathcal{B} \leftarrow tokenBlocking(\mathcal{D}_1, \mathcal{D}_2)$;
2 $\mathcal{B}' \leftarrow blockPurging(\mathcal{B})$;
3 $\mathcal{B}'' \leftarrow blockFiltering(\mathcal{B}')$;
4 $\mathcal{G} \leftarrow \{\}$;
5 **foreach** *entity* $e_i \in D$ **do**
6 $\quad B_{e_i} \leftarrow \bigcup_{b \in \mathcal{B}''} b \mid e_i \in b$;
7 $\quad$ **foreach** *block* $b \in B_{e_i}, e_i \in \mathcal{D}_1$ **do**
8 $\quad\quad$ **foreach** $e_j \in b : e_j \in \mathcal{D}_2$ **do**
9 $\quad\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{n_j\}$;
10 $\quad\quad\quad \mathcal{E} \leftarrow \cup \{(n_i, n_j)\}$;
11 $\quad\quad\quad weight(n_i, n_j) = WS(n_i, n_j, \mathcal{B}'')$
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 **end**
15 **return** $\mathcal{G}$;

Algorithm 2: Outline of the blocking workflows.

5.3 Blocking Workflows

The functionality of these solutions is outlined in Algorithm 2.

Token Blocking [25, 33] is first applied (Line 1), generating a separate block for each token in the attribute values of the given entity profiles. Any other blocking method can be used, too, but Token Blocking is the only parameter-free one.

Next, Block Purging [25, 25] is applied (Line 2) to remove oversized blocks, which comprise a large number of pairs, but very few (if any) are matching and have no other block in common. Similar to Token Blocking, this is a parameter-free approach. The core assumption is that *the larger a block is, the more likely it is to contain repeated pairs that are not duplicates.*

The same assumption lies at the core of the subsequent step (Line 3), Block Filtering [25]: it removes every entity from a specific portion of its largest blocks, thus reducing the unnecessary pairs that involve non-matching entities at a small cost in recall. Following [31], this ratio is 80%.

Based on the blocks resulting from the initial steps, a similarity graph $\mathcal{G}$ is created (Lines 4-14). This is an undirected graph, whose nodes correspond to entities and its edges connect the candidate pairs (Lines 9-10). Every edge is weighted according to the characteristics of blocks containing every one of the adjacent entities as well as the characteristics of common blocks (Line 11). These characteristics include the number of blocks, their *size*, i.e., their total number of entities, and their *cardinality*, i.e., their candidate pairs.

More specifically, the weighting scheme is the sole configuration parameter of the blocking workflows. Its rationale is similar to that of Block Purging and Block Filtering: 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 this context, the possible weighting schemes for two entities, e_i and e_j, are the following [25, 33]:

W1) Common Blocks: $\mathbf{CB} = |B_i \cap B_j|$, where B_x stands for the set of blocks containing entity e_x and $|B_x|$ for its size.
W2) Cosine=$|B_i \cap B_j|/\sqrt{|B_i| \cdot |B_j|}$=$CB/\sqrt{|B_i| \cdot |B_j|}$.
W3) Dice=$2 \cdot |B_i \cap B_j|/(|B_i| + |B_j|)$=$2 \cdot CB/(|B_i| + |B_j|)$.
W4) Jaccard=$|B_i \cap B_j|/|B_i \cup B_j|$=$CB/(|B_i| + |B_j| - CB)$.

Input: $\mathcal{D}_1$, $\mathcal{D}_2$:The data sources to be matched, w: the window size, WS: the weighting scheme
Output: C: the resulting set of candidate pairs

```
1  P ← SortedNeighborhood(D1, D2);
2  C ← {} ;                                              // Set of candidate pairs
3  foreach entity e_i ∈ D1 do
4  |  N_i ← {} ;                                         // Set of neighbors
5  |  positions_i ← P.getPositions(e_i);
6  |  foreach position ps ∈ positions_i do
7  |  |  N_i ← N_i ∪ (P.getNeighbors(ps, w) ∩ D2);
8  |  end
9  |  foreach entity e_j ∈ N_i do
10 |  |  sim_{i,j} = P.getSimilarity(e_i, e_j, WS);
11 |  |  C ← C ∪ (e_i, e_j, sim_{i,j});
12 |  end
13 end
14 return C;
```

Algorithm 3: Outline of the sorting-based workflows.

W5) Size Normalized Common Blocks:**SN-CB**=$\sum_{b \in B_i \cap B_j} 1/|b|$.
W6) Size Normalized Cosine: $SN\text{-}CB/\sqrt{SN\text{-}B_i \cdot SN\text{-}B_j}$, where $SN\text{-}B_i = \sum_{b \in B} 1/|b|$, with SN denoting size normalization and $|b|$ symbolizing the number of entities in block b.
W7) Size Normalized Dice: $2 \cdot SN\text{-}CB/(SN\text{-}B_i + SN\text{-}B_j)$.
W8) Size Normalized Jaccard: $SN\text{-}CB/(SN\text{-}B_i + SN\text{-}B_j - SN\text{-}CB)$.
W9) Cardinality Normalized Common Blocks: **CN-CB**=$\sum_{b \in B_i \cap B_j} 1/||b||$.
W10) Cardinality Normalized Cosine: CN-Cosine= $CN\text{-}CB/\sqrt{CN\text{-}B_i \cdot CN\text{-}B_j}$, where $CN\text{-}B_i = \sum_{b \in B} 1/||b||$, with CN denoting cardinality normalization and $||b||$ symbolizing the number of pairs in block b.
W11) Cardinality Normalized Dice: CN-Dice=$2 \cdot CN\text{-}CB/(CN\text{-}B_i + CN\text{-}B_j)$.
W12) Cardinality Normalized Jaccard: CN-Jaccard=$CN\text{-}CB/(CN\text{-}B_i + CN\text{-}B_j - CN\text{-}CB)$.
W13) Enhanced Common Blocks: ECB=$CB \cdot \log \frac{|B|}{|B_j|} \cdot \log \frac{|B|}{|B_j|}$.
W14) Enahnced Jaccard: EJS=Jaccard $\cdot \log \frac{|V|}{|v_j|} \cdot \log \frac{|V|}{|v_i|}$.

Note that the time and space complexity of Algorithm 2 is determined by the number of pairs in the final set of blocks $\mathcal{B}''$, i.e., $O(||\mathcal{B}''||)$. Note also that only CN-CB has been applied to Progressive Entity Matching before (it is called ARCS in [45]).

5.4 Sorting-based Workflows

The functionality of these solutions is outlined in Algorithm 3.

Initially, Sorted Neighborhood is applied (Line 1): first, it alphabetically sorts all tokens appearing in the attribute values of all input entities and then, it sorts in *random* order the entities corresponding to each token [45]. The resulting sorted list of entities is stored in an array P. A window w slides over this list to detect the candidate pairs. Its size w is fixed, given as a configuration parameter that should be at least 2 so that at least two entities co-occur in each window. We consider all integers in $[2, 10]$.

Subsequently, for each input entity, we retrieve its positions in the array P (Lines 3-5); each position ps yields a neighborhood, which includes all other entities in positions $P[ps + 1]$, $P[ps + 2]$,..., $P[ps + w]$, where w is the current size of the window. These entities are the candidate pairs

of the current entity e_i. They are all placed in a set N_i comprising the neighbors of e_i (Lines 6-8). Note that the same entity might appear multiple times in the neighborhood of e_i, due to different, contiguous tokens. Every neighboring entity e_j is then considered as a matching candidate of e_i (Lines 9-12) based on a similarity that stems from the closeness of their associated positions. The weighted pairs are aggregated in the set of candidate pairs that is returned as output (Lines 11-14).

We consider the following options for the *weighting scheme* WS that is given as configuration parameter to compute the similarity between two candidate matches:

(1) Absolute Co-occurrence Frequency counts the number of positions that co-occur in the window of size w: $\mathbf{ACF}(e_i, e_j, w) = |\{|ps_i - ps_j| < w : ps_i \in positions_i \wedge ps_j \in positions_j\}|$, where $positions_x$ is the set of positions associated with entity e_x.
(2) Normalized Co-occurrence Frequency, which is inspired from the Jaccard similarity: $NCF(e_i, e_j, w) = \frac{ACF(e_i,e_j,w)}{|positions_i|+|positions_j|-ACF(e_i,e_j,w)}$.
(3) Dice Normalized Co-occurrence Frequency: $DNCF(e_i, e_j, w) = 2 \times \frac{ACF(e_i,e_j,w)}{|positions_i|+|positions_j|}$.
(4) Cosine Normalized Co-occurrence Frequency: $CNCF(e_i, e_j, w) = \frac{ACF(e_i,e_j,w)}{\sqrt{|positions_i|\times|positions_j|}}$.
(5) Inverse Distance, which sums the inverse distances between two positions that are located within the same window w: $ID(e_i, e_j, w) = \sum \frac{1}{|p_i-p_j|} \forall p_i \in positions_i, p_j \in positions_j, |p_i - p_j| < w$.

Note that only NCF has already been applied to Progressive Entity Matching (it is called RCF in [45]). Note also that Algorithm 3 entails another parameter, the *functionality scope*, which can be:

(1) *Local Scope* is the one presented in Algorithm 3, emitting all candidate pairs for a particular, predetermined window size w.
(2) *Global Scope* repeats the processing in Lines 3-13 for a range of window sizes: from 2 to the given size w, which in this case sets the maximum value. In each iteration, the similarity scores of the candidate pairs identified in smaller windows are updated by adding the scores from the current window.

The time complexity of Algorithm 3 is dominated by the computation of candidate pair weights and the sorting of the tokens appearing in the attribute values of the input entities, i.e., $O(|C|+|T| \cdot \log |T|)$, where $|T|$ is the number of these tokens (we expect $|D_1| \ll |T|$ and $|D_2| \ll |T|$, because each entity contains multiple tokens in its attribute values). The space complexity is dominated by the size of the array P storing the sorted list of entities, $O(|P|)$.

5.5 Application to Deduplication

The above are crafted for Record Linkage, where the input comprises of two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, but can be easily adapted for the case of Deduplication, where the input is a single dataset $\mathcal{D}$ with duplicates in itself. For that, some minor changes are required in Algorithm 1. Line 1 needs to index $\mathcal{D}$, Line 6 to use all entities in $\mathcal{D}$ as queries, and Line 9 to ensure that no duplicate pairs are added in the output set C (i.e., that pair $<e_i, e_j>$, $i \neq j$, does not appear in the form $<e_j, e_i>$). This can be accomplished by predetermining the place of every entity in every new pair $<e_x, e_y>$ added to C, by requiring the entity with the lower id is placed on the left side of the pair. This unified form allows for automatically discarding duplicate pairs, given that C is a set. Algorithm 2 also requires minor changes to adapt to Deduplication. Line 1 provides as input to Token Blocking only the $\mathcal{D}$, while the entities e_i in Line 7 and e_j in Line 8 should be different (i.e., $i \neq j$), as both belong to $\mathcal{D}$. Finallly, Algorithm 3 requires that in Line 1, the Sorted Neighborhood receives as input only $\mathcal{D}$, while Line 3 goes through all entities in $\mathcal{D}$. In Line 7, instead of ensuring that every neighbor e_j is from a different data source, we need to ensure that it is different from e_i (i.e., $j \neq i$). Finally, every

pair $<e_i, e_j>$ added to the output set C in Line 11 should be ordered, similar to the Algorithm 1 adaptation, i.e., $< e_x, e_y > \Rightarrow x < y$.

6 SCHEDULING

The goal of Scheduling is to define the optimal processing order of the weighted candidate pairs produced by the two previous modules in Figure 3. Ideally, all matching pairs precede all non-matching ones. To this end, Scheduling receives as input the budget BU specified by the user along with the set of weighted candidate pairs C_w from Algorithms 1 and 3 or the similarity graph from Algorithm 2. The two forms of input are equivalent, and actually the former is transformed into the latter by creating an undirected graph $G = (V, E)$, where there is a separate node in V for every input entities, while the edges in E connect the candidate pairs and are weighted according to the respective similarity score.

Based on the similarity graph, the scheduling algorithms are distinguished into those focusing on edges or nodes. The former operate at a global level, considering all pairs, and the latter, at a local one, operating at the level of neighborhoods (i.e., they define a separate processing order for the candidates of each entity). More specifically, we introduce the following scheduling algorithms:

(1) **Edge-centric Scheduling** (EC). It defines a global processing order by sorting all pairs in decreasing weight so as to retain the BU top ranked ones. Its time complexity is $O(|E| \log |E|)$, but its space complexity is restricted to $O(|V| + |E|)$, because it suffices to use a priority queue that always contains the top-BU weighted pairs ($BU \ll |E|$).
(2) **Node-centric Scheduling.** First, it assigns a score to each node, which is equal to the average similarity in its neighborhood. Then, it sorts the nodes in decreasing score. Finally, it orders the edges of each node neighborhood in decreasing weight. Its overall time complexity is $O(|V|(\log |V| + |\bar{N}| \log |\bar{N}|))$, where $\bar{N}$ is the average size of a node neighborhood. Its space complexity is $O(|V|+|E|)$. There are two variants for the processing order of the candidate pairs:
   - **Depth-First Search** (DFS) starts with the top-weighted node, prioritizing all its edges in decreasing weight, then does the same with the next weighted node and so on.
   - **Breadth-First Search** (BFS) iteratively goes through the sorted list of nodes and in each round, it prioritizes the next top-weighted edge of the current node, if any.
(3) **Hybrid.** It combines the operation of all the above algorithms. First, it computes the average node neighborhood per entity. During this process, it keeps in memory the best edge/candidate pair per neighborhood. These edges are globally sorted in decreasing weight. This is equivalent to applying BFS for one iteration. If the budget is not exhausted after processing these top-weighted pairs, it applies DFS: it sorts all nodes in decreasing average weight and, starting with the top-weighted one, it prioritizes all edges in its neighborhood, except for the top-weighted one, which has already been processed. Then, it moves to the next weighted node and so on. The time and space complexities are the same as in node-centric scheduling.

The similarity graph is bipartite in the case of Record Linkage. Hence, for the node-centric and the hybrid algorithms, it suffices to weight and sort only the nodes of one partition (no repeated pairs are included in the output of Scheduling). Among these algorithms, only the Hybrid one has already been applied to Progressive Entity Matching (see Progressive Profile Scheduling in [45]).

7 EXPERIMENTAL ANALYSIS

The goal of our experimental evaluation is threefold:

(1) To identify most effective solutions per filtering type. We discuss the performance of NN, join, blocking and sorting-based workflows in Sections 7.2, 7.3, 7.4 and 7.5, respectively.

	D_1 [44]	D_2 [17]	D_3 [17]	D_4 [17]	D_5 [26]	D_6 [26]	D_7 [26]	D_8 [24]	D_9 [17]	D_{10} [31]
D_α	Rest.1	Abt	Amazon	DBLP	IMDb	IMDb	TMDb	Walmart	DBLP	IMDb
D_β	Rest.2	Buy	GB	ACM	TMDb	TVDB	TVDB	Amazon	GS	DBpedia
$\lvert D_\alpha \rvert$	339	1,076	1,354	2,616	5,118	5,118	6,056	2,554	2,516	27,615
$\lvert D_\beta \rvert$	2,256	1,076	3,039	2,294	6,056	7,810	7,810	22,074	61,353	23,182
$\lvert Dup \rvert$	89	1,076	1,104	2,224	1,968	1,072	1,095	853	2,308	22,863
CP	$7.7{\cdot}10^5$	$1.2{\cdot}10^6$	$4.1{\cdot}10^6$	$6.0{\cdot}10^6$	$3.1{\cdot}10^7$	$4.0{\cdot}10^7$	$4.7{\cdot}10^7$	$5.6{\cdot}10^7$	$1.5{\cdot}10^8$	$6.4{\cdot}10^8$

Table 1. The Record Linkage data sets used in our experiments. $|D_x|$ denotes the number of entities in data source D_x, $|Dup|$ the number of duplicates in the groundtruth and CP the Cartesian product.

	De_1 [51]	De_2 [34]	De_3 [51]	De_4 [33]	De_5 [33]	De_6 [33]	De_7 [33]	De_8 [33]
D	Cora	CDdb	Product	10K	50K	100K	200K	300K
$\lvert D \rvert$	1,878	2,161	9,763	10^4	$5{\cdot}10^4$	10^5	$2{\cdot}10^5$	$3{\cdot}10^5$
$\lvert Dup \rvert$	62,892	1,085	299	8,705	43,071	85,497	172,403	257,034
CP	$1.8{\cdot}10^6$	$2.3{\cdot}10^6$	$4.8{\cdot}10^7$	$5.0{\cdot}10^7$	$1.3{\cdot}10^9$	$5.0{\cdot}10^9$	$2.0{\cdot}10^{10}$	$4.5{\cdot}10^{10}$

Table 2. The Deduplication data sets used in our experiments. $|D|$ denotes the number of entities in data source D, $|Dup|$ the number of duplicates in the groundtruth and CP the Cartesian product.

Filtering type	Parameter Values
Common parameters	scheduling algorithm ∈ {DFS, BFS, TOP, HB} budget ∈ $\{i \times \lvert Dup \rvert, i \in [1, 10]\}$
NN workflows	indexing scheme ∈ {smallest, largest, both} similarity function ∈ {cosine, Euclidean} number of nearest neighbors ∈ [1, 5, 10] language model ∈ {The 10 models in Sec. 5.1}
Join workflows	indexing scheme ∈ {smallest, largest, both} similarity function ∈ {cosine, Euclidean} number of nearest neighbors ∈ [1, 5, 10] weighting scheme ∈ {BW, TF, TF-IDF} tokenizer ∈ {word unigrams, word bigrams, character n-grams $n \in [3, 5]$
Blocking workflows	weighting scheme ∈ {All WS from section 5.3}
Sorting-based workflows	window size ∈ $[1, 2, \ldots, 10]$ weighting scheme ∈ {ACF, NCF, DNCF, CNCF, ID} functionality scope ∈ {local, global}

Table 3. Configuration parameters per module.

(2) To assess the relative performance of the single best solution per filtering type in terms of progressive recall, run-time and memory consumption. This is examined in Section 7.6.
(3) To compare the overall best solution of our architecture template with four state-of-the-art baseline methods, namely DeepBlocker [50], Sparkly [35], Progressive Block Scheduling [45] and I-PES [11], with respect to progressive recall, run-time and memory footprint both in Record Linkage and in Deduplication. This analysis is performed in Section 7.7.

7.1 Experimental Setup

All experiments were implemented in Python version 3.8. All experiments were executed on a server running Ubuntu 22.04 LTS, equipped with 64GB RAM, an Intel Core i7-9700K @ 3.60GHz and an NVIDIA GeForce RTX 2080. The technical characteristics of the datasets used in our experiments are reported in Tables 1 and 2 for Record Linkage and Deduplication, respectively, in

Scheduling Algorithm	Indexing Scheme	Similarity Function	Language Model	k
EC	largest	Euclidean	S-GTR-T5	5
BFS	**both**	**Euclidean**	**S-GTR-T5**	**5**
DFS	both	Cosine	S-GTR-T5	1
Hybrid	both	Euclidean	S-GTR-T5	5

Table 4. The best NN workflows from our design space, with the overall top performer highlighted in bold.

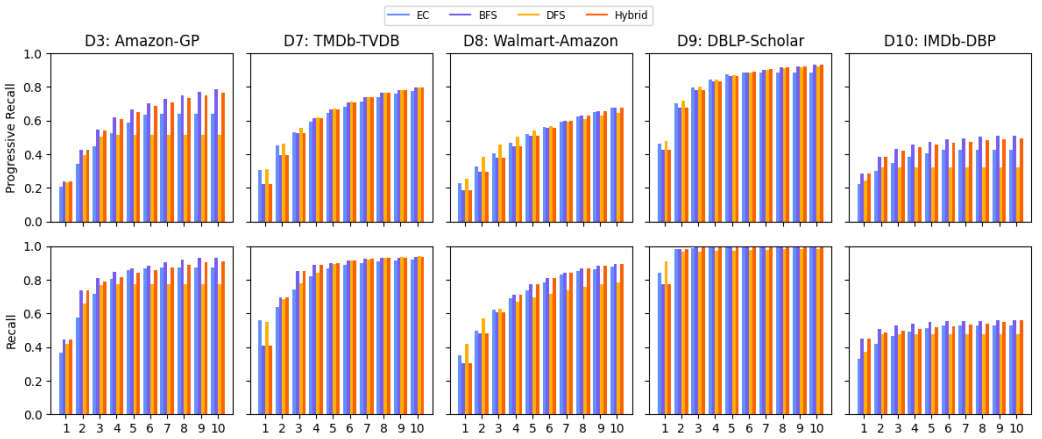


Fig. 4. Progressive recall and recall of the NN workflows in Table 4 across all budgets over selected datasets.

increasing order of computational cost in terms of the Cartesian product (i.e., last line of the table). All datasets are popular in the literature [17, 24, 33, 50]. See the Appendix [20] for more details.

In each dataset, we consider 10 different budgets, $BU_1, \ldots, BU_{10}$, where $BU_n = n \times |Dup|$, with $|Dup|$ denoting the number of duplicates in the corresponding dataset. For each budget, we perform grid search, considering all solutions generated by architectural template. Table 3 summarizes the considered solutions per filtering type. There are 180 and 270 different solutions of NN and join workflows, respectively, while the blocking and sorting-based workflows yield 14 and 100, respectively. These solutions are combined with the four different scheduling algorithms presented in Section 6. Due to lack of space, we cannot report the performance of all solutions generated in this way. Instead, for each filtering type, Sections 7.2-7.5 report only the best solution per Scheduling algorithm with respect to the *average distance from the top*.

More specifically, for each dataset, budget and filtering type, we first estimate the maximum progressive recall across all considered solutions and then, we estimate the distance of each solution from this maximum. We call this measure "*distance from the top*" and formally define it as: DFT=1-PR(so)/PR_{max}, where $PR(so)$ is the progressive recall of solution so and PR_{max} the overall maximum value. We estimate the average DFT of each solution across all datasets and budgets, and Sections 7.2-7.5 discuss the performance of the solution per scheduler with the lowest mean DFT.

7.2 NN workflows

The best NN workflows per scheduling algorithm across all datasets and budgets are reported in Table 4. They all employ the S-GTR-T5 language model, which is identified as the most effective one in [56], too. Most solutions combine the Euclidean similarity with 5 candidates per query entity. The only exception is DFS, where Cosine similarity takes a minor lead as long as a single candidate is returned per query entity. This essentially means that Scheduling is applied to the nearest neighbor per input entity, thus rendering the depth search inapplicable. Finally, most solutions

Scheduling Algorithm	Similarity Function	Tokenizer	Weighting Scheme	IndexingScheme	k
EC	Cosine	token unigram	TF-IDF	largest	10
BFS	**Euclidean**	**character 5-gram**	**TF-IDF**	**both**	**5**
DFS	Cosine	token unigram	TF-IDF	both	1
Hybrid	Euclidean	character 5-gram	TF-IDF	both	5

Table 5. The best Join workflows from our design space, with the overall top performer highlighted in bold.

index and query both data sources, a configuration that is more robust in all datasets. Only EC exclusively indexes the largest data source, using the smallest one as a query set. For this scheduling algorithm, the end result is practically identical with that of indexing both data sources, due to the large number of repeated candidate pairs generated by the latter, which are eliminated when EC merges them in a globally sorted candidate set.

The performance of these solutions with respect to progressive recall and to recall across all budgets in D_3 and D_7-D_{10} is reported in Figure 4. The performance for the rest of the datasets is presented in Figure 10 in the Appendix [20], together with the detailed memory requirements and the run-times in Figures 11 and 12, respectively. The memory footprint is around 1 GB in all cases, as it is dominated by the high-dimensional embedding vectors of the S-GTR-T5 model, while the differences in the run-time are insignificant, as it remains below 10 seconds in almost all cases.

Progressive recall yields two different patterns. In most datasets, D_2-D_6 and D_{10}, the BFS solution is the top performer, with the Hybrid one following in close distance: in half the cases, their average DFT, across all budgets, is practically identical, while their difference in D_2, D_3 and D_{10} is less than 3%. The EC solution consistently ranks third in these datasets except D_4, with an average progressive recall lower than BFS by 15% to 21%. The DFS underperforms all other algorithms, with its DFT increasing with the increase of the budget. This should be expected, because it identifies a single candidate per input entity, failing to provide more candidates, despite the largest budget.

This situation is reversed the remaining four datasets, i.e., D_1 and D_7-D_9. In these datasets, the number of duplicates is much lower than the total number of input entities. As a result, the candidates gathered by DFS suffice for maximizing the progressive recall. EC follows in close distance, with BFS and Hybrid exhibiting practically identical performance, ranking last.

The above patterns apply to all budgets in the corresponding dataset, i.e., the relative performance of the considered solutions is consistent across the 10 budgets in each dataset. This means that, in general, there is a high correlation between DFS and EC and a stronger one between BFS and Hybrid. The overall best NN solution applies the BFS algorithm to the 5 most similar candidates for each input entity according to the Euclidean similarity and the S-GTR-T5 embedding vectors. To this attests the relative performance of the NN solutions with respect to recall: BFS is consistently the top performer in half the datasets (D_2, D_3, D_5, D_6, D_{10}) typically followed by Hybrid, EC and DFS (in that order). Only in D_1, the situation is reversed, with DFS taking the lead. Yet, the differences are much smaller (insignificant in D_4, D_7, D_8, D_9) than those progressive recall.

7.3 Join workflows

The best join workflows per scheduling algorithm across all datasets and budgets are reported in Table 5. They all use TF-IDF, as it conveys more information than BW and TF. The BFS and the Hybrid solutions have the same configuration, whereas the EC and the DFS differ only in the indexing scheme and the number of nearest neighbors. The latter indexes and queries with both data sources to ensure robustness, while the former indexes only the largest data source, because its global sorting yields the same results as indexing both sources. DFS considers only the nearest neighbor of per profile, performing no depth search in practice.

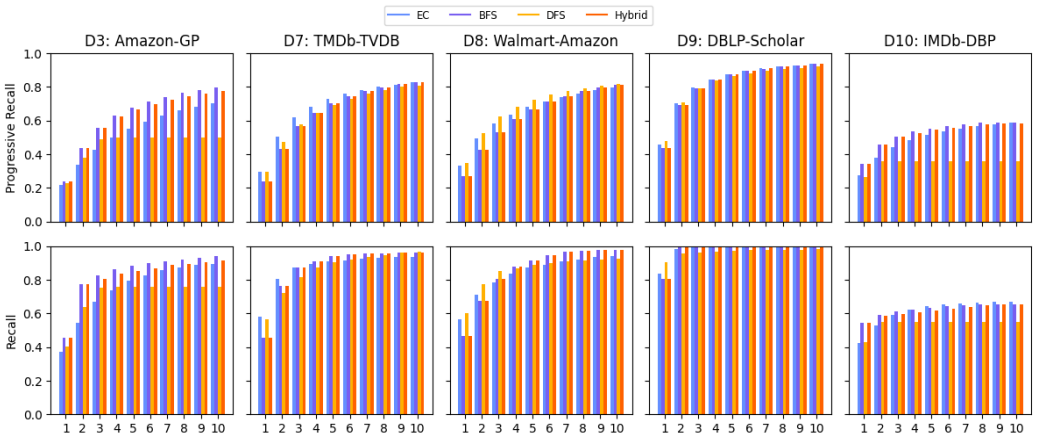


Fig. 5. Progressive recall and recall of the join workflows in Table 5 across all budgets over selected datasets.

The progressive recall of these join solutions per scheduling algorithm across all budgets in D_3 and D_7-D_{10} as well as the corresponding recall are reported in Figure 5 (refer to Figure 13 in the Appendix [20] for the rest of the datasets alongside the memory requirements and the run-times in Figures 14 and 15, respectively. Typically, the memory footprint does not exceed 100 MB, with EC being the only approach that does not index both data sources, thus requiring significantly lower memory. The run-times exhibit minor differences, remaining far below 10 seconds.

Regarding progressive recall, in five datasets (D_2, D_3, D_5, $D6$ and D_{10}), the BFS solution exhibits the highest values across all budgets, followed in close distance ($\ll$ 2%) by the Hybrid configuration. The EC configuration consistently ranks third, with an average DFT that ranges from 8% (D_{10}) to 16% (D_3). The DFS configuration consistently underperforms all others, typically falling short of the maximum progressive recall by at least 20%.

The remaining datasets verify the very high correlation between BFS and Hybrid, especially during the lower budgets, where they basically yield the same candidate pairs, due to the identical configuration. Both rank second in D_4 and D_7, where EC is the top performer, with DFS ranking last, lower by 1/3, on average. The situation is reversed in D_1 and D_8, where the DFS configuration outperforms all others, leaving BFS and Hybrid in the last place.

Regarding recall, the differences between the four join solutions are consistently much lower than that of progressive recall. In four datasets (D_4 and D_7-D_9), there is actually an insignificant difference between them across all budgets. In the other datasets, BFS takes a clear lead, with Hybrid typically following in close distance, while EC and DFS are usually ranked third and fourth, resp. The only exception is D_1, where DFS and EC are the top performers.

Overall, the best join solution applies BFS to the five most similar candidates per entity according to the Euclidean similarity between the character 5-grams vectors with TF-IDF weights.

7.4 Blocking workflows

The best blocking workflows per scheduling algorithm across all datasets and budgets are reported in Table 6. Note that all weighting schemes rely on the number of blocks shared by two entities. In half the cases, normalization (by size or cardinality) is also required to increase the distinctiveness and the accuracy of the weights.

The progressive recall alongside the recall of these four solutions across all budgets over D_3 and D_7-D_{10} is reported in Figure 6. The performance over the other datasets appears in Figure 16 in the Appendix [20], both the memory requirements and the running time, in Figures 17 and

Scheduling Algorithm	Weighting Scheme
EC	**CN-CBS**
BFS	CBS
DFS	SN-CBS
Hybrid	CBS

Table 6. The best blocking workflows from our design space, with the top performer highlighted in bold.

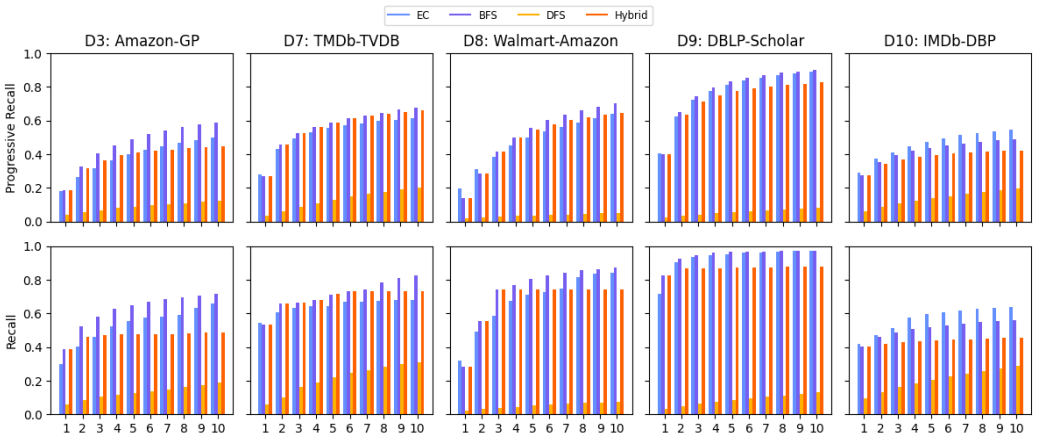


Fig. 6. Progressive recall and recall per budget of the blocking workflows in Table 6 over selected datasets.

18, respectively. There are insignificant differences among the four solutions with respect to the memory footprint, given that they basically differ in the weighting scheme, while their run-time mostly depends on the dataset at hand.

We observe that the DFS configuration consistently underperforms all others to a significant extent. It achieves its best performance in D_1 and D_5, where its progressive recall is lower than the maximum one by 15% and 22%, on average, across all budgets, respectively. In all other datasets, its average *DFT* exceeds 70% – in D_8 and D_9, this distance raises to a whole order of magnitude. Similar patterns pertain to recall, too.

The Hybrid configuration ranks third in most datasets with respect to both evaluation measures. In D_1 and D_5, it exhibits the worst performance among all schedulers, while in D_7 and D_8 it follows the top performer (BFS) in close distance.

There is a strong competition between the remaining solutions, the EC and the BFS one, both of which excel in 5 datasets. The former scores the highest progressive recall across all budgets in D_1, D_2, D_5, D_6 and D_{10}, and the latter in the rest. However, the difference between EC and BFS is much higher in the datasets, where the former ranks first: on average, EC underperforms BFS by less than 2% in D_4 and D_9 as well as by less than 9% in D_7 and D_8, whereas BFS underperforms EC by more than 10% in D_2, D_5 and D_6. Note that these patterns apply to recall and progressive recall.

Overall, EC with the CN-CBS weighting scheme is the overall best blocking solution.

7.5 Sorting-based workflows

The best sorting-based solutions per scheduling algorithm across all datasets and budgets are reported in Table 7. They are all combined with the global scope of functionality. This indicates that considering the combined evidence from multiple windows performs better than the relying on a single window. Note that the largest window size works best for most scheduling algorithms, allowing for a larger difference between the values of two matching entities.

Scheduling Algorithm	Weighting Scheme	Window Size	Functionality Scope
EC	**ID**	**10**	**Global**
BFS	ACF	10	Global
DFS	DICE	1	Global
Hybrid	ACF	10	Global

Table 7. The best sorting-based workflows from our design space, with the top performer highlighted in bold.

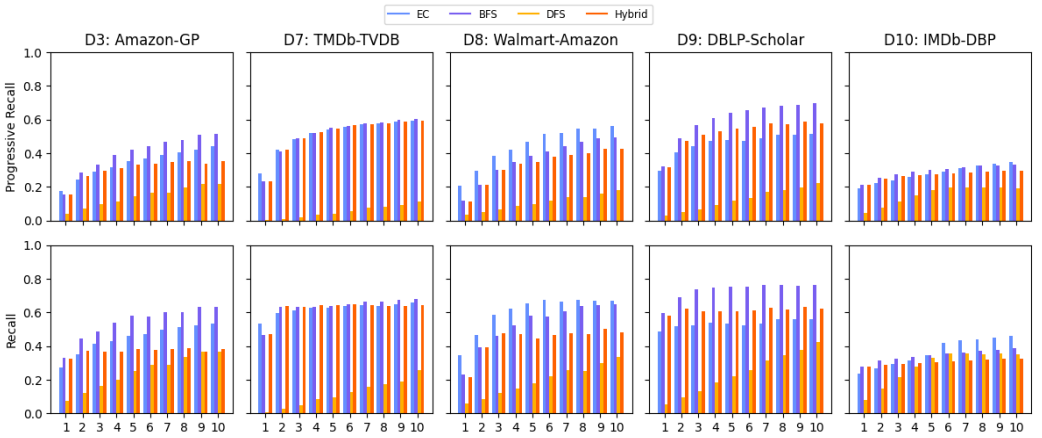


Fig. 7. Progressive recall and recall per budget of the best sorting-based workflows in Table 7 over selected datasets.

The effectiveness of these solutions for all budgets in D_3 and D_7-D_{10} is reported in Figure 7. The effectiveness over the other datasets is reported in Figure 19 in the Appendix [20], while the memory consumption and the run-time per budget and dataset is reported in Figures 20 and 21, respectively. In both cases, the differences are insignificant, due to the consistently low run-time (<10 sec) and memory footprint (<100 MB) – only DFS is slightly more memory efficient, due to the smaller window it uses.

Regarding progressive recall, the DFS configuration consistently underperforms all others to a significant extent, which raises to a whole order of magnitude in D_7. In the best case, in D_5, its progressive recall is lower than the highest one by 1/3, on average, across all budgets. Similarly, for recall, DFS ranks last in 7 datasets, with its average DFT ranging from 15% (D_2) to 83% (D_7).

On the other extreme lie the EC and the BFS solutions, with the former being the top performer in six datasets and the latter in the rest – this applies to both evaluation measures. In terms of progressive recall, EC takes a major lead over BFS in half the datasets, with a progressive recall higher by at least 10%. The only exception is D_7, where their difference is slightly above 1%. In terms of recall, the difference between the two solutions is consistently smaller, but raises above 30% in favor of EC in D_1, D_5 and D_6.

Finally, the behavior of Hybrid depends on the evaluation measure. For progressive recall, it consistently ranks third in all datasets where EC outperforms BFS (i.e., D_1-D_2 and D_5-D_8) – yet, its progressive recall is much higher than that of DFS. In all other datasets, it ranks second, as its performance is highly correlated with BFS. For recall, it performs well only in datasets where all solutions have similar performance, namely D_4 andn D_7. In all other cases, its average DFT ranges from 13% (D_6) to 41% (D_1).

The above patterns apply to each dataset, regardless of the budget. In other words, the relative performance of the four scheduling algorithms is not altered as the size of the budget increases.

We can conclude, therefore, that EC in combination with the ID weighting scheme, a window size of 10 and the global functionality scope constitutes the overall best sorting-based solution.

7.6 Overall best solution

The above experiments demonstrate that for the NN and join workflows, the BFS normally works best in datasets with high levels of noise (i.e., large portion of missing values) and high portion of duplicates for at least one the data sources. This applies to D_2, D_3, D_5, D_6 and D_{10}. In datasets where the matching entities share quite distinctive information, all Scheduling algorithms exhibit high performance, with minor differences between them. This applies to the bibliographic datasets, D_4 and D_{10}, due to the long, distinctive attribute values characterizing each entity (e.g., titles and author lists), as well as to D_7, due to the common movie titles and director or actor names (despite the high levels of noise). In datasets with low portion of duplicates for both data sources, i.e., D_1 and D_8, DFS is the top performer, when configured to search for the nearest neighbor per entity after indexing both data sources.

For the blocking and sorting-based workflows, the BFS and EC consistently outperform the other Scheduling algorithms. The former performs slightly better in datasets with duplicates sharing highly distinctive information (i.e., D_4, D_7 and D_9), whereas EC works best in all other datasets, which exhibit with high levels of noise and/or low portion of duplicates for at least one data source.

In this context, we now compare the best solutions identified in Sections 7.2-7.5, i.e., the BFS approach of the NN and join workflows, with the EC from the blocking and sorting-based workflows. Their progressive recall over the Record Linkage and Deduplication datasets is reported in Figures 8 and 9, respectively.

We observe two different patterns, depending on the type of dataset. For Record Linkage, the sorting-based solution ranks last in all datasets, but the D_1, with its average DFT exceeding 19% in all datasets; the larger the budget, the higher is its DFT, which indicates that only its top-weighted pairs are indeed duplicates.

In contrast, the join solution outperforms all others to a statistically significant extend in seven datasets (D_2-D_3, D_5-D_8 and D_{10}). It is actually the top performer across all budgets in all these datasets, except for the smallest two in D_6 and D_7. Its performance remains very high in the remaining datasets, too: in the bibliographic datasets (D_4 and D_9), its difference from the top performer (the NN solution) is statistically insignificant, while in D_1, it takes the lead over the blocking solution for the three largest budgets.

The situation is reversed in most Deduplication datasets, except for the smallest one (De_1), where we observe the same patterns as in Record Linkage. In De_2 and De_3, the blocking solution takes the lead, followed in close distance by the sorting-based one, with the join workflow ranking third and fourth, respectively. However, in the five largest workflows, the sorting-based solution consistently ranks first, with a major lead over the remaining solutions. Note that the join workflow does not scale beyond 50,000 entities (i.e., De_5), due to insufficient memory.

The run-time measurements across all budgets are presented in the Appendix [20], in Figures 23 and 25 for the Record Linkage and Deduplication datasets, respectively. The sorting-based solution is consistently the fastest approach across all budgets, with the blocking following in close distance. The NN and join solutions are slower by 1 or 2 orders of magnitude across all datasets. In the smallest datasets, NN is slower than join, but the latter becomes much slower as the number of input entities increases. This means that the join solution exhibits poor scalability, unlike the NN one. Note that for all solutions, the size of the budget does not affect the run-time, as Filtering and Weighting are independent of the budget. Only Scheduling is affected, albeit to a minor extent.

This significant difference in the scalability of the NN and join workflows should be attributed to the relative cost of their vectorization, indexing and querying phases. For the former, vectorization is typically a time-consuming process, due to the high dimensionality and the high number of parameters of S-GTR-T5 [56]. In contrast, indexing and querying is quite efficient, due to FAISS, a the state-of-the-art tool for approximate nearest neighbor search [1]. In contrast, the join workflows involve very efficient vectorization and indexing phases, with the querying one constituting the bottleneck, as it aggregates the posting lists of all tokens associated with each query entity; the number of tokens is high, due to the schema-agnostic settings, which consider all attribute values.

The memory footprints are presented in Figures 22 and 24 of the Appendix [20] for the Record Linkage and the Deduplication datasets, respectively. In most cases, the memory footprint of the NN workflows is higher by an order of magnitude than the other Filtering types, because the former leverages very high dimensional embedding vectors, while the latter operate directly on string values (hence, they depend heavily on the dataset size). Note, though, that the join solution does not scale to more than 50,000, due to a two-dimensional matrix that lies at the core of its implementation that depends on the size of the input dataset.

Overall, we can conclude that for the Record Linkage datasets, the top join workflow constitutes the best solution, trading the highest accuracy for the lowest scalability in terms time and memory efficiency. In contrast, the best NN workflow favors scalability over accuracy. For the Deduplication datasets, the best sorting-based workflow outperforms all others in terms of accuracy, run-time and memory efficiency.

7.7 Comparison to the state-of-the-art

To assess the performance of the join and NN workflows, we consider two recent state-of-the-art, open-source filtering approaches:

(1) DeepBlocker [50] is an NN workflow that combines FAISS for indexing and querying with self-supervised learning for improving the pre-trained embedding vectors provided by FAISS.
(2) Sparkly [35] is a join workflow that combines TF-IDF weights and BM25 scores with parallelization on top of Apache Spark.

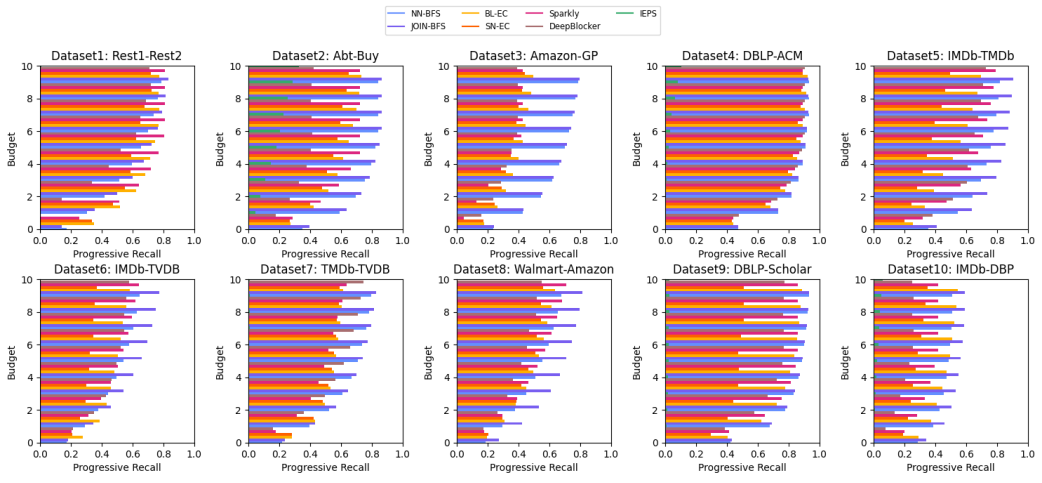


Fig. 8. Progressive recall for the best progressive method per Filtering type and the baseline methods over the Record Linkage datasets in Table 1.

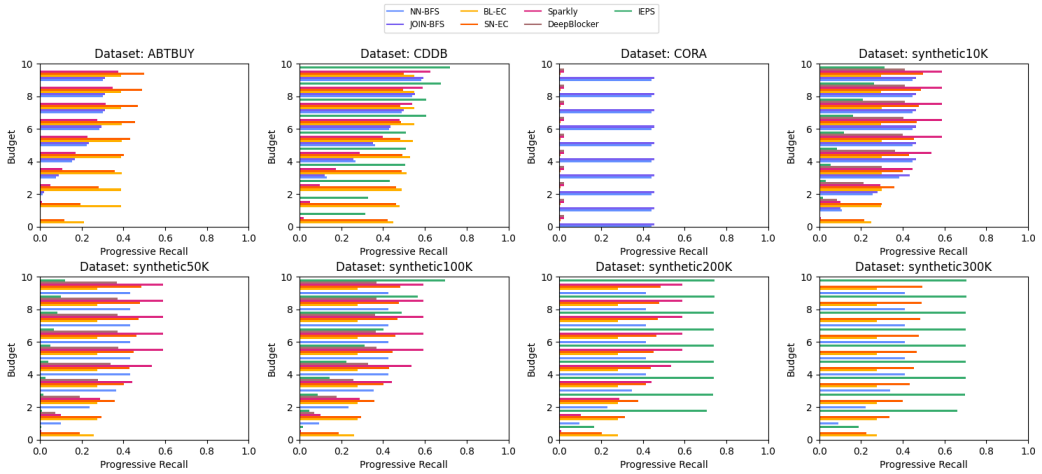


Fig. 9. Progressive recall for the best progressive method per Filtering type and the baseline methods over the Deduplication datasets in Table 2.

For their fine-tuning, we performed grid search over the two common parameters (see Table 3):

(1) the number of nearest neighbors $k \in [1, 5, 10]$
(2) the indexing scheme, which can be $\{smallest, largest, both\}$

For both methods, the best performance, which minimizes the average DFT, across all budgets and datasets, corresponds to $k = 5$. Regarding the indexing scheme, DeepBlocker works best when indexing the smallest dataset and querying with the largest one and vice versa for Sparkly. Note that DeepBlocker uses the cosine similarity by default and that its operation is stochastic, due to the random selection of instances to be labelled to form the automatically-created training set during self-supervised learning. As a result, in each dataset and budget, we consider its average performance over 5 iterations. Note also that Sparkly is combined with the character 3-grams that optimize its performance, as shown in [35].

As an additional baseline approach, we use the best workflow generated by the Progressive Incremental Entity Resolution framework, namely I-PES [11]. All parameters were fine-tuned according to the experimental analysis in [11]. The available implementation, though, estimates only recall, not the progressive one.

These three baseline methods are compared with the best join and sorting-based solutions with respect to Progressive Recall over the Record Linkage and the Deduplication datasets in Figures 8 and 9, respectively. Starting with the former, we observe that the join solution achieves the highest progressive recall for practically all budgets in all datasets, but the smallest one, where Sparkly takes the lead. In all other datasets, Sparkly and DeepBlocker underperform the join solution to a significant extent, with an average DFT usually higher than 20%. Even the sorting-based solution outperforms both Sparkly and DeepBlocker in D_3.

In the Deduplication datasets, our techniques outperform again the baseline methods to a significant extent. In the smallest dataset (De_1), the join workflow is the top performer, while in all others, the sorting-based solution takes the lead. Note that DeepBlocker scales up to De_6, which involves 100,000 entities, due to a two-dimensional matrix that lies at its core (similar to the join solution). The memory efficiency of these methods is presented in Figures 22 and 24. We observe that the sorting-based workflow consistently exhibits the lowest memory footprint, even by a whole

order of magnitude, especially when compared to DeepBlocker. The join solution is more memory efficient than the baselines in all Record Linkage datasets, where it achieves the highest accuracy.

For what concerns time efficiency, the join (and the sorting-based) solutions are consistently faster than the baseline methods over the Record Linkage datasets (Figure 23). Over the Deduplication datasets (Figure 25), the sorting-based solution is by far the fastest up to dataset De_4, which contains 10,000 entities. In larger datasets, the overhead of PySpark parallelization pays off and, thus, Sparkly becomes the fastest approach.

Overall, our solutions consistently outperform the baselines with respect to effectiveness in practically all budgets of all considered datasets (except for D_1, where Sparkly takes the lead). They are also quite time and memory efficient, but Sparkly excels in scalability, as it runs on PySpark.

8 CONCLUSIONS

We presented an architecture template for generating a wide diversity of Progressive Entity Matching solutions based on three modules that precede the matching and the clustering algorithms in an end-to-end ER pipeline. Through a thorough experimental analysis, we identified the four top performing solutions, one for each combination of filtering and weighting techniques. Our solutions consistently outperform the current state-of-the-art in terms of progressive recall, memory footprint and often run-time.

In the future, we will adapt all four Filtering types to real-time ER, where the goal is to match a query record in sub-second time [38]. For the NN and join workflows, this is a natural setting that requires fine-tuning their index. Pre-calculated similarities can be used for blocking workflows when the query record has already been indexed [4]. If not, Algorithm 2 should be adapted to work with the three indexes of DySimII, whose record insertion and query times remain practically stable, despite the increasing number of entities [40]. Finally, Algorithm 3 should integrate the braided AVL trees used by F-DySNI [39], which optimize the record insertion and query times of the sorting-based workflows.

ACKNOWLEDGMENTS

This work was partially funded by the EU project STELAR (Horizon Europe – Grant No. 101070122).

REFERENCES

[1] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020).
[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* abs/1607.04606 (2016).
[3] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.
[4] Peter Christen, Ross W. Gayler, and David Hawking. 2009. Similarity-aware indexing for real-time entity resolution. In *CIKM*. 1565–1568.
[5] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2021. An Overview of End-to-End Entity Resolution for Big Data. *ACM Comput. Surv.* 53, 6 (2021), 127:1–127:42.
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[7] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration.* Morgan & Claypool Publishers.
[8] Ju Fan, Jianhong Tu, Guoliang Li, Peng Wang, Xiaoyong Du, Xiaofeng Jia, Song Gao, and Nan Tang. 2024. Unicorn: A Unified Multi-Tasking Matching Model. *SIGMOD Rec.* 53, 1 (2024), 44–53.
[9] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2021. BEER: Blocking for Effective Entity Resolution. In *SIGMOD*. 2711–2715.
[10] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2021. Efficient and effective ER with progressive blocking. *VLDB J.* 30, 4 (2021), 537–557.
[11] Leonardo Gazzarri and Melanie Herschel. 2023. Progressive Entity Resolution over Incremental Data. In *EDBT*. 80–91.

[12] Bar Genossar, Avigdor Gal, and Roee Shraga. 2023. The Battleship Approach to the Low Resource Entity Matching Problem. *Proc. ACM Manag. Data* 1, 4 (2023), 224:1–224:25.
[13] Lise Getoor and Ashwin Machanavajjhala. 2012. Entity Resolution: Theory, Practice & Open Challenges. *PVLDB* 5, 12 (2012), 2018–2019.
[14] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB* 2, 1 (2009), 1282–1293.
[15] Arjit Jain, Sunita Sarawagi, and Prithviraj Sen. 2021. Deep Indexed Active Learning for Matching Heterogeneous Entity Representations. *PVLDB* 15, 1 (2021), 31–45.
[16] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.
[17] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *PVLDB* 3, 1 (2010), 484–493.
[18] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR*.
[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019).
[20] Jakub Maciejewski, Konstantinos Nikoletos, George Papadakis, and Yannis Velegrakis. 2022. Progressive Entity Matching: A Design Space Exploration. https://github.com/JacobMaciejewski/PER-Design-Space-Exploration/paper/PEMextended.pdf. In *SIGMOD (extended version)*.
[21] Venkata Vamsikrishna Meduri, Lucian Popa, Prithviraj Sen, and Mohamed Sarwat. 2020. A Comprehensive Benchmark Framework for Active Learning Methods in Entity Matching. In *SIGMOD*. 1133–1147.
[22] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.
[23] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*. 3111–3119.
[24] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. ACM, 19–34.
[25] Franziska Neuhof, Marco Fisichella, George Papadakis, Konstantinos Nikoletos, Nikolaus Augsten, Wolfgang Nejdl, and Manolis Koubarakis. 2024. Open benchmark for filtering techniques in entity resolution. *The VLDB Journal* (2024), 1–26.
[26] Daniel Obraczka, Jonathan Schuchart, and Erhard Rahm. 2021. Embedding-Assisted Entity Resolution for Knowledge Graphs. In *KGCW@ESWC*.
[27] George Papadakis, Vasilis Efthymiou, Emmanouil Thanos, Oktie Hassanzadeh, and Peter Christen. 2023. An analysis of one-to-one matching algorithms for entity resolution. *VLDB J.* 32, 6 (2023), 1369–1400.
[28] George Papadakis, Marco Fisichella, Franziska Schoger, George Mandilaras, Nikolaus Augsten, and Wolfgang Nejdl. 2023. Benchmarking Filtering Techniques for Entity Resolution. In *ICDE*. IEEE, 653–666.
[29] George Papadakis, Ekaterini Ioannou, Emanouil Thanos, and Themis Palpanas. 2021. *The Four Generations of Entity Resolution*. Morgan & Claypool Publishers.
[30] George Papadakis, Nishadi Kirielle, Peter Christen, and Themis Palpanas. 2024. A Critical Re-evaluation of Record Linkage Benchmarks for Learning-Based Matching Algorithms. In *ICDE*. 3435–3448.
[31] George Papadakis, Georgios M. Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas, and Manolis Koubarakis. 2020. Three-dimensional Entity Resolution with JedAI. *Inf. Syst.* 93 (2020), 101565.
[32] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2021. Blocking and Filtering Techniques for Entity Resolution: A Survey. *ACM Comput. Surv.* 53, 2 (2021), 31:1–31:42.
[33] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *PVLDB* 9, 9 (2016), 684–695.
[34] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. 2015. Progressive Duplicate Detection. *TKDE* 27, 5 (2015), 1316–1329.
[35] Derek Paulsen, Yash Govind, and AnHai Doan. 2023. Sparkly: A Simple yet Surprisingly Strong TF/IDF Blocker for Entity Matching. *PVLDB* 16, 6 (2023), 1507–1519.
[36] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.
[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn.*

Res. 21 (2020), 140:1–140:67.
[38] Banda Ramadan and Peter Christen. 2015. Unsupervised Blocking Key Selection for Real-Time Entity Resolution. In *PAKDD*. 574–585.
[39] Banda Ramadan, Peter Christen, Huizhi Liang, and Ross W. Gayler. 2015. Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution. *ACM J. Data Inf. Qual.* 6, 4 (2015), 15:1–15:29.
[40] Banda Ramadan, Peter Christen, Huizhi Liang, Ross W. Gayler, and David Hawking. 2013. Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution. In *PAKDD*. 47–58.
[41] Alieh Saeedi, Markus Nentwig, Eric Peukert, and Erhard Rahm. 2018. Scalable Matching and Clustering of Entities with FAMER. *Complex Syst. Informatics Model. Q.* 16 (2018), 61–83.
[42] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR* abs/1910.01108 (2019).
[43] Sunita Sarawagi and Anuradha Bhamidipaty. 2002. Interactive deduplication using active learning. In *KDD*. ACM, 269–278.
[44] Pavel Shvaiko, Jérôme Euzenat, Fausto Giunchiglia, Heiner Stuckenschmidt, Ming Mao, and Isabel F. Cruz (Eds.). 2010. *Proceedings of the 5th International Workshop on Ontology Matching (OM-2010), Shanghai, China, November 7, 2010*. Vol. 689.
[45] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *TKDE* 31, 6 (2019), 1208–1221.
[46] Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, and Felix Naumann. 2022. Entity Resolution On-Demand. *PVLDB* 15, 7 (2022), 1506–1518.
[47] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-training for Language Understanding. In *NIPS*.
[48] Chenchen Sun, Zhijiang Hou, Derong Shen, and Tiezheng Nie. 2022. Progressive Entity Matching via Cost Benefit Analysis. *IEEE Access* 10 (2022), 3979–3989.
[49] Chenchen Sun, Yuyuan Jin, Yang Xu, Derong Shen, Tiezheng Nie, and Xite Wang. 2023. Exploring the Design Space of Unsupervised Blocking with Pre-trained Language Models in Entity Resolution. In *ADMA*, Vol. 14176. 228–244.
[50] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *PVLDB* 14, 11 (2021), 2459–2472.
[51] Norases Vesdapunt, Kedar Bellare, and Nilesh N. Dalvi. 2014. Crowdsourcing Algorithms for Entity Resolution. *PVLDB* 7, 12 (2014), 1071–1082.
[52] Runhui Wang and Yongfeng Zhang. 2024. Pre-trained Language Models for Entity Blocking: A Reproducibility Study. In *NAACL*. 8712–8722.
[53] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. In *NIPS*.
[54] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. 2013. Pay-As-You-Go Entity Resolution. *TKDE* 25, 5 (2013), 1111–1124.
[55] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *NeurIPS*. 5754–5764.
[56] Alexandros Zeakis, George Papadakis, Dimitrios Skoutas, and Manolis Koubarakis. 2023. Pre-trained Embeddings for Entity Resolution: An Experimental Analysis. *PVLDB* 16, 9 (2023), 2225–2238.
[57] Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi, and Felix Naumann. 2023. BrewER: Entity Resolution On-Demand. *PVLDB* 16, 12 (2023), 4026–4029.

Received July 2024; revised September 2024; accepted November 2024