Progressive Entity Resolution with Node Embeddings

Giovanni Simonini, Luca Gagliardelli, Michele Rinaldi[†], Luca Zecchini,
Giulio De Sabbata, Adeel Aslam, Domenico Beneventano and Sonia Bergamaschi

Università degli Studi di Modena e Reggio Emilia, Modena, Italy
<name.surname>@unimore.it
[†] *260345@studenti.unimore.it*

Abstract

Entity Resolution (ER) is the task of finding records that refer to the same real-world entity, which are called matches. ER is a fundamental pre-processing step when dealing with dirty and/or heterogeneous datasets; however, it can be very time-consuming when employing complex machine learning models to detect matches, as state-of-the-art ER methods do. Thus, when time is a critical component and having a partial ER result is better than having no result at all, progressive ER methods are employed to try to maximize the number of detected matches as a function of time.

In this paper, we study how to perform progressive ER by exploiting graph embeddings. The basic idea is to represent candidate matches in a graph: each node is a record and each edge is a possible comparison to check—we build that on top of a well-known, established graph-based ER framework. We experimentally show that our method performs better than existing state-of-the-art progressive ER methods on real-world benchmark datasets.

Keywords

Entity Resolution, Pay-as-you-go, Data Cleaning, Graph Embedding

1. Introduction

Entity Resolution (ER) is the task of identifying records (a.k.a. profiles) that refer to the same real-world entity in datasets [1]. It is a fundamental task for preparing dirty data to avoid duplicates, which lead to inconsistencies and may compromise the downstream analysis [2]. It is also the only viable way to join tables in the absence of foreign/primary key constraints.

The naïve solution of comparing each and every pair of records to determine whether they belong to the same entity (i.e., they *match*) has a quadratic complexity; hence, it is impractical with large datasets. To mitigate this problem, *blocking* is typically employed to partition the considered data into *blocks* and to perform the all-pairs comparison only within each block. Blocking is usually achieved by extracting from each record one or more *blocking keys* that are used to index that record into specific blocks. For instance, given a dataset with client contacts, a simple blocking strategy could be to index together records that share some attribute values, such as the client name or the phone number.

Yet, a client may use a nickname (e.g., William/Bill) and have multiple phone numbers (e.g., home/work). Thus, to define effective blocking key extraction strategies that discard many

SEBD 2022: The 30th Italian Symposium on Advanced Database Systems, June 19-22, 2022, Tirrenia (PI), Italy

0000-0002-3466-509X (G. Simonini); 0000-0001-5977-1078 (L. Gagliardelli); 0000-0002-4856-0838 (L. Zecchini); 0000-0001-6616-1753 (D. Beneventano); 0000-0001-8087-6587 (S. Bergamaschi)

CEUR Workshop Proceedings (CEUR-WS.org)

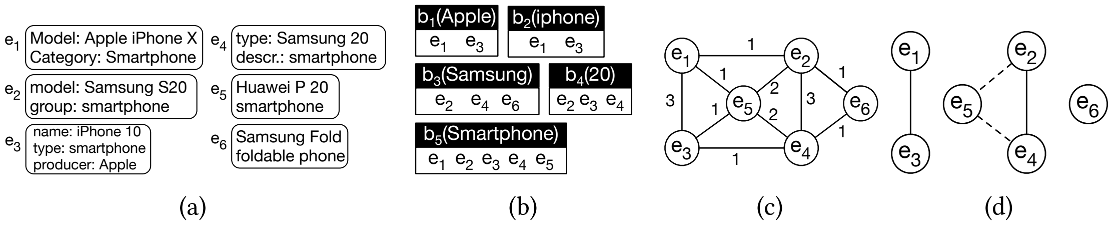


Figure 1: Meta-blocking example.

superfluous comparisons while correctly indexing together matching records is a hard task. This task becomes even harder when considering heterogeneous datasets, since aligning the attributes can be prohibitive when dealing with Web data or data lakes. In such a scenario, the state-of-the-art solution is to employ *meta-blocking* [3, 4].

The basic idea of meta-blocking is to extract schema-agnostic blocking keys (i.e., by considering the record as a unique, concatenated snippet of text, discarding its schema) and then to restructure the yielded blocks to improve the efficiency. The former step allows to deal with schema heterogeneity and to achieve a high level of recall. For instance, by employing *Token Blocking*, each token appearing in a record (regardless of the attribute in which it appears) is used as a blocking key. Thus, on one hand, there is no need to perform schema alignment beforehand if the data is messy (e.g., in an e-commerce dataset, sony appears in the Brand attribute in a record and in the Title attribute in another one); on the other hand, the number of generated blocking keys (hence comparisons) can be very high, slowing the process and yielding low precision. To increase precision, in the meta-blocking step, the blocks are restructured by representing them in the following way: each record is represented as a node in a graph, called *blocking graph*; an edge is present between two nodes if the two corresponding records appear together in at least one block (i.e., they share at least one blocking key); each edge is weighted according to the number and the characteristics (e.g., the size) of the blocks that the two adjacent nodes share; finally, a graph-pruning algorithm is applied and the remaining edges represent the final candidate pairs of matching records.

Figure 1a reports an example of a record collection gathered from different data sources. The schema of each record may be different from the others—a common trait of data collected from the Web. Token Blocking is applied to the record collection: each token in the record is a blocking key, which means that the record is indexed in a block with all the records in which that token appears. Without performing schema alignment, the only viable way is to consider each blocking key (i.e., token) regardless of the attribute in which it appears.

The result is the blocking collection of Figure 1b. On one hand, in these blocks, all records are indexed together in at least one block—in other words, all matching pairs will be checked. On the other hand, many comparisons are superfluous, since this strategy does not filter many pairs; further, many pairs appear multiple times across the blocks (for records that share multiple tokens). At the end of the day, the high capability of such a schema-agnostic blocking strategy to find all matching pairs is traded for a higher number of comparisons to perform, which is a burden for the ER process.

A solution is to employ meta-blocking. All the records of Figure 1b are represented as nodes in a graph and the edges of the graph are weighted accordingly to the co-occurrences in the blocks of their adjacent nodes (Figure 1c). This blocking graph is processed with graph pruning algorithms to retain only promising edges (i.e., comparisons). Figure 1d reports as an example the output of a pruning algorithm that takes as a threshold for each node the average of the weights of its edges, then applies that local threshold to each node neighborhood.

Such an approach has two main advantages: the first is that all redundant comparisons are inherently removed (we can have only one edge between two nodes); the second is that shifting to a graph representation (from blocks) allows to implement *progressive* strategies (as explained below) and exploit the findings of the research community about *graph embedding*, which is explored in this work.

Progressive Entity Resolution with Meta-blocking. Meta-blocking was designed to support *batch* ER [5], i.e., to generate a set of candidate pairs that are evaluated in batch, without a specific order. Thus, meta-blocking per se is not designed to maximize the emission of likely-to-match pairs of records, but in our previous work [6] we demonstrated how to exploit a blocking graph to perform ER progressively.

In particular, the best performing method is *Progressive Profile Scheduling*, denoted as PPS. The basic idea of PPS is to store all top-1 comparisons per node in the blocking graph in a heap data structure. Then, PPS attractively emits the comparisons until the heap is empty. While populating the main heap of comparisons mentioned above, PPS also maintains an auxiliary heap where it inserts all the nodes of the graph, weighted accordingly to an aggregate measure of their adjacent edges. For instance, a node has a weight equal to the average of the weights of its adjacent edges in the blocking graph. This auxiliary heap is employed to attractively retrieve at a time the node that is most likely to have a match. Then, while the primary heap emits comparisons, a parallel process can generate new comparisons from the current node extracted from the auxiliary heap and fill the primary one. In this way, the memory footprint of the algorithm is $\mathcal{O}(n)$, where n is the number of records.

Our Contribution. Our intuition is to represent each node of the blocking graph as an embedding by exploiting recent advances in the field [7, 8, 9]. By doing so, each node is represented through a vector that embeds its characteristics taking into account both the structure of the sub-graph in which it appears and the weights of the edges of that sub-graph. Thus, more information can be exploited to select the right candidates compared to the traditional methods based on the blocking graph, which only consider a node and its adjacent edges at a time—analyzing sub-graphs would be too expensive.

In Section 2, we provide the notation, the definition of the problem, and the basic concepts behind graph embedding, which are employed to present our method in Section 3. The experimental demonstration of the efficacy of our method is reported in Section 4, which shows how our method outperforms the state-of-the-art graph-based PPS method. Further related work and conclusions (which also introduce future directions for this research) are presented in Sections 5 and 6, respectively.

2. Preliminaries

In a dirty dataset $\mathcal{D}$, two records that refer to the same real-world entity are defined as *matching records*; we denote this with $r_i \equiv r_j$. As other state-of-the-art ER frameworks [3, 10, 11], we rely on *blocking* for scaling *pairwise matching*, which is the actual task to determine whether two records are matching or not.

In the first phase, blocking is applied and a set $\mathcal{B}$ of possibly overlapping clusters of records (called *blocks*) is generated. All the pairs of records appearing within a block are *candidate matches*, denoted with $\langle r_i, r_j \rangle$.

In the second phase, a binary matching function is applied to all candidate pairs entailed by $\mathcal{B}$: the matching function $\mu : \mathcal{D} \times \mathcal{D} \rightarrow \{\texttt{true}, \texttt{false}\}$ takes as input two records and returns `true` in case they match, `false` otherwise. The matching function can be a trained machine/deep learning model [10, 11, 12], a manually-designed function [13], or a human oracle in a crowdsourcing setup [14]. The output of this phase is a set of matching pairs, which are clustered to avoid ambiguities (e.g., by applying a transitive clustering)—each cluster of matches represents an entity in the real world.

2.1. Graph-based Meta-blocking

In *graph-based meta-blocking* (*meta-blocking* for simplicity), a collection of blocks $\mathcal{B}$ is represented by a weighted graph $\mathcal{G}_{\mathcal{B}}\{V_{\mathcal{B}}, E_{\mathcal{B}}, \mathcal{W}_{\mathcal{B}}\}$ called *blocking graph.* V is the set of nodes representing all records $r_i \in \mathcal{D}$. An edge between two records exists if they appear together in at least one block; thus, $E = \{e_{ij} : \exists r_i, r_j \in \mathcal{D} \mid |\mathcal{B}_{ij}| > 0\}$ is the set of edges, with $\mathcal{B}_{ij} = \mathcal{B}_i \cap \mathcal{B}_j$, where $\mathcal{B}_i$ and $\mathcal{B}_j$ are the sets of blocks containing r_i and r_j respectively. $\mathcal{W}_{\mathcal{B}}$ is the set of edge weights. Meta-blocking tries to capture the *matching likelihood* of two records in the weight of the edge that connects them. For instance, *Block Co-occurrence Frequency* (CBS) [3] assigns to the edge between two records r_u and r_v a weight equal to the number of blocks they share, i.e., $w_{uv}^{CBS} = |\mathcal{B}_u \cap \mathcal{B}_v|$.

Then, in order to keep only the most promising edges, we can apply suitable edge-pruning strategies. In this way, after this pruning step, each connected pair of nodes forms a new block of the restructured blocking collection.

Meta-blocking can operate by keeping all the candidate comparisons that are weighted above a certain threshold or in a top-*k* fashion. We call the first case *Weighted Pruning*, while *Cardinality Pruning* the latter. The weight threshold, or the k for the top-*k* approach, can be defined at local level (i.e., for each record/node in the graph) or at global level (i.e., for all the edges in the graph). Hence, the combination of those strategies yields the following *pruning strategies*: *(i) Weighted Edge Pruning*, where edges with a weight lower than the given threshold are pruned; *(ii) Cardinality Edge Pruning*, where edges are sorted in descending order with respect to their weights, and then only the first k are kept; *(iii) Weighted Node Pruning*, which considers in turn each node r_i and its adjacent edges, and edges with a weight lower than the given threshold are pruned; *(iv) Cardinality Node Pruning*, which similarly to the previous one is node-centric, but where a cardinality threshold k_i is used instead of a weight threshold.

Finally, the state-of-the-art progressive ER method based on the blocking graph [3], instead of applying pruning, tries to prioritize the edges according to their weight—i.e., to prioritize the comparisons in order to maximize the recall as a function of the number of comparisons.

3. Progressive Meta-blocking with Node Embeddings

A limitation of the existing blocking-graph-based methods is that for each comparison likelihood is estimated by analyzing only the two adjacent nodes that its corresponding edge connects. Thus, for instance, if a set of nodes is part of a clique, this information is not captured just by looking at two nodes at a time. Generally, graph patterns that involve more than two nodes are not captured. Our intuition is that such an information could be exploited. Yet, we do not want to count triangles, cliques of unknown size, etc. The main reasons are the following: firstly, it would be computationally expensive; secondly, we do not know which structure could be useful; finally, it would be difficult to take into account all information—e.g., should we count the cliques? Should we assign weights to them? Should we also give importance to quasi-cliques?

An effective solution is to exploit *node embeddings* [9]. This allows to bypass the detection of such graph patterns thanks to techniques that capture latent information about the graph structure and assign a set of weights to each node according to that information.

The basic idea is to measure somehow the similarity between two nodes in the network and compare it to the similarity in a vector space (the *embedding space*). In particular, we focus on unsupervised approaches based on random walks. These methods use a walk approach to generate (sample) network neighborhoods for nodes. For every node, they generate its network neighborhood by choosing in some way (e.g., randomly following the edges) the next node of the walk, until a certain number of steps (the *walk sampling*) is reached. This iterative process associates nodes that appear in a path together since they are considered more similar. Then, to compute the representative vector embeddings, the nodes and their co-occurrences in the random paths are given as an input to a shallow two-layers neural network, which extracts the vector embedding for each node as in the skip-gram model [15].

3.1. Progressiveness with Node Vectors

Once that the vector embeddings have been generated for all the nodes, we employ LSH to build bands of decreasingly similar pairs of nodes. Each band can be dynamically computed; thus, for instance, we start by considering pairs with a cosine similarity greater than 0.9 (i.e., between 1 and 0.9). Then, in the second iteration, we consider pairs with a similarity greater than 0.8 and remove from that set the pairs already considered in the first iteration, and so forth—for lower similarly thresholds.

When considering each level, we actually compute the similarity of each candidate pair and insert it in a heap. Then, we start the emission of the pairs in non-increasing weight order.

4. Experiments

Hardware and Software. All experiments were performed on a machine equipped with four Intel Xeon E5-2697 2.40 GHz (72 cores), 216 GB of RAM, running Ubuntu 18.04. We employed the *SparkER* library [17] to perform Token Blocking and the blocking graph generation. All code is implemented in Python 3.7.

Dataset	$\|\mathbf{D_1}\|$	$\|\mathbf{D_2}\|$	$\|\mathcal{M}\|$	$\|\mathbf{C}\|$	Recall	Precision	$F1$
DblpAcm	2.6k	2.3k	2.2k	46.2k	0.999	$4.81{\cdot}10^{-2}$	$9.18{\cdot}10^{-2}$
ScholarDblp	2.5k	61.3k	2.3k	832.7k	0.998	$2.80{\cdot}10^{-3}$	$5.58{\cdot}10^{-3}$
Movies	27.6k	23.1k	22.8k	26.0M	0.976	$8.59{\cdot}10^{-4}$	$1.72{\cdot}10^{-3}$
ImdbTmdb	5.1k	6.0k	1.9k	109.4k	0.988	$1.78{\cdot}10^{-2}$	$3.50{\cdot}10^{-2}$
ImdbTvdb	5.1k	7.8k	1.1k	119.1k	0.985	$8.90{\cdot}10^{-3}$	$1.76{\cdot}10^{-2}$
TmdbTvdb	6.0k	7.8k	1.1k	198.6k	0.989	$5.50{\cdot}10^{-3}$	$1.09{\cdot}10^{-2}$

Table 1
The datasets used in the experimental study.

Measures. We employ recall and precision, defined as follows:

$$Recall = \frac{\#\{\textit{matching pairs indexed in blocks}\}}{\#\{\textit{existing matching pairs}\}} \qquad Precision = \frac{\#\{\textit{matching pairs indexed in blocks}\}}{\#\{\textit{pairs indexed in blocks}\}}$$

and their harmonic mean (i.e., F-score) in order to refer to the overall performance of a blocking strategy—with the assumption that enough time is given to complete the entire ER process. Instead, to assess the progressiveness, we plot the recall as a function of the comparisons yielded by a progressive method.

Datasets. Table 1 lists the 6 real-world datasets employed in our experiments: $|D_x|$ stands for the number of records in a data source, $|\mathcal{M}|$ for the number of matching pairs, $|C|$ for the number of candidate pairs yielded by schema-agnostic Token Blocking [3]—the block collection generated with Token Blocking is the input for our methods, as for standard meta-blocking and PPS [6]. They have different characteristics and cover a variety of domains. Each dataset involves two different, but overlapping data sources, where the ground truth of the real matches is known. `DblpAcm` matches scientific articles extracted from dblp.org and dl.acm.org [18]. `ScholarDblp` matches scientific articles extracted from scholar.google.com and dblp.org [18]. `Movies` matches information about films that are extracted from imdb.com and dbpedia.org [3]. `ImdbTmdb`, `ImdbTvdb`, and `TmdbTvdb` match movies and TV series extracted from IMDB, TheMovieDB and TheTVDB [19], as suggested by their names.

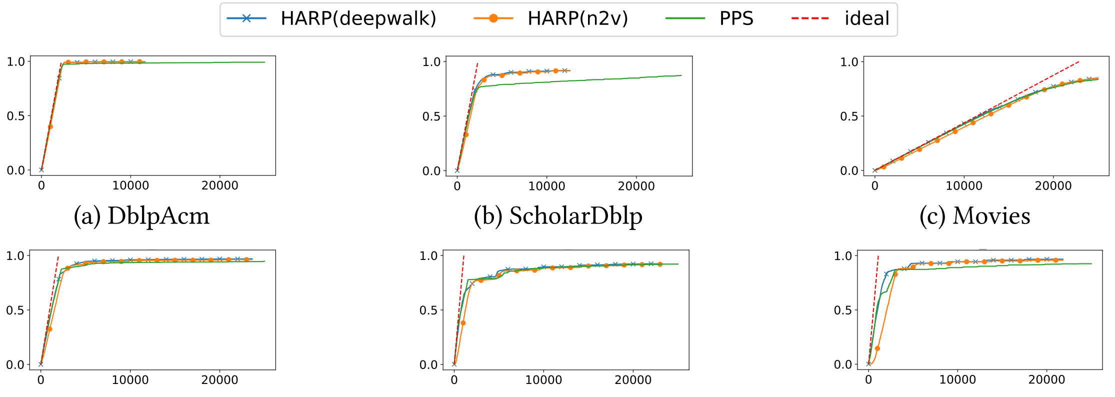
Algorithms and baseline. We rely on HARP [9] for generating the graph embeddings[1]. In particular, we employed both *DeepWalk* [7] and *node2vec* [8] with the following configuration, which we found to work well on all considered datasets. For each node, we consider its neighborhood composed of nodes that can be reached within 10 and 40 hops for generating the random walks. For generating the vector embeddings, we consider a window size of 2 to feed the skip-gram model (i.e., the shallow neural network that is used to yield the embeddings). It is worth to notice that by varying these parameters, even significantly (e.g., 20 hops, 80 random walks, and a window of size 5), we basically obtain the same results reported here, but with a significantly higher runtime.

As a baseline, we employed PPS [6], which, to the best of our knowledge, is the state-of-the-art schema-agnostic and unsupervised progressive ER algorithm.

Results. Figure 2 reports the progressive recall of the considered methods: our proposal based on HARP and the baseline (i.e., PPS). The ideal method, i.e., the method that ideally emits only matching pairs, is depicted with a red, dashed line.

On `ScholarDblp` (Figure 2b), HARP has a steeper recall curve than PPS after the recall reaches 0.8 (they perform basically the same before that). On `TmdbTvdb` (Figure 2f), HARP has

[1] https://github.com/GTmac/HARP



(a) DblpAcm (b) ScholarDblp (c) Movies

(d) ImdbTmdb (e) ImdbTvdb (f) TmdbTvdb

Figure 2: Progressive recall (i.e., recall per number of performed comparisons).

a steeper recall curve than PPS only when employing *DeepWalk* [7] for a recall lower than 0.8, while it always performs better for a recall greater than 0.8. With all other datasets, HARP and PPS behave almost identically. However, we report that HARP can reach a final recall (i.e., if the process ends) that is higher than the one of PPS of 2% on average. Hence, HARP seems to be a better choice if the ER process might have enough time to complete.

As for the runtime, HARP is between 3 and 10 times slower that PPS. Yet, this is typically not a limiting problem: the entity matching functions employed in real-world scenarios, based on machine/deep learning models [11, 12], are at least two orders of magnitude slower than the average time of emission per pair of both HARP and PPS—in other words, the overhead of HARP is negligible compared to the time that the actual matching requires.

5. Related Work

A plethora of approaches and algorithms have been proposed for the problem of Entity Resolution (ER) [3, 20, 21]; please refer to [1] for a complete survey. We identify two main research challenges about ER in the literature: *(i)* how to devise a matching function that given two records identifies whether they match or not [10]; *(ii)* how to scale such a matching function. In this work we focus only on the second challenge.

Scaling ER. Schema-agnostic blocking [3] has been proposed to scale ER without the burden of dealing with schema heterogeneity, when operating with big data sources. *Batch* blocking methods [3, 10] aim at maximizing the recall of a set of candidate pairs, while maintaining a high level of precision. The output is an unordered set of pairs with no priority; thus, if the ER process has to be early terminated due to lack of time and/or resources, only a small portion of the matches would be found. Differently, *progressive* methods [6, 22, 23] aim at prioritizing candidate pairs, i.e., to emit candidates that are actually matches as soon as possible. Thus, in case of early termination, the amount of detected matches is maximized. The state-of-the-art method for schema-agnostic progressive ER is PPS [6]; we compare against it in Section 4.

Graph Embeddings for ER. Graph embeddings [24, 25] have been employed for designing entity matching algorithms (i.e., to match pairs of records) while dealing with schema heterogeneity (i.e., in a schema-agnostic fashion). Yet, here we study a complementary problem, since [24] tackles neither the problem of blocking nor the one of progressive ER.

6. Conclusions and Future Work

By employing the well-known blocking graph framework for representing candidate pairs in an Entity Resolution (ER) task, we devised a method that allows to prioritize those candidates. We show how to exploit graph embeddings to capture hidden patterns in the graph, which can be a good hint for detecting matches. The method we propose can be employed when resources and/or time are limited to yield an approximate ER, since it has been shown to always achieve the same or even a better performance than PPS on real-world datasets.

Finally, we are currently working on a framework to unify the blocking and matching phases by exploiting graph embeddings. This work is the first step to explore that research direction.

References

[1] G. Papadakis, E. Ioannou, E. Thanos, T. Palpanas, The Four Generations of Entity Resolution, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2021.

[2] S. Bergamaschi, D. Beneventano, F. Mandreoli, R. Martoglia, F. Guerra, M. Orsini, L. Po, M. Vincini, G. Simonini, S. Zhu, L. Gagliardelli, L. Magnotta, From Data Integration to Big Data Integration, in: A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years, volume 31 of *Studies in Big Data*, Springer, 2018, pp. 43–59.

[3] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, M. Koubarakis, Three-dimensional Entity Resolution with JedAI, Inf. Syst. 93 (2020) 101565.

[4] L. Gagliardelli, G. Papadakis, G. Simonini, S. Bergamaschi, T. Palpanas, Generalized Supervised Meta-blocking, PVLDB 15 (2022) accepted for publication.

[5] G. Simonini, L. Gagliardelli, S. Bergamaschi, H. V. Jagadish, Scaling entity resolution: A loosely schema-aware approach, Inf. Syst. 83 (2019) 145–165.

[6] G. Simonini, G. Papadakis, T. Palpanas, S. Bergamaschi, Schema-Agnostic Progressive Entity Resolution, IEEE TKDE 31 (2019) 1208–1221.

[7] B. Perozzi, R. Al-Rfou, S. Skiena, DeepWalk: Online Learning of Social Representations, in: KDD, ACM, 2014, pp. 701–710.

[8] A. Grover, J. Leskovec, node2vec: Scalable Feature Learning for Networks, in: KDD, ACM, 2016, pp. 855–864.

[9] H. Chen, B. Perozzi, Y. Hu, S. Skiena, HARP: Hierarchical Representation Learning for Networks, in: AAAI, AAAI Press, 2018, pp. 2127–2134.

[10] A. Doan, P. Konda, P. Suganthan G. C., Y. Govind, D. Paulsen, K. Chandrasekhar, P. Martinkus, M. Christie, Magellan: Toward Building Ecosystems of Entity Matching Solutions, Commun. ACM 63 (2020) 83–91.

[11] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, Deep Learning for Entity Matching: A Design Space Exploration, in: SIGMOD Conference, ACM, 2018, pp. 19–34.

[12] Y. Li, J. Li, Y. Suhara, A. Doan, W. Tan, Deep Entity Matching with Pre-Trained Language Models, PVLDB 14 (2020) 50–60.

[13] L. Gagliardelli, G. Simonini, S. Bergamaschi, RulER: Scaling Up Record-level Matching Rules, in: EDBT, OpenProceedings.org, 2020, pp. 611–614.

[14] D. Firmani, S. Galhotra, B. Saha, D. Srivastava, Robust Entity Resolution Using a CrowdOracle, IEEE Data Eng. Bull. 41 (2018) 91–103.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed Representations of Words and Phrases and their Compositionality, in: NIPS, 2013, pp. 3111–3119.

[16] S. Bergamaschi, L. Gagliardelli, G. Simonini, S. Zhu, Bigbench workload executed by using apache flink, Procedia Manufacturing 11 (2017) 695–702.

[17] L. Gagliardelli, G. Simonini, D. Beneventano, S. Bergamaschi, SparkER: Scaling Entity Resolution in Spark, in: EDBT, OpenProceedings.org, 2019, pp. 602–605.

[18] H. Köpcke, A. Thor, E. Rahm, Evaluation of entity resolution approaches on real-world match problems, PVLDB 3 (2010) 484–493.

[19] D. Obraczka, J. Schuchart, E. Rahm, EAGER: Embedding-Assisted Entity Resolution for Knowledge Graphs, arXiv preprint arXiv:2101.06126 (2021).

[20] L. Gagliardelli, S. Zhu, G. Simonini, S. Bergamaschi, BigDedup: A Big Data Integration Toolkit for Duplicate Detection in Industrial Scenarios, in: TE, volume 7 of *Advances in Transdisciplinary Engineering*, IOS Press, 2018, pp. 1015–1023.

[21] G. Simonini, L. Zecchini, S. Bergamaschi, F. Naumann, Entity Resolution On-Demand, PVLDB 15 (2022) 1506–1518.

[22] T. Papenbrock, A. Heise, F. Naumann, Progressive Duplicate Detection, IEEE TKDE 27 (2015) 1316–1329.

[23] S. Galhotra et al., Efficient and effective ER with progressive blocking, VLDB J. 30 (2021) 537–557.

[24] R. Cappuzzo, P. Papotti, S. Thirumuruganathan, Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks, in: SIGMOD Conference, ACM, 2020, pp. 1335–1349.

[25] F. Benedetti, D. Beneventano, S. Bergamaschi, G. Simonini, Computing inter-document similarity with context semantic analysis, Inf. Syst. 80 (2019) 136–147.