

Entity Resolution On-Demand for Querying Dirty Datasets

(Discussion Paper)

Giovanni Simonini^{1,*}, Luca Zecchini¹, Felix Naumann² and Sonia Bergamaschi¹

¹University of Modena and Reggio Emilia, Italy

²Hasso Plattner Institute, University of Potsdam, Germany

Abstract

Entity Resolution (ER) is the process of identifying and merging records that refer to the same real-world entity. ER is usually applied as an expensive cleaning step on the entire data before consuming it, yet the relevance of cleaned entities ultimately depends on the user's specific application, which may only require a small portion of the entities. We introduce BrewER, a framework designed to evaluate SQL SP queries on unclean data while progressively providing results as if they were obtained from cleaned data. BrewER aims at cleaning a single entity at a time, adhering to an ORDER BY predicate, thus it inherently supports top- k queries and stop-and-resume execution. This approach can save a significant amount of resources for various applications. BrewER has been implemented as an open-source Python library and can be seamlessly employed with existing ER tools and algorithms. We thoroughly demonstrated its efficiency through its evaluation on four real-world datasets.

Keywords

Entity Resolution, Data Integration, ELT

1. Entity Resolution On-Demand

Entity Resolution (ER) is the process of identifying and merging records in a dataset that refer to the same real-world entity [1]. It is a fundamental task for data cleaning and integration [2]. An entity is considered completely resolved when all of its records have been matched and their values consolidated through data fusion [3] to yield a unique representative record. So, state-of-the-art ER methods use: (i) a binary matching function (i.e., a *matcher*) to detect matches between pairs of records, which can be computationally expensive; (ii) a resolution function (e.g., majority voting) to remove inconsistencies in attribute values and yield a unique representative record from a cluster of matching records. Since ER is an inherently quadratic problem, usually also a preliminary blocking step [4] is needed to make it scale. In this step, a blocking function is used to build a set of candidate pairs of records (i.e., possible matches), relying on their similarity, to be checked by the matching function.

SEBD 2023: 31st Symposium on Advanced Database Systems, July 02–05, 2023, Galzignano Terme, Padua, Italy


*Corresponding author.

✉ giovanni.simonini@unimore.it (G. Simonini); luca.zecchini@unimore.it (L. Zecchini); felix.naumann@hpi.de (F. Naumann); sonia.bergamaschi@unimore.it (S. Bergamaschi)

🆔 0000-0002-3466-509X (G. Simonini); 0000-0002-4856-0838 (L. Zecchini); 0000-0002-4483-1389 (F. Naumann); 0000-0001-8087-6587 (S. Bergamaschi)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

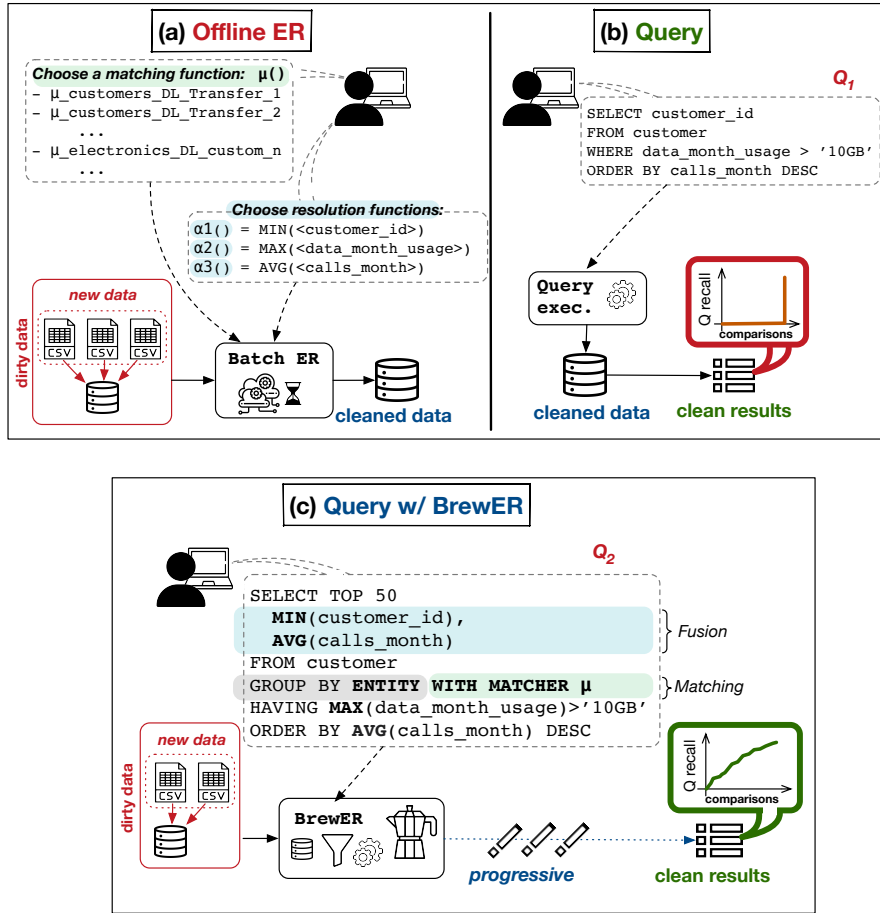


Figure 1: The traditional ER pipeline (a) that has to be defined and executed before querying the data (b) to get exact results. Instead, BrewER (c) allows the user to specify the ER algorithm to use inside the query, then it resolves entities one at a time to yield the result progressively.

1.1. Executing queries on dirty data with existing solutions

Traditionally, ER is employed as a cleaning step before using the data. Yet, in many practical scenarios this might not be convenient:

Example 1.1. Ellen is a data scientist building a machine learning model to predict customer churn for a telecom company, with the following requirements: (i) she has limited time to add new data to her dataset, which will contain duplicates; (ii) she has business priorities: it is better to have clean data for high-value customers (i.e., those that make more phone calls) than for low-value ones, and only customers with a certain data usage (e.g., those that have a monthly data usage greater than 10 GB) should be considered—she can express this with Query 1 in Figure 1b.

For ER, Ellen already has a matching function to choose (adapting some internal pre-trained deep learning models) and she knows rules for resolving the conflicts in the attribute values of the clusters of matching records (e.g., $AVG(calls_month)$, $MAX(data_month_usage)$, etc.).

The example above depicts a common scenario for data practitioners (e.g., data scientists), characterized by:

- An information need: only some entities are relevant and some entities are more relevant than others;
- Time constraints: data might become outdated quickly and/or users want to do a fast exploration of relevant portions of cleaned data.

Example 1.2. *To get correct results for the query (i.e., taking into account that some records are duplicates) Ellen employs a traditional ER framework to clean the entire dataset (Figure 1a). However, she soon realizes that ER is the bottleneck due to its inherently quadratic complexity and the cost of the matching function, which involves expensive operations based on deep neural network models. As a result, it takes a significant amount of time to clean the entire dataset using ER. Furthermore, to build the right ER pipeline is not a trivial task: she would need to debug the ER pipeline with the data at hand (e.g., to check if the matcher she is employing is performing well for high-value customers), but she cannot stop the ER process after receiving a handful of the entities to inspect—this is because those entities might not be relevant for the query or might be only partially resolved, which could lead to incorrect results. Alternatively, she would have to manually select records from the dataset to test the ER pipeline, which is also time-consuming.*

The motivating example highlights the need for a more efficient and targeted approach to ER that prioritizes cleaning.

1.2. A novel approach to execute queries on dirty data

We propose BrewER¹ [5], an ER framework that aims to provide an efficient and targeted approach to ER by evaluating SQL SP (Selection and Projection) queries on dirty data and returning results as if they were issued on cleaned data. The key feature of BrewER is its ability to perform ER progressively, guided by an ORDER BY clause, to incrementally return the most relevant results to the data scientist. This approach avoids matching and resolving entities that are not part of the final result, thereby saving time and resources. Additionally, BrewER inherently supports top-*k* queries and allows for stop-and-resume execution. To enable this progressive approach to ER, BrewER introduces a special “GROUP BY ENTITY WITH MATCHER [matcher of choice]” operator, meaning that matching records should be grouped according to the selected matcher—then, filtering conditions on the entities can be applied by means of the HAVING clause. Overall, BrewER offers a more efficient and targeted approach to ER, which can save time and resources, especially for data scientists dealing with large and complex datasets.

Example 1.3. *Using BrewER, Ellen can easily adapt her original SQL query to work with dirty data by employing a special GROUP BY statement and moving the selection statements into the HAVING clause, predicated on each group (i.e., each entity), as shown in Figure 1c. Additionally, she specifies the resolution functions for ER within the SQL query as aggregate functions. Once Ellen has specified her new, equivalent query, BrewER executes it directly on the dirty data, applying*

¹This paper is a revisited version of the one published at the 48th International Conference on Very Large Databases (VLDB 2022).

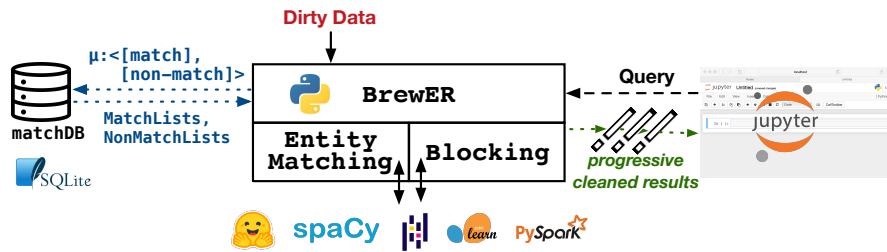


Figure 2: The BrewER framework architecture.

ER progressively on the right portion of the data to yield correct results incrementally. This allows Ellen to receive the first entities in a fraction of the time required by existing ER frameworks. She can explore new data without completely cleaning it and maximize the ER efforts on the entities she actually needs for her task. Moreover, BrewER allows Ellen to stop the execution at any time with the guarantee that the results produced so far are correct. She can then inspect the results of the ER process for entities of interest and debug it if needed. This feature saves time and resources compared to traditional ER frameworks, where debugging and testing the ER pipeline requires a complete cleaning of the entire dataset. Finally, BrewER keeps track of both executed comparisons and resolved entities to avoid repeating the same operations when multiple queries are issued on the same data. This further improves the efficiency of the ER process, allowing Ellen to perform her analysis more quickly and accurately.

More generally, our proposed approach is well-suited for addressing one of the major challenges in data lake management systems [6]: to support extraction and cleaning as part of the integration pipeline on-demand. Similarly, on-demand data transformation that returns results in a timely manner is a fundamental requirement of ELT (Extract-Load-Transform) pipelines, especially when combined with top- k queries for debugging transformations [7].

The main contributions of BrewER can be summarized as follows. We formalize the concept of ER-on-demand, which involves progressively cleaning and emitting entities that satisfy queries issued directly on dirty datasets, and propose an algorithm for it. Finally, we implement this algorithm in an open-source system and extensively evaluate it on four real-world datasets, demonstrating its effectiveness.

2. The BrewER Framework

BrewER is designed as a flexible and adaptable framework, as illustrated in Figure 2. It is implemented as a Python library, whose code is publicly available on GitHub². This approach allows the seamless integration of BrewER into Python workflows in Jupyter³ notebooks, as we show below in Section 3. Being BrewER agnostic towards the selected blocking and matching functions, users can integrate it with their preferred binary matching libraries (such as DeepMatcher [8], Ditto [9], etc.) and blocking techniques (like Magellan [10], JedAI [11],

²<https://github.com/dbmodena/BrewER>

³<https://jupyter.org>

```

SELECT [TOP  $k$ ]  $\langle \alpha_j(A_j) \rangle$ 
FROM  $\mathcal{D}$ 
[WHERE  $\varphi$ ]
GROUP BY  $ENTITY$  WITH MATCHER  $\mu$ 
[HAVING  $\langle \alpha_j(A_j) \rangle$  {LIKE | IN | < | ≤ | > | ≥ | =}  $const$ ]
[ORDER BY  $\alpha_j(A_j)$  [ASC | DESC]]

```

Figure 3: Query syntax in BrewER.

or SparkER [12]). The system then performs ER in an on-demand fashion while executing the user’s query with the algorithm presented in [5].

In particular, BrewER builds on the output of the blocking function and performs a preliminary filtering of the blocks, keeping only the ones containing records whose values might lead to the generation of an entity appearing in the result of the query. The records of the blocks that pass the filtering are then inserted in a priority queue, keeping for each one the list of its candidate matches. The priority is defined according to the value of the attribute appearing in the ORDER BY clause, in ascending or descending order. BrewER iterates on the priority queue, considering at each iteration the head element: if it is a record, its candidate matches are checked generating a completely resolved entity; otherwise (i.e., it is a completely resolved entity), it is emitted or discarded based on whether or not it satisfies the query.

To optimize the performance of the matching functions and avoid re-comparing candidate pairs, BrewER maintains separate databases for the lists of matching and non-matching records for each matching function adopted by the user. To save space, users may opt to store only the final resolved entities—the resolution functions cannot change across queries in this case.

2.1. Supported queries

Figure 3 reports the syntax of valid queries in BrewER. Please note that we maintain the capability of filtering dirty records directly by expressing WHERE clauses. Instead, selection predicates on the resolved entities have to be expressed with HAVING conditions. BrewER supports several aggregate functions: MIN, MAX, AVG, and user-defined aggregations, such as MEDIAN and VOTE (a.k.a. majority voting). The choice of this set of functions was driven by two key observations: (i) they cover most real-world use cases; (ii) they can be naturally declared as part of SQL queries. The only limitation for a user-defined aggregation is that it has to be *bounded*, i.e., a function that takes as input one or many values and returns an aggregated value located between the minimum and the maximum of those values (for numerical attributes) or chosen among them (for categorical attributes).

3. Experiments and Demonstration

Through our experimental evaluation [5], we point out the benefits that BrewER can generate in terms of elapsed time and saved resources. In Table 1, we report the characteristics of the datasets used in our experiments, presenting significant differences regarding the size (in terms of both records and attributes) and the domain, covering commercial products (cameras in the

Dataset	#Records	#Matches	#Entities (AVG Size)	#Attributes	Domain
SIGMOD20	13.58k	12.01k	3.06k (4.4)	4	cameras
SIGMOD21	1.12k	1.08k	190 (5.9)	4	USB sticks
Altosight	12.47k	12.44k	453 (27.534)	4	USB sticks
Funding	17.46k	16.70k	3.11k (5.6)	17	organizations

Table 1
Characteristics of the selected datasets.

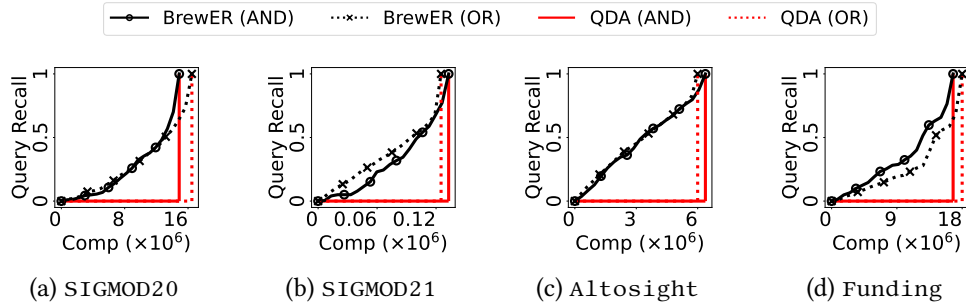


Figure 4: Progressive recall.

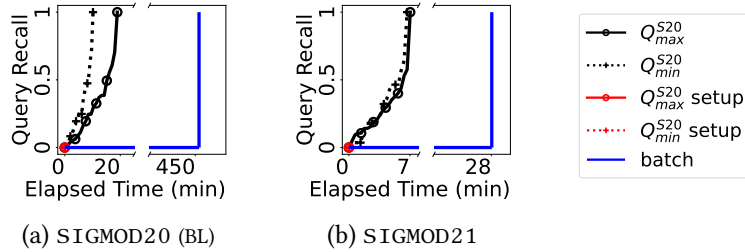


Figure 5: Query execution runtime.

case of SIGMOD20⁴, part of the Alaska benchmark [13], and USB sticks for SIGMOD21⁵ and its superset provided by Altosight⁶) and organizations (Funding⁷ [14]).

In Figure 4, we show the results obtained by running batches of *conjunctive* (i.e., with the HAVING conditions in AND) and *disjunctive* (i.e., with the HAVING conditions in OR) queries with BrewER on the four datasets. The plot shows the average number of comparisons needed to reach a certain query recall (i.e., the emission of a certain percentage of resulting entities). BrewER is able to return the entities with a high priority in a small fraction of the time required for performing the entire cleaning process, which has to be carried out by the batch algorithms to get the query results (here we consider as a baseline QDA [15], a query-driven

⁴<http://www.inf.uniroma3.it/db/sigmod2020contest>

⁵<https://dbgroup.ing.unimo.it/sigmod21contest>

⁶<https://altosight.com>

⁷https://raw.githubusercontent.com/qcri/data_civilizer_system/master/grecord_service/gr/data/address/address.csv

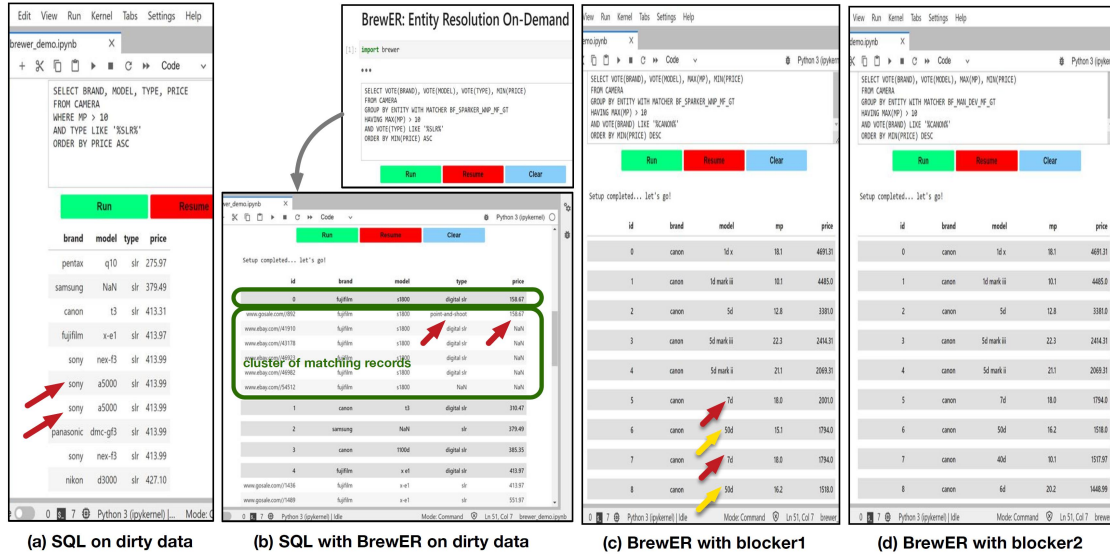


Figure 6: Demonstration scenarios: querying dirty datasets (a-b), ER pipeline debugging (c-d).

batch approach and the closest prior work to BrewER). Figure 5, reporting a similar experiment (considering in this case the queries in the batches yielding the largest and the smallest result sets), allows us to highlight the significant difference in terms of elapsed time compared to the traditional batch approach. In our full research paper [5], you can find out more details about the described experiments and several additional experiments covering the impact of blocking, of different aggregate function, and the shortcomings of the existing related approaches. In our demonstration [16], we show the benefits of BrewER for data practitioners, addressing in particular two scenarios, depicted in Figure 6 and described below.

3.1. Querying dirty datasets

BrewER makes it possible to run queries on dirty datasets obtaining the progressive emission of the cleaned resulting entities (Figure 6b), as soon as they are obtained, avoiding the inconsistencies that would be raised by running the query directly on the dirty dataset (Figure 6a). BrewER inherently supports top-k queries, thus Ellen can run such a query for the quick emission of the results with the highest priority; once inspected the returned entities, she can decide whether to resume the query to get the complete result set.

3.2. ER pipeline debugging

BrewER makes it also possible to obtain early insights on the quality of the ongoing cleaning process, assessing the goodness of the chosen combination of blocking and matching functions. As depicted in Figure 6c, Ellen can exploit top-k queries to check the absence of inconsistencies in the result set. If some issues are spotted (e.g., duplicate records not matched because of a too aggressive blocking function or a weak matching function), she can intervene and redesign the

ER pipeline, saving a significant amount of time and resources compared to batch solutions, which allow to perform such controls only after the completion of the cleaning process.

4. Related Work

The shortcomings of the traditional batch approach to ER are pointed out in literature and different solutions have been proposed to overcome its limitations in dynamic scenarios. In particular, the related work can be grouped into two main research directions: *progressive* approaches and *query-driven* approaches. These methods present several significant differences compared to BrewER [5], whose *on-demand* approach implies the co-existence of both aspects.

4.1. Progressive ER

Progressive approaches to ER [17, 18, 19, 20, 21] try to maximize the impact of ER on a dirty dataset in a limited amount of time. The key idea of these methods is to prioritize the comparisons for the candidate pairs of records for which the probability to match is higher. Thus, it is not possible for the user to define a priority based on their interests, as done in BrewER through the `ORDER BY` clause of the query. Furthermore, operating at match level and not at entity level, these approaches do not guarantee to dispose of clean entities before completing the ER process, while BrewER progressively returns the clean entities appearing in the result of the query, according to the user-defined priority.

4.2. Query-driven ER

Query-driven approaches to ER [15, 22] aim at performing ER only on the portion of the dataset which is needed to answer the query. These solutions are the closest prior works to BrewER, operating at block level to detect the comparisons that are not relevant for the query at hand. Nevertheless, query-driven approaches are not designed to support the progressive emission of the entities; thus, it is needed to wait for the end of the cleaning process to be able to inspect the results of the query. Moreover, they can support only a limited range of aggregate functions (e.g., the average or the majority voting are not supported).

5. Conclusion

We presented BrewER [5], a framework for Entity Resolution (ER) that allows users to filter entities of interest from dirty data without having to clean the entire dataset. BrewER achieves this by evaluating SQL SP queries on dirty data and progressively returning results as if they were issued on cleaned data. The system is flexible and adaptable, allowing users to integrate their preferred binary matching and blocking techniques. BrewER has been implemented as an open-source Python library, which can be seamlessly integrated in data science workflows (e.g., in Jupyter notebooks) We demonstrated the efficacy of BrewER on four real-world datasets and have shown that its overhead is negligible in real-world use cases. Future work includes exploring how to support SQL SPJ queries for multi-table dirty datasets and additional features for ER pipeline debugging.

References

- [1] P. Christen, *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*, Data-Centric Systems and Applications (DCSA), Springer, 2012. doi:10.1007/978-3-642-31164-2.
- [2] X. L. Dong, D. Srivastava, *Big Data Integration*, Synthesis Lectures on Data Management (SLDM), Morgan & Claypool Publishers, 2015. doi:10.2200/S00578ED1V01Y201404DTM040.
- [3] J. Bleiholder, F. Naumann, *Data Fusion*, ACM Computing Surveys (CSUR) 41 (2008) 1:1–1:41. doi:10.1145/1456650.1456651.
- [4] G. Papadakis, D. Skoutas, E. Thanos, T. Palpanas, *Blocking and Filtering Techniques for Entity Resolution: A Survey*, ACM Computing Surveys (CSUR) 53 (2021) 31:1–31:42. doi:10.1145/3377455.
- [5] G. Simonini, L. Zecchini, S. Bergamaschi, F. Naumann, *Entity Resolution On-Demand*, Proceedings of the VLDB Endowment (PVLDB) 15 (2022) 1506–1518. doi:10.14778/3523210.3523226.
- [6] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, P. C. Arocena, *Data Lake Management: Challenges and Opportunities*, Proceedings of the VLDB Endowment (PVLDB) 12 (2019) 1986–1989. doi:10.14778/3352063.3352116.
- [7] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, C. Welton, *MAD Skills: New Analysis Practices for Big Data*, Proceedings of the VLDB Endowment (PVLDB) 2 (2009) 1481–1492. doi:10.14778/1687553.1687576.
- [8] 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: Proceedings of the International Conference on Management of Data (SIGMOD), ACM, 2018, pp. 19–34. doi:10.1145/3183713.3196926.
- [9] Y. Li, J. Li, Y. Suhara, A. Doan, W. Tan, *Deep Entity Matching with Pre-Trained Language Models*, Proceedings of the VLDB Endowment (PVLDB) 14 (2020) 50–60. doi:10.14778/3421424.3421431.
- [10] P. Konda, S. Das, P. Suganthan G. C., A. Doan, A. Ardalani, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, V. Raghavendra, *Magellan: Toward Building Entity Matching Management Systems*, Proceedings of the VLDB Endowment (PVLDB) 9 (2016) 1197–1208. doi:10.14778/2994509.2994535.
- [11] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, M. Koubarakis, *Three-dimensional Entity Resolution with JedAI*, Information Systems (IS) 93 (2020) 101565:1–101565:17. doi:10.1016/j.is.2020.101565.
- [12] L. Gagliardelli, G. Simonini, D. Beneventano, S. Bergamaschi, *SparkER: Scaling Entity Resolution in Spark*, in: Proceedings of the International Conference on Extending Database Technology (EDBT), OpenProceedings.org, 2019, pp. 602–605. doi:10.5441/002/edbt.2019.66.
- [13] V. Crescenzi, A. De Angelis, D. Firmani, M. Mazzei, P. Merialdo, F. Piai, D. Srivastava, *Alaska: A Flexible Benchmark for Data Integration Tasks*, arXiv preprint (2021). doi:10.48550/arXiv.2101.11259.

- [14] D. Deng, W. Tao, Z. Abedjan, A. Elmagarmid, I. F. Ilyas, G. Li, S. Madden, M. Ouzzani, M. Stonebraker, N. Tang, Unsupervised String Transformation Learning for Entity Consolidation, in: Proceedings of the International Conference on Data Engineering (ICDE), IEEE Computer Society, 2019, pp. 196–207. doi:10.1109/ICDE.2019.00026.
- [15] H. Altwaijry, D. V. Kalashnikov, S. Mehrotra, Query-Driven Approach to Entity Resolution, Proceedings of the VLDB Endowment (PVLDB) 6 (2013) 1846–1857. doi:10.14778/2556549.2556567.
- [16] L. Zecchini, G. Simonini, S. Bergamaschi, F. Naumann, BrewER: Entity Resolution On-Demand, Proceedings of the VLDB Endowment (PVLDB) 16 (2023).
- [17] S. E. Whang, D. Marmaros, H. Garcia-Molina, Pay-As-You-Go Entity Resolution, IEEE Transactions on Knowledge and Data Engineering (TKDE) 25 (2013) 1111–1124. doi:10.1109/TKDE.2012.43.
- [18] T. Papenbrock, A. Heise, F. Naumann, Progressive Duplicate Detection, IEEE Transactions on Knowledge and Data Engineering (TKDE) 27 (2015) 1316–1329. doi:10.1109/TKDE.2014.2359666.
- [19] D. Firmani, B. Saha, D. Srivastava, Online Entity Resolution Using an Oracle, Proceedings of the VLDB Endowment (PVLDB) 9 (2016) 384–395. doi:10.14778/2876473.2876474.
- [20] G. Simonini, G. Papadakis, T. Palpanas, S. Bergamaschi, Schema-agnostic Progressive Entity Resolution, in: Proceedings of the International Conference on Data Engineering (ICDE), IEEE Computer Society, 2018, pp. 53–64. doi:10.1109/ICDE.2018.00015.
- [21] L. Gazzarri, M. Herschel, Progressive Entity Resolution over Incremental Data, in: Proceedings of the International Conference on Extending Database Technology (EDBT), OpenProceedings.org, 2023, pp. 80–91. doi:10.48786/edbt.2023.07.
- [22] H. Altwaijry, S. Mehrotra, D. V. Kalashnikov, QuERy: A Framework for Integrating Entity Resolution with Query Processing, Proceedings of the VLDB Endowment (PVLDB) 9 (2015) 120–131. doi:10.14778/2850583.2850587.