Progressive Query-driven Entity Resolution

Luca Zecchini[0000−0002−4856−0838]

University of Modena and Reggio Emilia, Italy
luca.zecchini@unimore.it

Abstract. *Entity Resolution (ER)* aims to detect in a dirty dataset the records that refer to the same real-world entity, playing a fundamental role in data cleaning and integration tasks. Often, a data scientist is only interested in a portion of the dataset (e.g., *data exploration*); this interest can be expressed through a query. The traditional *batch* approach is far from optimal, since it requires to perform ER on the whole dataset before executing a query on its *cleaned* version, performing a huge number of useless comparisons. This causes a waste of time, resources and money. Proposed solutions to this problem follow a *query-driven* approach (perform ER only on the useful data) or a *progressive* one (the entities in the result are emitted as soon as they are solved), but these two aspects have never been reconciled. This paper introduces `BrewER` framework, which allows to execute *clean queries on dirty datasets* in a *query-driven* and *progressive* way, thanks to a preliminary filtering and an iteratively managed sorted list that defines emission priority. Early results obtained by first `BrewER` prototype on real-world datasets from different domains confirm the benefits of this combined solution, paving the way for a new and more comprehensive approach to ER.

Keywords: Entity Resolution · Data Integration · Data Cleaning.

1 Introduction

Entity Resolution (ER) is a fundamental task for data integration [11], aiming to detect in a dirty dataset the records (*duplicates* [10]) that represent the same real-word object (*entity*). The duplicates are detected by applying a *matching function* (e.g., a trained binary classifier) to each possible pair of records (or to the pairs formed by records appearing in the same block, if a *blocking function* [12] is applied), in order to determine if they refer or not to the same entity (in the first case, they are referred to as *matches*).

Once a cluster of matches is found, its records are merged to create a single consistent record representing the entity (*data fusion* [7]), removing dataset redundancy. The matches often present missing, wrong or conflicting values; data fusion is performed through the application of a *resolution function*, which determines the value to be assigned to each attribute of the entity according to the *aggregation function* (e.g., maximum/minimum value, majority voting, etc.) defined for it by the data scientist.

In concrete situations (e.g., *data exploration*), a data scientist is often only interested in a specific portion of the dataset; this interest can be expressed

through a query. Since performing a query on the dirty dataset may lead to an inconsistent result, it is necessary to perform ER on the dataset; then, the query can be executed on the obtained *cleaned* version. As shown in Figure 1a, all the entities appearing in the result are returned to the data scientist at the end of this pipeline.

When computational resources are limited and/or time is a critical component, this approach (called *batch*) is far from optimal, wasting time, resources and money (e.g., in the case for *pay-as-you-go* contracts, widely used by cloud providers) to perform comparisons which are guaranteed to be useless. These comparisons are required to generate entities with no chance of appearing in the result (e.g., the query in Figure 2 returns only `Canon` cameras, so each comparison performed to retrieve an entity whose brand is `Nikon` is useless and should be avoided) and cause performance degradation, as the data scientist can only run the query after all comparisons have been performed.

In order to overcome the described problems, an innovative approach (Figure 1b) must be able to perform *clean queries on dirty datasets* and it is required to be both *query-driven* (i.e., to perform ER only on the portion of data effectively useful to answer the query, according to the `WHERE` clauses of the query itself) and *progressive* (i.e., to emit the entities appearing in the result as soon as they are solved, following the ordering expressed by the `ORDER BY` clause). This is exactly the aim of `BrewER`.

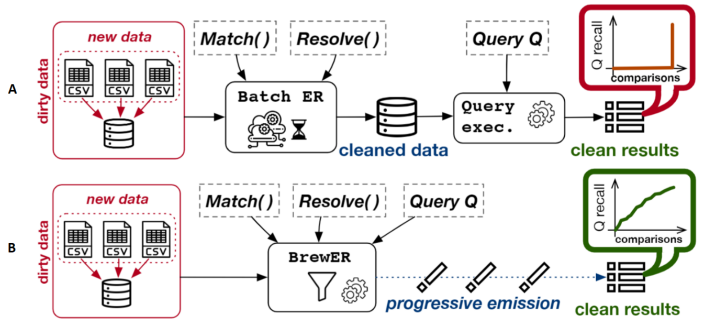


Fig. 1: The traditional *batch* pipeline and the one proposed by `BrewER`.

2 Related Work

Solutions in literature [4, 6, 5] propose a *Query-Driven Approach (QDA)* to ER, executing clean queries on dirty datasets performing comparisons only on the portion of data relevant for the executed query; however, the adopted techniques are not suitable for supporting the progressive emission of the results (e.g., the `ORDER BY` clause is not managed). On the other hand, even progressive solutions have been presented [16, 13, 15], but neither of them considers the possibility of an integration with QDA principles, which is far from trivial. A draft of combined approach based on a graph structure has been presented in [14], but it is limited to approximate solutions for a keyword-search scenario. Therefore, `BrewER` approach represents a novelty in literature.

3 BrewER: A Progressive Query-driven ER Framework

`BrewER` framework for ER reconciles the described approaches, executing clean queries on dirty datasets in a query-driven and progressive way.

The implementation of the query-driven approach consists of a preliminary filtering of the blocks, which aims to keep only the blocks that could generate an entity appearing in the result. `BrewER` approach to the eventual blocking function adopted by the user is agnostic; if no blocking is performed, the whole dataset can be interpreted as a single block. Furthermore, blocks are transitively closed. If the `WHERE` clauses of the query are in `OR` (at the moment, only conjunctive and disjunctive queries are supported), it is checked that at least one record in the block satisfies at least one of the defined clauses; on the other hand, if the clauses are in `AND`, it is verified that all the clauses, each considered by itself, are satisfied by at least one record in the block. The records which satisfy at least one of the clauses are called *seed records*. The aggregation function for each attribute is defined by the user; `MIN`, `MAX`, `AVG`, `VOTE` (majority voting) and `RANDOM` are supported, with `SUM` to be implemented. In case of numeric attributes, the described filtering is not applied using functions that can generate new values (i.e., `AVG` and `SUM`).

The progressive emission is obtained through an iteratively managed sorted list (Figure 2) called *Ordering List (OL)*. The records appearing in the blocks that pass the filtering are marked as *unsolved* (ER not yet performed) and inserted in `OL`, each one with a list containing the identifiers of its *neighbours* (i.e., the records in the same block). At the beginning of each iteration, the elements in `OL` are sorted according to the ordering mode and the attribute, called *Ordering Key (OK)*, expressed by the `ORDER BY` clause of the query. Then, the first element (i.e., the one with the highest emission priority) is checked. If it is marked as *unsolved* (2.1a), it is compared with its neighbours (even for the matching function `BrewER` adopts an agnostic approach); once identified the cluster of matches, all the matching elements are removed from `OL`, while a single element representing the cluster (with the aggregated `OK` value), marked as *solved*, is inserted (2.1b). If it is marked as *solved* (2.2a), the resolution function is applied on the represented cluster: if the obtained entity satisfies the query, it is emitted; otherwise, it is discarded. Comparisons involve seed neighbours first: if a non-seed does not match any seed, it can be discarded. Iterations can run until `OK` is empty or can be stopped after the emission of k entities (`TOP(K)` queries).

Optimizations. In case of *discordant ordering* (`MIN`/`DESC` or `MAX`/`ASC`), it is possible to optimize the described algorithm by inserting in `OL` only the seed records, while the non-seed records in their blocks only appear in their lists of neighbours (fewer comparisons). This is possible because if a matching non-seed neighbour (whose `OK` value is not therefore sorted in `OL`) alters the `OK` value for the first element, the generated *solved* record priority is updated by changing its position when sorting `OL` at the beginning of the next iteration (*delayed emission*), guaranteeing the correctness of the ordering (while in `MAX`/`DESC` or `MIN`/`ASC` cases this variant could alter the correct emission ordering).

Fig. 2: `BrewER` in action.

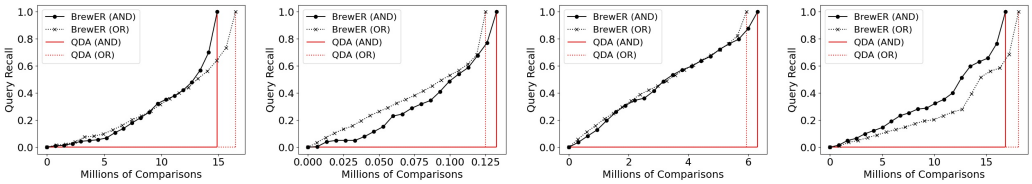
4 Evaluation

The evaluation of `BrewER`, whose implementation is realized in `Python`, is performed on real-world datasets from different domains (Table 1) with known ground truth: `SIGMOD20` [2, 8] (camera specifications from e-commerce websites, pre-processed using a variant of the algorithm described in [17]), `SIGMOD21` [3] (USB stick specifications from e-commerce websites) and its superset (both provided by Altosight [1]), `Funding` [9] (organizations presenting financing requests). All strings are put in lowercase and `OK` values cast to float, filtering out the records whose `OK` is `null`, since they do not alter the emission ordering.

Plots in Figures 3 and 4 show the early results, in terms of progressive *query recall* (number of emitted entities / size of the result set) after `x` performed comparisons (ground truth as matching function), obtained computing mean values on batches of 20 queries (both for conjunctive and disjunctive case), selected as the ones emitting most entities out of wider batches of at least 50 queries. Values for the considered attributes are randomly selected from lists containing the most common ones. Figure 3 clearly shows the progressive nature of *BrewER* and highlights its potential in anticipating emissions, considering as batch baseline an adapted version of QDA [5]. When progressiveness is lower because of delayed emissions (discordant case), optimized algorithm generates a further significant reduction of the comparisons (Figure 4; analogous plots for disjunctive case).

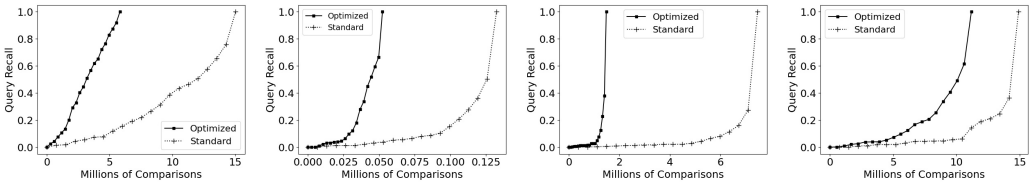
Table 1: Characteristics of the selected datasets.

Name	Records	Duplicates	Entities (Mean Size)	Attributes	Ordering Key
SIGMOD20 [8, 17]	13.58k	12.01k	3.06k (4.439)	5	Megapixels
SIGMOD21	1.12k	1.08k	190 (5.879)	5	Price
Altosight	12.47k	12.44k	453 (27.534)	5	Price
Funding [9]	17.46k	16.70k	3.11k (5.609)	18	Amount



(a) SIGMOD20 Brand AND Model Brand OR Brand (b) SIGMOD21 Brand AND Size Brand OR Brand (c) Altosight Brand AND Size Brand OR Brand (d) Funding Source AND Legal_Name Source OR Source

Fig. 3: Progressive query recall in `MAX/DESC` and `MIN/ASC` cases (no blocking).



(a) SIGMOD20 Brand AND Model (b) SIGMOD21 Brand AND Size (c) Altosight Brand AND Size (d) Funding Source AND Legal_Name

Fig. 4: Progressive query recall in `MAX/ASC` and `MIN/DESC` cases (no blocking).

5 Conclusions and next steps

Early results confirm the benefits of the approach adopted by `BrewER`, in terms both of reduction of performed comparisons and of progressive emission of the results, paving the way for new and more comprehensive solutions to ER tasks.

`BrewER` has a lot of room for improvement (the implementation itself needs to be optimized, even considering the migration to a faster language), with significant scenarios to be deepened and integrated in the framework; as for discordant case and non-seed record comparisons, it is fundamental to find out and avoid all situations causing useless comparisons. Benefits have to be evaluated in case of blocking: the actual agnostic approach is considered as a strength, since it allows to combine `BrewER` with the most innovative solutions in this field (the same happens for matching functions), but even the possibility of including blocking itself in the progressive pipeline has to be investigated. Cases to be studied are that of `TOP(K)` queries (supposed to take full advantage from this approach) and that of temporal series, while missing value imputation, together with the possibility of keeping track of executed queries, can turn `BrewER` into a powerful data preparation tool, leading to a progressive cleaning of the dataset. Furthermore, since ER can be seen as a case of binary classification, `BrewER` impact is not strictly bound to this field, and it is important to study how to extend the presented techniques to other classification tasks.

`BrewER` is going to be presented and further explored in a dedicated research paper, containing the formalized algorithm and new experiments covering some of these cases; the code will be made available at the time of its publication.

References

1. Altosight Website. https://altosight.com/
2. SIGMOD 2020 Programming Contest Website. http://www.inf.uniroma3.it/db/sigmod2020contest/
3. SIGMOD 2021 Programming Contest Website. https://dbgroup.ing.unimo.it/sigmod21contest/
4. Altwaijry, H., Kalashnikov, D.V., Mehrotra, S.: Query-Driven Approach to Entity Resolution. Proc. VLDB Endow. **6**(14), 1846–1857 (2013). https://doi.org/10.14778/2556549.2556567
5. Altwaijry, H., Kalashnikov, D.V., Mehrotra, S.: QDA: A Query-Driven Approach to Entity Resolution. IEEE Trans. Knowl. Data Eng. **29**(2), 402–417 (2017). https://doi.org/10.1109/TKDE.2016.2623607
6. Altwaijry, H., Mehrotra, S., Kalashnikov, D.V.: QuERy: A Framework for Integrating Entity Resolution with Query Processing. Proc. VLDB Endow. **9**(3), 120–131 (2015). https://doi.org/10.14778/2850583.2850587
7. Bleiholder, J., Naumann, F.: Data fusion. ACM Comput. Surv. **41**(1), 1:1–1:41 (2008). https://doi.org/10.1145/1456650.1456651
8. Crescenzi, V., De Angelis, A., Firmani, D., Mazzei, M., Merialdo, P., Piai, F., Srivastava, D.: Alaska: A Flexible Benchmark for Data Integration Tasks. CoRR **abs/2101.11259** (2021), https://arxiv.org/abs/2101.11259
9. Deng, D., Tao, W., Abedjan, Z., Elmagarmid, A.K., Ilyas, I.F., Li, G., Madden, S., Ouzzani, M., Stonebraker, M., Tang, N.: Unsupervised String Transformation Learning for Entity Consolidation. In: ICDE 2019. pp. 196–207. IEEE (2019). https://doi.org/10.1109/ICDE.2019.00026
10. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate Record Detection: A Survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007). https://doi.org/10.1109/TKDE.2007.250581
11. Papadakis, G., Ioannou, E., Thanos, E., Palpanas, T.: The Four Generations of Entity Resolution. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2021). https://doi.org/10.2200/S01067ED1V01Y202012DTM064
12. Papadakis, G., Skoutas, D., Thanos, E., Palpanas, T.: Blocking and Filtering Techniques for Entity Resolution: A Survey. ACM Comput. Surv. **53**(2), 31:1–31:42 (2020). https://doi.org/10.1145/3377455
13. Papenbrock, T., Heise, A., Naumann, F.: Progressive Duplicate Detection. IEEE Trans. Knowl. Data Eng. **27**(5), 1316–1329 (2015). https://doi.org/10.1109/TKDE.2014.2359666
14. Pietrangelo, A., Simonini, G., Bergamaschi, S., Naumann, F., Koumarelas, I.K.: Towards Progressive Search-driven Entity Resolution. In: SEBD 2018. CEUR Workshop Proceedings, vol. 2161. CEUR-WS.org (2018), http://ceur-ws.org/Vol-2161/paper16.pdf
15. Simonini, G., Papadakis, G., Palpanas, T., Bergamaschi, S.: Schema-Agnostic Progressive Entity Resolution. IEEE Trans. Knowl. Data Eng. **31**(6), 1208–1221 (2019). https://doi.org/10.1109/TKDE.2018.2852763
16. Whang, S.E., Marmaros, D., Garcia-Molina, H.: Pay-As-You-Go Entity Resolution. IEEE Trans. Knowl. Data Eng. **25**(5), 1111–1124 (2013). https://doi.org/10.1109/TKDE.2012.43
17. Zecchini, L., Simonini, G., Bergamaschi, S.: Entity Resolution on Camera Records Without Machine Learning. In: DI2KG@VLDB 2020. CEUR Workshop Proceedings, vol. 2726. CEUR-WS.org (2020), http://ceur-ws.org/Vol-2726/paper3.pdf