BrewER: Entity Resolution On-Demand

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

Giovanni Simonini
University of Modena and Reggio Emilia
Modena, Italy
simonini@unimore.it

Sonia Bergamaschi
University of Modena and Reggio Emilia
Modena, Italy
sonia@unimore.it

Felix Naumann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

The task of entity resolution (ER) aims to detect multiple records describing the same real-world entity in datasets and to consolidate them into a single consistent record. ER plays a fundamental role in guaranteeing good data quality, e.g., as input for data science pipelines. Yet, the traditional approach to ER requires cleaning the entire data before being able to run consistent queries on it; hence, users struggle to tackle common scenarios with limited time or resources (e.g., when the data changes frequently or the user is only interested in a portion of the dataset for the task).

We previously introduced BrewER, a framework to evaluate SQL SP queries on dirty data while progressively returning results as if they were issued on cleaned data, according to a priority defined by the user. In this demonstration, we show how BrewER can be exploited to ease the burden of ER, allowing data scientists to save a significant amount of resources for their tasks.

PVLDB Reference Format:
Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi, and Felix Naumann. BrewER: Entity Resolution On-Demand. PVLDB, 16(12): 4026 - 4029, 2023.
doi:10.14778/3611540.3611612

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbmodena/brewer_demo.

1 ENTITY RESOLUTION ON-DEMAND

Entity Resolution (ER) [3], also known as duplicate detection or record linkage, has the goal of detecting in a given dataset (or across multiple datasets) the records describing the same real-world entity. ER represents one of the core tasks of big data integration [5], thus playing a key role in the data preparation pipelines [7] required in every data science process to guarantee the successful extraction of value from the collected input data.

The same real-world entity can be represented in several heterogeneous ways. It is often hard to correctly associate these representations to the described entity, and therefore to identify them as *matches*. Further, the naïve solution for ER presents a quadratic complexity, since it requires comparing all possible pairs of records to determine if they match. Blocking techniques tackle this crucial scalability issue, discarding as many obvious non-matches as possible and restricting comparisons to similar elements. Blocking and matching therefore constitute the main bricks for ER pipelines in big data scenarios. They are usually followed by a clustering step to ensure the consistency of the matches. Finally, data fusion generates a single record from every cluster of matches, i.e., a unique, *resolved* representation of the entity it refers to. Once all entities have been resolved, the dataset can be queried without the worry of producing duplicate results or inconsistencies. Thus, given a dirty dataset, a user has to clean it entirely first (Figure 1a), and only then a query can be issued (Figure 1b).

This traditional *batch* approach can be a significant burden in terms of time and resources. For instance, state-of-the-art matchers employ deep neural network models that can be resource-draining when applied to large datasets [9]. Further, the batch approach is not suitable to dynamic scenarios, where data changes frequently and practitioners consuming it dispose of a finite amount of resources and time to perform their tasks before the data becomes outdated. For example, they might want to quickly run meaningful queries to perform *data exploration* or to feed a *BI dashboard*: to clean the entire data upfront when only a portion of it is actually needed might not be affordable in terms of time (e.g., data has to be fresh) and money (e.g., in case of *pay-as-you-go* contracts, very common in cloud-based solutions).

To clean only the data useful to answer the user's queries, we previously presented BrewER [13], an algorithm and a framework to perform ER in an *on-demand* fashion. BrewER evaluates SQL SP (*Selection* and *Projection*) queries with ordering on the dirty data and returns the results as if they were issued on the cleaned data. BrewER performs ER only on the portion of the dataset needed to answer the issued query (according to the conditions expressed by the WHERE clause) and returns the cleaned entities satisfying the query as soon as they are obtained. In addition, results are returned in a progressive fashion, according to a priority defined by the user through the ORDER BY clause (example in Figure 1c). The user specifies how to clean the data within the query itself, selecting the blocking and the matching functions to perform ER (BrewER is agnostic towards them, so the user can exploit the ones that best fit the scenario at hand) and the aggregation functions to create the consolidated records from the clusters of detected matches.

Previous work had been proposed to perform ER at query time or progressively, but BrewER is the first contribution to propose a solution that combines the two [13]. In fact, *query-driven* approaches [1, 2] aim at performing ER only on the portion of the dataset that is meaningful for answering the query, but they are not designed to support the progressive emission of the results (thus, they still operate in a batch way) and enable to use a limited range of aggregation functions (e.g., the average or the mode cannot be supported). On the other hand, *progressive* solutions [11, 12, 14] are built to prioritize the comparisons of the most promising candidate

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611612

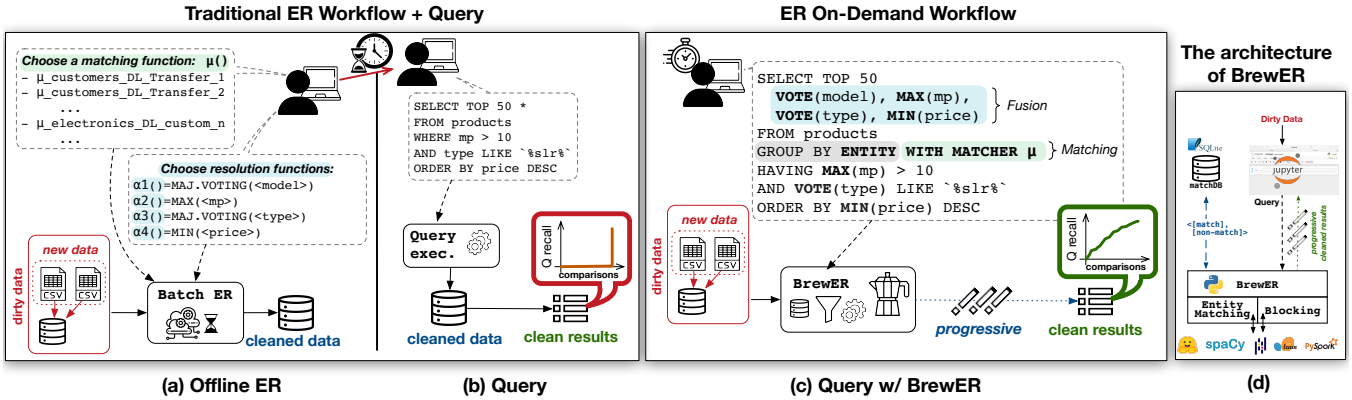

Figure 1: A traditional *batch* ER pipeline (a) needed to run the data scientist's query (b); the same query executed with BrewER on dirty data, but returning clean results, progressively (c); the architecture of BrewER (d).

pairs of records; thus, they do not guarantee to produce clean entities before the end of the ER process and do not allow the user to define a priority on the results.

After providing an overview of BrewER in Section 2, in Section 3 we demonstrate how BrewER can be used by data scientists to save a significant amount of resources while exploring the data or preparing it for their downstream tasks.

2 AN OVERVIEW OF BREWER

BrewER is designed to address two main requirements: *(i)* when a query is run on the dirty dataset, ER has to be performed only on the portion of the dataset needed for answering the query; *(ii)* the resulting entities have to be returned in a progressive fashion as soon as they are obtained, according to the priority defined by the user. The latter requirement is significantly challenging, since it demands to correctly sort the entities even before they are generated using data fusion, only relying on the original records that can produce them. In this section, we give an overview of how BrewER overcomes these challenges, while the detailed description of the algorithm is provided in the research paper [13].

Firstly, BrewER builds on the candidate pairs of records detected by the selected blocking function. BrewER considers the blocks of records determined by these candidates and performs a preliminary filtering of the blocks to maintain only the ones that can potentially lead to the generation of one or more entities appearing in the result of the query. This process is driven by the `HAVING` clause.

Say we want to query a camera dataset to retrieve the single-lens reflex (SLR) cameras with a resolution greater than 10 megapixels. In this case, the `HAVING` clause can be expressed as the conjunction of two conditions, defined on the attributes expressing the type (i.e., "`type LIKE '%slr%'`") and the resolution (i.e., "`mp > 10`") of the camera, respectively. An entity is emitted as part of the result only if the values of its attributes, obtained by aggregating the ones of its matching records, comply with these conditions. It is possible to know a priori if a block can generate entities able to satisfy the query or not. In fact, if none of the records in the block presents a resolution greater than 10 megapixels, it is impossible for the generated entities to present a value that satisfies the query (same consideration for the type). In this preliminary phase, BrewER goes through the blocks and maintains only the ones: *(i)* where every condition is independently satisfied by at least one record, if the conditions are *conjunctive* (i.e., connected by the `AND` operator); *(ii)* where at least one of the conditions is satisfied by at least one record, if they are *disjunctive* (i.e., connected by the `OR` operator).

Having detected the blocks to be preserved, BrewER inserts their records into a priority queue, with the priority of a record determined by its value for the attribute used in the `ORDER BY` clause (i.e., the *ordering key*), according to the specified ordering mode. For every record, we also keep track of its neighbors (i.e., the records appearing together with it in a candidate pair).

BrewER operates in an iterative way, checking at every iteration the head of the priority queue. If the head represents one of the original dirty records, it is necessary to perform ER on it, using the selected matcher to determine if its neighbors describe the same real-world entity. Every time a match is detected, this check is repeated recursively for the matching neighbor, obtaining an exhaustive cluster of matches. We prevent useless or redundant comparisons by tracking the pairs already evaluated and the records already assigned to an entity in a previous iteration. Then, BrewER removes from the priority queue the head and its matches, inserting the consolidated record representing the entity, whose attribute values are obtained using the selected aggregation functions (e.g., `MIN` for the price, `VOTE` for the model, etc.). In particular, the aggregated value for the ordering key determines the priority of the entity in the next iterations.

When the head of the priority queue is already a consolidated record (i.e., an entity obtained as described above), BrewER checks if it satisfies the query. In this case, the entity is emitted as part of the result, otherwise discarded. For the supported aggregation functions (i.e., `MIN`, `MAX`, `AVG`, and `VOTE`), it is impossible for the entities produced in the next iterations to present a higher priority (since none of the remaining dirty records presents a value for the ordering key greater than the one of the head), ensuring the correctness of the emission order for the resulting entities.

BrewER is implemented as a Python library. Figure 1d depicts its architecture and highlights that it can be seamlessly integrated

in Python workflows in Jupyter notebooks. BrewER is designed to be extensible: users can combine it with blockers and matchers from their favorite libraries. Although we focus on running queries directly on the dirty dataset, representing the case where BrewER has the greatest impact, it is possible to benefit from the previously executed queries, constructing on their results to further lighten the burden of ER. In fact, running a query allows us to collect the classifications performed by the matcher, which can be maintained in a SQLite database (the *matchDB* in Figure 1d) and used to prevent repeated comparisons and overlook resolved entity records. Thus, when running a new query using the same matcher, BrewER can exploit these hints to further improve the performance.

3 DEMONSTRATION SCENARIOS

We introduce the two representative scenarios that we will cover in the demonstration of BrewER[1]. For this purpose, we use a Python Jupyter notebook running locally on a laptop. This simple and familiar setting allows the users to fully focus on the core feature of BrewER: to benefit from this framework, it is enough to run on a dirty dataset a basic SQL SP query, with the only additional effort to specify how the dirty data must be cleaned. In each scenario, besides showing some example queries to quickly introduce the audience to the task at hand, we will allow the attendees to interact with the notebook: they will be able to edit the queries and to change the ER parameters (i.e., blocking, matching, and aggregation functions), exploring the results with the UI to appreciate the benefits of BrewER and to experience the impact of their choices on the ER process.

Datasets. We will provide users with a set of *dirty datasets*, composed of the reference datasets used in the research paper [13] (i.e., cameras, USB sticks, and organizations) plus several additional ones (e.g., an extended version of cameras and further datasets of commercial products from the Alaska benchmark [4] and multiple datasets from the Magellan Data Repository[2]). These datasets cover different domains and are highly heterogeneous in terms of cleanliness, number of attributes, and number of records, ranging from the 1K records of the smallest subset of USB sticks to the 29K records of the full camera dataset, on which the batch approach would take several hours to perform the entire cleaning process [13]. Each dataset comes with its ground truth, so the users will be able to assess the efficacy (precision/recall) of each step in the ER pipeline and the correctness of the results of the given queries.

Blockers and Matchers. We will also provide all needed building blocks for an ER pipeline: *(i)* a set of *blockers*, such as traditional methods [8], manually devised solutions, and advanced unsupervised (meta-)blocking techniques [6]; *(ii)* a set of *matchers*, including rule-based matchers and state-of-the-art binary classifiers based on machine learning [8] and deep learning [10] models.

3.1 Scenario 1: Querying Dirty Datasets

Ellen is a data analyst who needs to build a BI dashboard to analyze the price and the characteristics of cameras sold on several popular e-commerce stores. She has been asked to consider only SLR cameras, with a minimum resolution of 10 megapixels, and to focus on those with the lowest price among them. Further, she has been asked to extract the data with short notice and with a strict deadline for a company business meeting. Fortunately, it is very simple for Ellen, who knows SQL, to come up with a query to find the cheapest SLR cameras with at least 10 megapixels in resolution. Unfortunately, by issuing the SQL query on the dirty data, she obtains inconsistent results. In fact, considering the results depicted in Figure 2a, she notices the presence of duplicate records (e.g., the two records describing the Sony a5000 camera) and other data quality issues. Ellen already has some pre-trained matchers from previous projects on dirty product datasets that she can try on this new data—or she can exploit a pay-as-you-go LLM-based service (e.g., GPT-3) as a binary matcher. She also knows that she can employ some unsupervised blocking techniques to accelerate the ER process, but it would still take hours and a significant amount of resources to clean the entire dataset.

BrewER allows Ellen to overcome this situation and quickly obtain a consistent result. As depicted in Figure 2b, she can write the query in a dedicated text area of the notebook, generated using a simple Jupyter widget. Within the query, she declares the matcher to be used in the specific `GROUP BY ENTITY` clause and the aggregation functions for the attributes of interest (e.g., the minimum value for the price). Then, after clicking on the green Run button, the resulting cleaned entities will start to appear in the area below as soon as they are obtained, one by one in a progressive fashion, correctly sorted by ascending price. The execution will automatically stop after the emission of the number of entities required by Ellen.

As we can see in Figure 2b, the top cleaned entities are significantly different from the results obtained on the dirty data shown in Figure 2a. BrewER allows Ellen to analyze the produced entities through the UI, to better understand what happened during the ER process. In fact, by simply clicking on the row representing the entity, she can expand the table and visualize the matching records that were aggregated to produce it, understanding why an attribute presents a certain value. Through this feature, Ellen can discover the reasons behind the inconsistencies of the dirty results: for instance, the record determining the price of the cheapest model did not fulfil the condition defined on the type in the `WHERE` clause, thus was erroneously discarded.

In this scenario, after introducing some relevant cases to give the audience an insight on the benefits of BrewER for this task, the attendees will be able to explore the dirty datasets according to their interests, directly experiencing the key role of ER for obtaining consistent results (e.g., to be served as input for training a machine learning model) and how the choice of different aggregation criteria impacts on the results of the query.

3.2 Scenario 2: ER Pipeline Debugging

BrewER can significantly speed up not only the access to the cleaned results of queries issued on dirty datasets, but also the design of ER pipelines. In fact, as shown in the previous scenario, the choice of different aggregation functions, as well as different combinations of blocking and matching functions, can lead to the production of significantly different results.

[1]Please also see the accompanying demo video: https://youtu.be/nSd-_wAFkss
[2]https://sites.google.com/site/anhaidgroup/useful-stuff/the-magellan-data-repository

Figure 2: Demonstration scenarios: 1. Querying dirty datasets (a-b); 2. ER pipeline debugging (c-d).

Anna is a data engineer that employs BrewER to get quick insights into the goodness of the ER pipeline she is designing at a negligible cost compared to existing solutions. In fact, BrewER automatically and dynamically selects a portion of the data to be cleaned relevant for the task at hand—expressed through a query. By assessing the quality of the ER pipeline on that portion, Anna is more confident that it will be suited for the task compared to a pipeline tuned on a random sample of the data. Further, by having access to the entities in the result progressively as soon as they are produced, Anna can quickly interrupt the ER process as soon as she spots an issue, saving a significant amount of time and resources—with a batch approach she would have to wait until the end of the processing of the entire batch of data to spot the same issue.

BrewER perfectly supports exploratory top-k queries, allowing to debug the designed ER pipeline during the cleaning process in a stop-and-resume fashion. For instance, if a top-10 query produces the results in Figure 2c, Anna understands from the presence of duplicates the inaccuracy of her pipeline and she can immediately stop the cleaning process to solve the problem. The designed blocking technique was too aggressive and pruned some matches from the candidate set, preventing the complete resolution of some entities. After changing the blocking settings, Anna can run her query again: Figure 2d shows how the new setting solved the issue.

Since BrewER allows saving the status of the cleaning process in case of early termination, this time Anna can simply click on the Resume button to continue the cleaning process, running for instance a further top-k query for inspecting more entities, then resuming the process again for the complete emission of the results.

In this scenario, the attendees will be able to build an ER pipeline interactively, selecting different blockers and matchers to combine with a SQL query of interest. By inspecting the cleaned results returned progressively, they will be able to detect possible issues with the current pipeline, and in this case modify it on the fly and converge to a good solution.

In summary, BrewER is a highly interactive entity resolution system that allows users to issue ad-hoc queries to dirty data as if it was already cleaned.

REFERENCES

[1] Hotham Altwaijry et al. 2013. Query-Driven Approach to Entity Resolution. *PVLDB* 6, 14 (2013), 1846–1857.

[2] Hotham Altwaijry et al. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *PVLDB* 9, 3 (2015), 120–131.

[3] Vassilis Christophides et al. 2021. An Overview of End-to-End Entity Resolution for Big Data. *CSUR* 53, 6 (2021), 127:1–127:42.

[4] Valter Crescenzi et al. 2021. Alaska: A Flexible Benchmark for Data Integration Tasks. arXiv preprint arXiv:2101.11259.

[5] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration*. Morgan & Claypool Publishers.

[6] Luca Gagliardelli et al. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT*. OpenProceedings.org, 602–605.

[7] Mazhar Hameed and Felix Naumann. 2020. Data Preparation: A Survey of Commercial Tools. *SIGMOD Record* 49, 3 (2020), 18–29.

[8] Pradap Konda et al. 2016. Magellan: Toward Building Entity Matching Management Systems. *PVLDB* 9, 12 (2016), 1197–1208.

[9] Yuliang Li et al. 2020. Deep Entity Matching with Pre-Trained Language Models. *PVLDB* 14, 1 (2020), 50–60.

[10] Sidharth Mudgal et al. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. ACM, 19–34.

[11] Thorsten Papenbrock et al. 2015. Progressive Duplicate Detection. *TKDE* 27, 5 (2015), 1316–1329.

[12] Giovanni Simonini et al. 2018. Schema-agnostic Progressive Entity Resolution. In *ICDE*. IEEE Computer Society, 53–64.

[13] Giovanni Simonini et al. 2022. Entity Resolution On-Demand. *PVLDB* 15, 7 (2022), 1506–1518.

[14] Steven Euijong Whang et al. 2013. Pay-As-You-Go Entity Resolution. *TKDE* 25, 5 (2013), 1111–1124.