Entity Resolution On-Demand

Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, Felix Naumann[#]
Università degli Studi di Modena e Reggio Emilia, Italy, {name.surname}@unimore.it
[#]Hasso Plattner Institute, University of Potsdam, Germany, felix.naumann@hpi.de

Abstract

Entity Resolution (ER) aims to identify and merge records that refer to the same real-world entity. ER is typically employed as an expensive cleaning step on the entire data before consuming it. Yet, determining which entities are useful once cleaned depends solely on the user's application, which may need only a fraction of them. For instance, when dealing with Web data, we would like to be able to filter the entities of interest gathered from multiple sources without cleaning the entire, continuously-growing data. Similarly, when querying data lakes, we want to transform data on-demand and return the results in a timely manner—a fundamental requirement of ELT (*Extract-Load-Transform*) pipelines.

We propose *BrewER*, a framework to evaluate SQL SP queries on dirty data while progressively returning results as if they were issued on cleaned data. *BrewER* tries to focus the cleaning effort on one entity at a time, following an ORDER BY predicate. Thus, it inherently supports *top-k* and stop-and-resume execution. For a wide range of applications, a significant amount of resources can be saved. We exhaustively evaluate and show the efficacy of *BrewER* on four real-world datasets.

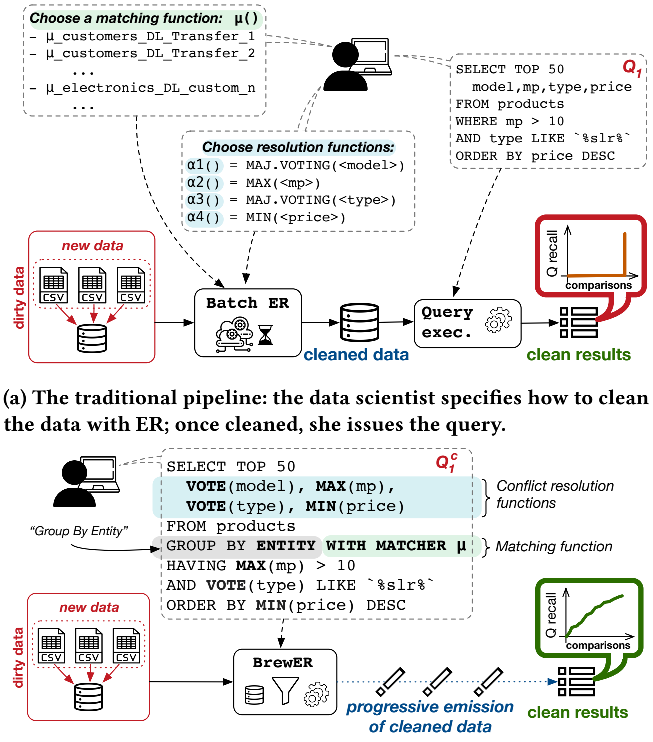
PVLDB Reference Format:
Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, Felix Naumann. Entity Resolution On-Demand. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

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

1 Introduction

Entity Resolution (ER) is the task of identifying and merging records in a dataset that refer to the same real-world entity. It is a fundamental operation for data cleaning and integration. A traditional use-case of ER is master-data-management, for instance to detect multiple representations of the same customer or product. More recently, ER has been employed to remove redundancy and errors from the data to significantly improve the final accuracy of machine learning models trained on it [21].

State-of-the-art ER methods compare a pair of records by using a binary *matching function* (a.k.a. *matcher*), exploiting machine/deep-/transfer learning [4, 15] or human-defined rules [7] to identify *matches*, i.e., record pairs that pertain to the same entity. To detect all matching records, in principle, all pairs must be compared. Hence, ER is very expensive, also due to the compute-intensive operations required by the matching functions, e.g., computation of string similarities or inference with complex neural networks.


Choose a matching function: µ()
- µ_customers_DL_Transfer_1
- µ_customers_DL_Transfer_2
...
- µ_electronics_DL_custom_n
...
Choose resolution functions:
α1() = MAJ.VOTING(<model>)
α2() = MAX(<mp>)
α3() = MAJ.VOTING(<type>)
α4() = MIN(<price>)
Q1
dirty data
new data
CSV
CSV
CSV
Batch ER
cleaned data
Query exec.
Q recall
comparisons
clean results

```
SELECT TOP 50
  model,mp,type,price
FROM products
WHERE mp > 10
AND type LIKE `%slr%`
ORDER BY price DESC
```

(a) The traditional pipeline: the data scientist specifies how to clean the data with ER; once cleaned, she issues the query.

Q_1^c
Conflict resolution functions
"Group By Entity"
Matching function
dirty data
new data
CSV
CSV
CSV
BrewER
progressive emission of cleaned data
Q recall
comparisons
clean results

```
SELECT TOP 50
  VOTE(model), MAX(mp),
  VOTE(type), MIN(price)
FROM products
GROUP BY ENTITY WITH MATCHER µ
HAVING MAX(mp) > 10
AND VOTE(type) LIKE `%slr%`
ORDER BY MIN(price) DESC
```

(b) The ER-on-demand pipeline: the data scientist specifies how to clean the data within the query.

Figure 1: Querying dirty datasets.

Matching is only the first step: once the matches forming an *entity cluster* have been identified, they have to be combined to remove inconsistencies in attribute values—this task is called *data fusion* [6], *merging* [5], or *consolidation* [14]. The function employed for removing the ambiguities in the values of each attribute is called *conflict resolution function* (or simply *resolution function*). Thus, to yield a unique representative record from a cluster of matching records, a resolution function is needed for each attribute of the schema. A resolution function specifies how to transform a multiset of attribute values in a unique *representative* value. For instance, a commonly employed resolution function is the *majority voting* [14], where the most frequent attribute value is selected as representative. We say that an entity is completely *resolved* when all of its records have been matched and their values have been consolidated to yield a unique representative record.

Limitations of existing approaches Traditionally, ER is employed as a data cleaning/preparation step *before* using the data. Yet, in many practical scenarios this might not be convenient:

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Example (The problem). *Ellen is a data scientist building a machine learning model to predict the price of SLR cameras.* (i) *She has limited time to add more data to her dataset and clean it; also, new data might be arriving periodically.* (ii) *The data might contain duplicates. For performing ER, she already has a matching function to choose (or more than one to try) for the data at hand—e.g., a machine learning model trained on the data she already has and/or exploiting transfer learning and/or ad-hoc rules—and she expresses rules for resolving the conflicts in the attribute values of the clusters of matching records (e.g.,* `AVG(price)`*,* `MAX(resolution)`*, etc.).* (iii) *She has business priorities: it is better to have clean data for expensive cameras than for inexpensive ones, and only modern cameras with a minimum resolution of* 10 megapixels *have to be considered—she can express this with a query.*

Figure 1a shows how Ellen specifies data extraction and cleaning, i.e., the SQL query that selects the entities she is interested in (priority on expensive products given by the ordering predicate), the matching function, and the resolution functions.

As in the example above, oftentimes practitioners (e.g., data scientists) have a specific task at hand characterized by:

- *An information need*: only a portion of the entities is relevant to their task; to clean the entire data just to run a selective query on it is a waste of resources;
- *Time constraints*: time can be limited and decisions based on the data have to be made quickly; time can be limiting when new data arrives or changes with high frequency and users want to quickly explore it with queries (e.g., *top-k* queries).

Example (Solution with traditional ER). *With a traditional ER framework, Ellen performs ER on the entire data at her disposal by applying matching and resolution functions. Once the entire data is cleaned, she can issue the query and explore or query the result (Figure 1a). In such a scenario, ER is the bottleneck due to:* (i) *its inherently quadratic complexity, which blocking and filtering techniques can only alleviate [29];* (ii) *the cost of matching functions, as state-of-the-art matchers involve expensive operations based on string-similarity measures computation [15] or deep neural network models [22].*

This approach is time-consuming. In fact, to check whether a new source contains useful data for her analysis with a query, she has to first clean it completely: all the entities are resolved and then filtered to produce results emitted in batch. Further, for debugging the ER pipeline with the data at hand (e.g., to check if the matcher she is employing is performing well for expensive SLR cameras), she cannot stop the ER process after receiving a handful of the entities to inspect and then resume the processing: those entities might not be relevant for the query or might be partially resolved, hence yielding incorrect results. Alternatively, she would have to manually select records from the dataset to test the ER pipeline, which is time-consuming as well.

Thus, it would be beneficial to *prioritize* the cleaning efforts on the entities according to their relevance for the practitioner's task.

Our contribution We propose the *BrewER* framework, which evaluates SQL `SP` (*Selection* and *Projection*) queries on dirty data, and returns results as if they were issued on cleaned data (as shown in Figure 1b). The main feature of *BrewER* is to perform ER *progressively*, guided by an `ORDER BY` clause, to incrementally return the most relevant results to the data scientist. *BrewER* avoids as much as possible matching and resolving entities that are not part of the final result, and it inherently supports *top-k* queries, as well as *stop-and-resume* execution. *BrewER* introduces a special "`GROUP BY ENTITY WITH MATCHER [matcher of choice]`" operator, which is interpreted as a "group by entity" statement, i.e., knowing that matching records should be grouped according to the selected matcher.

Example (ER-on-demand solution). *As shown in Figure 1b, Ellen just needs to rewrite her query (Q_1 in Figure 1a) by employing a special* `GROUP BY` *statement and moving the selection statements into the* `HAVING` *clause—predicating on each group, i.e., each entity. She also specifies the conflict resolution functions for ER within the SQL query, as aggregate functions. Notice that Q_1^c in Figure 1b and Q_1 in Figure 1a are equivalent if issued on cleaned data. Then, BrewER executes the query directly on the dirty data, applying ER progressively on the right portion of the data to yield correct results incrementally.*

Ellen receives the first entities in a fraction of the time required by existing ER frameworks. This allows her to explore new data without completely cleaning it, and to maximize the ER efforts on the entities she actually needs for her task. Furthermore, she can stop the execution at any time with the guarantee that the results produced so far are correct; then, she can resume the query evaluation at her need. Thus, she can inspect the result of ER process for entities of interest and debug it, if needed. Finally, BrewER keeps track of both executed comparisons and resolved entities, to avoid recomputing the same operations when multiple queries are issued on the same data.

Another example is the stock market trading scenario, where an entity matching algorithm on a high frequency financial news feed may have very limited time to match companies' records and yield useful information [42]. Further, typically only a subset of the entities is relevant for each operation and a priority may be defined, such as the trading volume or other financial metrics.

Moreover, our proposed approach is suitable for tackling a major challenge for data lake management systems [26]: to support on-demand extraction and cleaning as part of the integration pipeline and on-demand query answering. 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 to debug transformations [10].

The main contributions of *BrewER* are summarized in the following. We formalize *ER-on-demand*, where the goal is to progressively clean and emit entities satisfying queries issued directly on dirty datasets. We introduce an ER-on-demand algorithm for SQL `SP` queries and its variation for optimizing a special—yet common—case. We implement our algorithms in an open-source system called *BrewER*, that we exhaustively evaluate on four real-world datasets, showing its effectiveness.

The remainder of the paper is structured as follows. Section 2 examines related work. Section 3 provides preliminaries and formalizes the notion of *ER-on-demand*. Our algorithms and the workflow for executing ER-on-demand in *BrewER* are described in Section 4 and evaluated in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Entity Resolution (ER) has been a longstanding problem for data integration and cleaning [16]. See [9] for a recent survey on ER and its open challenges.

Progressive ER Madhavan et al. [24] firstly proposed a *progressive* (a.k.a. *pay-as-you-go*) data integration system, implemented for Google Base, to progressively integrate as much Web data as possible as it runs, given a limited amount of time and resources. The progressive approach has been employed for schema mapping [35] and ER [42]. In particular, to perform ER, oftentimes the resources are limited (e.g., computational power or human time to debug the ER pipeline), or the data has to be elaborated within a certain time to be valuable for the downstream application consuming it. To address this challenge, existing progressive ER methods [17, 19, 32, 37, 42] try to evaluate candidate matches by their likelihood of being actual matches (typically estimated through a proxy measure derived from a blocking strategy), so to discover as many matches as possible, as quickly as possible. Thus, a progressive ER method incrementally adds records to an entity cluster, which might remain incomplete until that the entire data has been processed. Hence, an SQL SP query cannot be simply issued at any time on the output of a progressive method. In fact, its result might be incorrect: the representative records may have values derived applying resolution functions on a partially identified entity cluster—thus yielding possibly incorrect values. *BrewER* aims at finding and resolving complete entity clusters, whose representative records satisfy an SQL SP query; moreover, it does that progressively, while following an ORDER BY predicate. Both *BrewER* and existing progressive methods deal with a *single-type entity* dataset at a time, i.e., a dataset with a unique schema. In Section 5.2.2, we further discuss and experimentally compare the progressive ER methods and *BrewER*.

Batch query-driven ER To the best of our knowledge, the closest works to our own are *QDA* [2] and *QuERy* [3]. *QDA* takes as input a single block B (see Section 3.1.3 for *blocking*) and a selection predicate P, and then analyzes which record pairs do not need to be compared to identify all entities in B that satisfy P. *QDA* is not designed for a progressive execution and to support ORDER BY clauses. Moreover, *QDA* requires to apply the resolution functions to the output of each match, hence it cannot support functions considering more than two values, such as AVG and VOTE. *BrewER* is blocking-agnostic (i.e., it is not limited to one block) and supports a wider class of resolution functions, including AVG and VOTE.

QuERy [3] supports SQL SPJ queries by introducing two special selection and join operators, which are called *polymorphic* as they accept as input not only records (as regular operators), but also objects representing blocks (called *sketches*). A sketch is a concise representation of all the potential representative records that a block may yield (i.e., all the possible outcomes of the cleaning of a block). For instance, a sketch employs a *range* data type to represent numerical attributes, and a *set of hashed values* to represent categorical attributes. To evaluate a query, *QuERy* builds a query tree (i.e., an execution plan) with the *polymorphic operators* for a dataset composed of clean records and sketches. When the query tree is executed, if a sketch reaches the topmost operator (i.e., passes all predicates), the corresponding block has to be cleaned since it can contain useful entities, otherwise it is discarded. The representative records (i.e., the cleaned entities) yielded from the cleaned block are then pushed back into the query tree to be re-evaluated—to check if they actually pass the predicates. *QuERy*'s main limitation is that its polymorphic operators work at the block level and cannot define the progressiveness of the ER execution within each block. *BrewER* could be employed within each block to reduce the number of comparisons that are evaluated and progressively pass resolved entities up to the query tree. Then, for a complete integration of *BrewER* and ORDER BY clauses, a *polymorphic sort* operator should be designed to compare and sort records and sketches. We do not investigate the integration of *QuERy* and *BrewER* in this paper.

Finally, the idea of *query-driven* ER has been explored also for answering *keyword queries* [36, 45]. In our previous preliminary work [33] we explored how to yield an approximate progressive result to a *keyword query* over a dirty dataset. *BrewER* guarantees an exact (i.e., not approximate) result and supports SQL SP queries.

3 Preliminaries

3.1 Entity Resolution Model

We consider a dirty dataset $\mathcal{D}$ with schema $\mathcal{D}[A_1, ..., A_m]$. Each attribute A_j has a domain (or *type*) T_{A_j} of values that the records can assume. A record $r \in \mathcal{D}$ is represented as a tuple $r = (id, r[A_1], ..., r[A_m])$, where id is a unique identified for each r, and $r[A_j] \in \{T_{A_j} \cup \emptyset\}$ is a projection to the value that the record assumes for the j-th attribute (null values are admitted).

Different records in a dataset that *belong* (i.e., refer) to the same real-world *entity* are called *matching records* (or simply *matches*). Entity Resolution (ER) aims to identify the disjoint clusters of records representing the entities of $\mathcal{D}$ and to synthesize a single *representative record* $\varepsilon = (id, \varepsilon[A_1], ..., \varepsilon[A_m])$ for each cluster—so to produce $\mathcal{D}^c$, the *cleaned* version of $\mathcal{D}$. In this paper, we adopt a standard ER framework [5, 14, 15] employing a *matching function* for determining the matching records that form entity clusters and *conflict resolution functions* for consolidating ambiguous attribute values within each cluster. We also support optional *blocking*, which is often employed to scale ER by avoiding comparing obvious non-matching pairs of records [29].

3.1.1 *Matching function*

A *matching function* (a.k.a. *matcher* [15]) is a binary function $\mu : \mathcal{D} \times \mathcal{D} \to \{True, False\}$ that takes as input two records, compares them, and decides whether they are matches or not. We do not assume the matching function to be transitive, i.e., if $\mu(r_x, r_y) = True$ and $\mu(r_y, r_z) = True$, it might be that $\mu(r_x, r_z) = False$. To design a transitive matching function is difficult in practice [5]. Yet, we consider the matches to be transitive, otherwise results would be inconsistent—e.g., declaring $\langle r_1, r_2 \rangle$ and $\langle r_2, r_3 \rangle$ as matching pairs while declaring $\langle r_1, r_3 \rangle$ as non-matching.

Our framework is matcher-agnostic: it can support any kind of matching function, such as a DNF of similarity join predicates on multiple attributes [18, 20], a human judging the pairs of records on a crowdsourcing platform [17], an unsupervised matcher based on generative models [43], or a complex deep learning model exploiting pre-trained language models [22] or transfer learning [23]. Our framework allows indicating the matching function for a particular ER task within an SQL query denoted by μ^Q.

3.1.2 *Conflict resolution functions*

The *conflict resolution functions* (or simply *resolution functions*) transform a cluster of records into a single record. Records belonging to the same entity cluster often have inconsistent values for their attributes (e.g., camera records referring to the same entity may have different prices, names, etc.). Thus, for each attribute, a resolution function is applied to remove conflicts: it takes as input a multiset of attribute values and returns a single value.

We declare a resolution function for an attribute A_j through an SQL *aggregate function* α_j. Given a cluster of records $\mathcal{E} = \{r_1, ..., r_k\}$ representing a single entity, each α_j takes as input the list of values that A_j assumes in those records and returns a single value $\varepsilon[A_j] \in T_{A_j}$. The aggregate functions supported in our framework are: MIN, MAX, AVG, and user-defined bounded aggregations (defined in Section 3.1.5), such as MEDIAN and VOTE (a.k.a. *majority voting*). The choice of this set of functions was driven by two considerations: *(i)* they cover most real-world use cases; *(ii)* they can be naturally declared as part of SQL queries.

3.1.3 Blocking Comparing all pairs of records in $\mathcal{D}$ has a quadratic complexity and, typically, the matching function is expensive to compute: state-of-the-art functions involve either string-similarity computation [15] or deep neural network models [22]. To overcome this problem, *blocking* is employed to partition $\mathcal{D}$ in *blocks*, i.e., partitions of records, and limits the all-pairs comparison to records within each block [8, 29]. Given a record r, we call *candidate set* the set of records that appear together with it in blocks—each record in it is called *candidate match* or simply *candidate*. Our framework is blocking-agnostic, meaning that any blocking strategy (or even no blocking strategy) can be employed, including unsupervised techniques [27, 28, 40], as we experimentally show in Section 5.5.

3.1.4 Query-agnostic (traditional) ER algorithms We call *traditional* ER algorithms those algorithms that employ matching and conflict resolution functions in a query-agnostic way, i.e., without filtering for a specific part of the data and without order preferences. Traditional ER algorithms may follow any match-resolve strategy, such as *batch ER* [8], which performs blocking, applies the matching function in random order and finally performs the conflict resolution, or *progressive ER* [17, 19, 32, 37, 42], presented in Section 2.

3.1.5 Record bounds and bounded aggregation Given a record r, its candidate set, and an aggregate function α_j for a numeric attribute A_j, we call *lower bound* and *upper bound* of r the minimum and maximum value that the entity to which r belongs can assume for the attribute A_j, respectively.

When an aggregate function α_j yields a consolidated value $\varepsilon[A_j]$ for a set of matches $\mathcal{E}$ that is always $\varepsilon[A_j] \in [min(V^{\mathcal{E}}_{A_j}), max(V^{\mathcal{E}}_{A_j})]$, we call it *bounded aggregation*; otherwise, we call it *unbounded aggregation* (e.g., SUM, which can produce a final value $\varepsilon[A_j]$ that is greater than the maximum value of $V^{\mathcal{E}}_{A_j}$). We consider only MIN, MAX, AVG, MEDIAN and VOTE for our examples or experiments, but *BrewER* inherently supports any user-defined bounded aggregation (UDF). We do not study unbounded aggregation.

Further, we distinguish between *fixed* and *free* bounded aggregate functions for numeric attributes. Given a numeric attribute A_j, a *fixed* aggregate function can yield only values $\varepsilon[A_j] \in V^{\mathcal{E}}_{A_j}$, i.e., the value that the resolved entity ε assumes in A_j is among the values that its records assume for A_j. Examples of fixed aggregate functions are MIN, MAX, and VOTE. A *free* aggregate function, on the other hand, can generate $\varepsilon[A_j] \in [min(V^{\mathcal{E}}_{A_j}), max(V^{\mathcal{E}}_{A_j})]$, i.e., the value assumed by A_j in the resolved entity ε can be a *new* value not among the values that its records assume in A_j, yet bounded from them. An example of free aggregate function is AVG.

```
SELECT [TOP k] ⟨αj(Aj)⟩
FROM 𝒟
[WHERE φ]
GROUP BY ENTITY WITH MATCHER μ
[HAVING ⟨αj(Aj)⟩ {LIKE|IN|<|≤|>|≥|=} const⟩
[ORDER BY αj(Aj) [ASC|DESC]]
```

Figure 2: Query syntax in *BrewER*.

3.2 ER-on-demand Model

We first introduce the type of queries supported by our framework, then describe the characteristics of our ER-on-demand algorithm.

3.2.1 Supported queries BrewER supports SQL SP queries with an ordering predicate. Like existing progressive methods (Section 2) and most state-of-the-art ER frameworks, such as Magellan [15], JedAI [28, 31], DeepMatcher [25], Ditto [22], or Tamr [39], we focus only on *single-type entity* datasets (e.g., electronic goods). That is, *BrewER* applies ER on dirty datasets that can be represented with a unique schema. In such a scenario, SP predicates and ordering predicates are sufficient for users to express their information needs and priorities, respectively. The JOIN operator would be useful when dealing with *multi-type entity* datasets, i.e., when ER is applied jointly on multiple dirty datasets with different schemata [41]—we will investigate this dimension in future work.

Figure 2 presents the syntax supported in *BrewER*. The GROUP BY ENTITY clause declares that the query should return results aggregated by entities: in *BrewER* "ENTITY" is a reserved word and must be combined with a matching function μ. The *resolution functions* are specified in the SELECT clause as a list $\langle \alpha_j(A_j) \rangle$, where each element is an aggregate function α_j combined to an attribute A_j. The WHERE clause serves just as a filter applied directly to the initial dirty data $\sigma_\varphi(\mathcal{D})$, i.e., it filters records *before* the cleaning. From now on, for simplicity, we omit φ in our discussion, being a filter applied to the dirty data independently of the ER process. The filtering on the entities *after* the ER process is defined in the HAVING clause, which predicates over aggregate values of the groups (i.e., values of the entities). In the HAVING clause, we currently support numeric comparisons (<, ≤, >, ≥, =) for numeric attributes and dates, and string comparisons (=, LIKE, IN) for textual attributes. Finally, we currently support ordering for a single attribute.

In *BrewER*, a valid query Q^c has the structure presented above in Figure 2. With Q we denote the corresponding query for cleaned data, where: *(i)* the GROUP BY statement is removed, *(ii)* the HAVING predicates are expressed as WHERE conditions, *(iii)* no aggregation is specified in the selection statement, and *(iv)* there is an ORDER BY condition on the same attribute of Q^c. In practice, Q^c issued on a dirty data $\mathcal{D}$ yields the same results of Q issued on the cleaned data $\mathcal{D}^c$ (cleaned with the same matcher and resolution functions of Q^c). Examples of Q and Q^c are shown in Figure 1.

The ORDER BY clause allows to benefit from the progressive emission of the entities. Yet, a user can define a query without using it. In such a case, *BrewER* chooses a random (even textual) attribute for the ordering. Similarly, a user may express a query without a selection predicate. In this case, all entities are emitted progressively, following the ORDER BY clause, in a *pay-as-you-go* fashion.

3.2.2 ER-on-demand algorithm Given a (dirty) dataset $\mathcal{D}$ and a *BrewER* SQL query Q^c, we want an algorithm to guarantee a correct partial result when the execution is terminated early. That algorithm should perform ER progressively, following the query

Q^c and its ORDER BY clause, and not up-front on the entire data—i.e., *on-demand*. In fact, traditional ER algorithms do not guarantee the correct result in case of early termination: some of the emitted entities may not be completely resolved, thus possibly sorted in the wrong order and/or erroneously retained/discarded due to unresolved inconsistencies of their values. Thus, an *ER-on-demand algorithm* allows to significantly save computational resources and time if the user wants to stop the execution after inspecting the first k emitted records. Further, *top-k* queries and *stop-and-resume* execution are inherently supported.

We now formally define an ER-on-demand algorithm. We denote with $Q(\mathcal{D}^c)$ the results of an SQL SP query Q with an ORDER BY clause issued on $\mathcal{D}^c$, which is the cleaned version of the dirty dataset $\mathcal{D}$ obtained by using a traditional ER algorithm. The corresponding valid version of the query Q in *BrewER* is Q^c, i.e., a query written according to the syntax in Figure 2.

Definition 1. Given a dirty dataset $\mathcal{D}$ and an SQL SP query Q with an ORDER BY clause, an *ER-on-demand* algorithm to evaluate Q^c (the *BrewER* version of Q) on $\mathcal{D}$ satisfies the following conditions:

(1) *Correctness*: Let t be an arbitrary target time at which the results are needed: $Q^c_t(\mathcal{D}) \subseteq Q(\mathcal{D}^c)$ and $Q^c_t(\mathcal{D})$ is correctly sorted according to the ORDER BY condition. Typically, t is significantly smaller than the time needed to produce $\mathcal{D}^c$ in its entirety.
(2) *Monotonicity*: $Q^c_{t_1}(\mathcal{D}) \subseteq Q^c_{t_2}(\mathcal{D})$ for any $t_1 < t_2$.
(3) *Equivalence*: If the traditional ER algorithm and the ER-on-demand algorithm both have enough time to terminate, they produce the same results for the query, i.e., $Q^c_{t_\infty}(\mathcal{D}) \equiv Q(\mathcal{D}^c)$.

Notice that a traditional ER algorithm does not satisfy Conditions 1 and 2. Let us consider a progressive ER algorithm, a query Q^c, and a dirty dataset $\mathcal{D}$, and assume that the representative record ε of a cluster of matching records $\{r_1, r_2, r_3\}$ does not satisfy Q^c. Let us further assume that if we resolve only $\{r_1, r_2\}$ the resulting (incomplete) representative record ε' satisfies Q^c (a common scenario with real-world data). Now, say that after running for a time t_1, the progressive algorithm identifies only two matches: $\{r_1, r_2\}$. So, if we interrupt the execution at t_1 and issue Q^c, $\varepsilon_{t_1} \equiv \varepsilon'$ satisfies Q^c and is erroneously emitted as a result. Hence, $Q^c_{t_1}(\mathcal{D}) \not\subseteq Q(\mathcal{D}^c)$ and thus *correctness* is not satisfied. Then, if we run the progressive algorithm until the discovery of the remaining match r_3 (i.e., at time $t_2 > t_1$) and issue again Q^c, $\varepsilon_{t_2} \equiv \varepsilon$ does not satisfy Q^c. Hence, $Q^c_{t_1}(\mathcal{D}) \not\subseteq Q^c_{t_2}(\mathcal{D})$ and thus *monotonicity* is not satisfied.

4 ER-on-demand with *BrewER*

The high-level design of *BrewER* is depicted in Figure 3. *BrewER* is an extensible framework, enabling users to plug-in their favorite libraries for binary matching functions (e.g., DeepMatcher [25], Ditto [22], etc.) and blocking (e.g., Magellan [15]). Then, *BrewER* takes care of the ER-on-demand execution of the user's query, as explained in the remainder of this section. To avoid re-comparing candidate pairs with expensive matching functions, the lists of matching and non-matching records are maintained in a database, for each matching function employed by the users—if the matching function changes, the matches change as well. Thus, *BrewER* can retrieve, exploit and update those lists when executing a new query. Users can also choose to store only final resolved entities to save space—in this case, the resolution functions cannot change across queries.

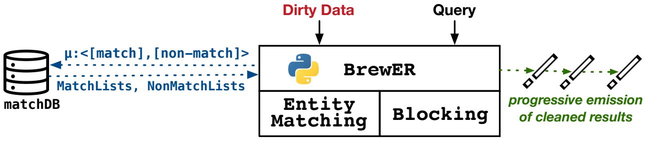


Figure 3: *BrewER* overview.

At the core of *BrewER* lies our ER-on-demand algorithm, which we introduce in Section 4.1 and formally present in Section 4.2.

4.1 Algorithm overview

The ideal ER-on-demand algorithm would start by cleaning the first entity in $\mathcal{D}$ that should be emitted for the query Q issued on cleaned data $\mathcal{D}^c$, and emit that entity as its first result. Then, it would start over with the *next entity that should be cleaned and emitted* for Q, and so forth.

To design a practical ER-on-demand algorithm, we first define *seed records*, which are the records that guide our algorithm in seeking the *next entity that should be cleaned and emitted.* We also define a *seed query*, i.e., the query to extract the seed records. The general idea is to insert the seed records and their candidate matches in a priority queue, which is exploited for ordering the entities (to which the seed records belong) that satisfy Q^c, while cleaning them.

4.1.1 Seed query We first consider the case where fixed aggregate functions are employed, then we complete the discussion with free aggregate functions.

Considering a valid *BrewER* query Q^c employing only fixed aggregate functions and its corresponding query Q for cleaned data, we observe that if all the records in a cluster of matches do not satisfy any of the selection conditions of Q, then that cluster cannot yield an entity that is part of the result of Q^c. Thus, given a set of candidate matches, if none of the involved records satisfies at least one of the selection conditions of Q, those comparisons can be avoided. On the other hand, if there exists at least one record r_s satisfying one of the selection conditions of Q, those comparisons should be considered. We call r_s a *seed record* (or simply *seed*).

Consider now a valid *BrewER* query Q^c with only a free aggregate function, e.g., a query with the condition HAVING AVG(mp) = 10 issued on the dataset of Figure 5a. It may occur that no record satisfies the corresponding selection condition in Q, i.e., WHERE mp = 10, but a cluster of matching records actually has an average of values for the attribute mp equal to 10. The process for fixed aggregate functions would not identify a seed and we would miss a correct result. Yet, a free aggregation is also a bounded aggregation, so we can discard any record r_i if $\theta_{\mathrm{mp}} \notin [min(V^{C_i}_{\mathrm{mp}}), max(V^{C_i}_{\mathrm{mp}})]$, where $V^{C_i}_{\mathrm{mp}}$ is the set of values assumed by the candidates C_i of r_i (with $r_i \in C_i$) for the attribute mp, and θ_{mp} is the parameter of the selection clause (i.e., $\theta_{\mathrm{mp}} = 10$). This is possible because the entity to which r_i belongs cannot assume a value for mp outside the range $[min(V^{C_i}_{mp}), max(V^{C_i}_{mp})]$. Thus, for a free aggregate function on an attribute A_j, r_i is a seed record if: *(i)* for the equality operator, $\theta_{A_j} \in [min(V^{C_i}_{A_j}), max(V^{C_i}_{A_j})]$; *(ii)* for > (or <) inequality operator, $r_i[A_j] > \theta_{A_j}$ (or $r_i[A_j] < \theta_{A_j}$).

The *seed query* Q^{seed} from a *BrewER* query Q^c yields the *seed set*, i.e., the set of all seed records. It is obtained with a projection of all attributes of $\mathcal{D}$ and a selection composed of the logical disjunction of the *set of basic predicates* $\mathcal{P}$ derived from the HAVING clause of Q^c as follows. If the HAVING clause of Q^c involves a fixed aggregate function, its corresponding selection predicate ϕ of Q is added to $\mathcal{P}$. If the HAVING clause of Q^c involves a free aggregate function for the attribute A_j: *(i)* for the equality operator, we add to $\mathcal{P}$ a predicate ϕ of the form θ_{A_j} BETWEEN$(min(V^{C_i}_{A_j}), max(V^{C_i}_{A_j}))$; *(ii)* for > (or <) inequality operator, we add to $\mathcal{P}$ a predicate ϕ of the form $A_j > \theta_{A_j}$ (or $A_j < \theta_{A_j}$). No ordering is needed for the seed query. We use the logical disjunction, even for conjunctive queries (e.g., Figure 4), since the seed records can match and yield any entity that satisfies Q^c, although each of them individually may not satisfy all predicates of Q. Hence, the seed query is defined as $Q^{seed} = \sigma_{\bigvee \phi \in \mathcal{P}}(\mathcal{D})$.

Seeds and blocking When blocking is employed[1], *BrewER* computes the transitive closure of all the candidate pairs by merging blocks that overlap and stores the resulting connected components of records in an auxiliary data structure, called *component list*. A *block index* is maintained as well: an index where each connected component is a key that points to the lists of blocks that has been merged to yield that component. Then, *BrewER* removes from the *component list* all the components that do not contain any seed record: they cannot yield any result for Q^c. Moreover, the *set of basic predicates* $\mathcal{P}$ (defined above) can be exploited to filter out further components that do not lead to any useful entity for answering Q^c. Consider for instance Q^c_1 of Figure 1: if a component does not contain any record that has "slr" in its type attribute, then it cannot yield any entity satisfying Q^c_1—even if the predicate on the megapixels is satisfied, since the two conditions are conjunctive. Hence, for conjunctive queries, *BrewER* builds a query Q^b_i for each i-th predicate in $\mathcal{P}$. So, if a Q^b_i applied to a component returns an empty set, that component is discarded. Finally, the *block index* is employed to retrieve the blocks that have been retained in the *component list*, which are the only blocks considered by *BrewER* for the processing.

4.1.2 *Ordering entities while resolving them*

The desired ER-on-demand algorithm is an iterative algorithm capable of identifying the *next entity that should be cleaned and emitted* as soon as possible, i.e., with the fewest calls to the matching function μ^Q. This can be approximated by determining the lower/upper bound of the value of the ORDER BY clause for the entity, so to define whether it should be emitted before or after all the other entities.

In fact, each entity has as ordering value the highest or lowest value of its records (see Section 3.1.5). For instance, consider the first entity that has to be emitted: its final value might be determined by one or more records that are not in the seed set and that are higher/lower than all the values of the seeds. Hence, the comparisons cannot involve only records in the seed set, but a broader set, composed of both the seeds and their candidate matches, must be considered. Each element of that set can be inserted in a priority queue (according to the value of each record); then, the algorithm can iteratively evaluate the head (i.e., the record with the current highest/lowest value) and determine its bound. Thus, as soon as a record has a lower/upper bound that is greater/lower than or equal to the head of the queue, it can be emitted.

```
SELECT TOP 50                                    Q_1
  model,mp,type,price
FROM products
WHERE mp > 10
AND type LIKE `%slr%`
ORDER BY price DESC
```

(a) Q_1

```
SELECT TOP 50                                    Q_1^c
  VOTE(model), MAX(mp),
  VOTE(type), MIN(price)
FROM products
GROUP BY ENTITY WITH MATCHER μ
HAVING MAX(mp) > 10
AND VOTE(type) LIKE `%slr%`
ORDER BY MIN(price) DESC
```

(b) Q^c_1

```
SELECT *                                         Q_1^seed
FROM products
WHERE mp > 10
OR type LIKE `%slr%`
```

(c) Q^{seed}_1

Figure 4: The query to be issued on clean data (a), a valid *BrewER* query to be issued on dirty data (b) and its seed query (c), following the example of Figure 1.

	id	brand	model	type	mp	price
ε_1	r_1	canon	eos 400d	dslr	10.1	185.00
	r_2	eos canon	rebel xti	reflex	1.01	115.00
	r_3	canon	eos 400d	dslr	10.1	165.00
ε_2	r_4	nikon	d-200	–	–	150.00
	r_5	nikon	d200	dslr	10.2	130.00
ε_3	r_6	nikon	coolpix	compct	8.0	90.00
ε_4	r_7	canon nikon olympus	olypus-1	dslr	–	90.00

(a) A dirty dataset.

ORDER BY AVG(PRICE) DESC		(VOTE)	(VOTE)	(MAX)	**(AVG)**
	id	model	type	mp	price
	ε_1	eos 400d	dslr	10.1	155.00
	ε_2	d-200	dslr	10.2	140.00

(b) The result for Q^c_1 of Figure 4b, with α(price) ≡ AVG(price) instead of MIN(price), issued on the dirty dataset.

ORDER BY MIN(PRICE) DESC		(VOTE)	(VOTE)	(MAX)	**(MIN)**
	id	model	type	mp	price
	ε_2	d-200	dslr	10.2	130.00
	ε_1	eos 400d	dslr	10.1	115.00

(c) The result for Q^c_1 of Figure 4b issued on the dirty dataset.

Figure 5: A dirty dataset (a) and clean query results (b and c) for different aggregate functions on the attribute price.

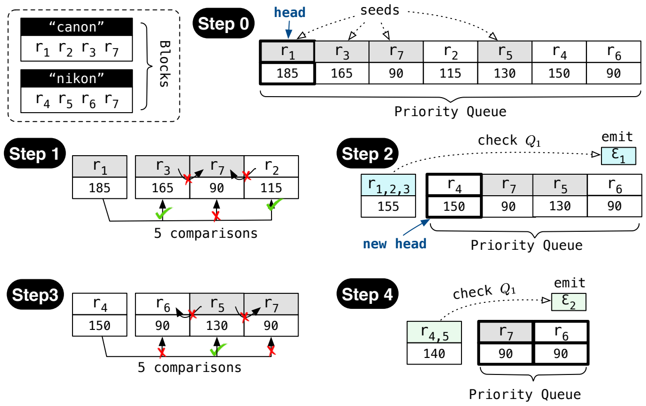


Figure 6: Example of *BrewER* in the AVG/DESC case.

Example 1. *Figure 5a shows a dirty dataset that Ellen (the data scientist) wants to query with Q^c_1 of Figure 4. Figures 5b and 5c report the results of the query employing* AVG *and* MIN *aggregate functions on* price*, respectively. Ellen is employing* AVG(price) *and a blocking strategy that inserts in the same block all the records that share at least one token in* brand *(the blocks are at the top of Figure 6).*

*With a traditional ER approach, 12 pairs of records are compared (6 pairs from the block "*canon*" and 6 from the block "*nikon*") to clean the dataset and obtain the first results for the query Q^c_1. Hence, Ellen*

[1] If blocking is not employed, the entire dataset is still considered as a single block.

has to wait the time for the complete ER for even just the first correct result, namely ε_1. Instead, if she employs BrewER, ε_1 is returned after just 5 comparisons, as explained next.

First, the seed query of Q_1^c is generated (Figure 4c). The seed records $\{r_1, r_3, r_5, r_7\}$ are then extracted with Q_1^{seed} and processed by BrewER with their candidates r_2, r_4 and r_6 (from the blocks). The process is shown in Figure 6. A priority queue is populated with the seeds and their candidates, according to their ordering value (Step 0). We employ a Max Priority Queue *because the required ordering is descending, otherwise it should be a* Min Priority Queue. *The head of the queue potentially belongs to the entity with the highest price, i.e., the first entity to be emitted. All records that match with the head have to be identified to compute the average of the ordering values. Thus, as shown in Step 1, the head record is compared to all its candidates. For each matching record of the head, BrewER recursively compares also its candidates (if not already compared), so to obtain a final entity cluster, as the matching function might not be transitive. At Step 2, the entity ε_1 to which the head r_1 belongs can be bounded, i.e., its ordering value is now known. At this stage, if the value of ε_1 is greater or equal than the new head record value of the queue (i.e., r_4), ε_1 can be emitted—no other entity can have a greater ordering value. Yet, we need to check whether ε_1 actually satisfies Q_1^c. Hence, the filtering predicates of Q_1 are applied. A total of 5 comparisons have been executed to provide Ellen the first correct result for her query, in its correct order. Then, in Steps 3 and 4 the same process is repeated.*

4.2 The *BrewER* algorithm

The *BrewER* algorithm can handle any combination of *bounded aggregations* (see Section 3.1.5) and ordering (ASC/DESC). Here, for ease of presentation, we consider the case of entities emitted in non-increasing order of a given attribute, i.e., ORDER BY $\alpha(\cdot)$ DESC, and $\alpha(\cdot) \equiv$ MAX$(\cdot)$ as resolution function of the ordering value. With other aggregate functions (AVG, VOTE, etc.) the presented algorithm does not change. Also, when we refer to the *value* (or *ordering value*) of a record or of an entity, we mean the value assumed by the ORDER BY attribute—these are the only values of the records that are relevant to the algorithm, i.e., the values that can affect the order of emission of the entities.

4.2.1 Algorithm description As input, the *BrewER* algorithm (Algorithm 1) takes the dirty dataset $\mathcal{D}$, a query Q^c, and the lists *CandLists*, *MatchLists* and *NonMatchLists*. *CandLists* is a list whose i-th element is a list itself containing all candidate matches of the i-th record in $\mathcal{D}$ generated with blocking. If blocking is not employed, *BrewER* considers the all-pairs comparison scenario. With large datasets, it is always preferable to employ blocking in order to avoid the quadratic complexity. Thus, we assume that *CandLists* fits in memory (as it does for all the experiments in Section 5). Alongside *CandLists*, the two complementary lists of lists *MatchLists* and *NonMatchLists* keep track of the matches and non-matches that have already been compared with μ^Q, respectively—they have the same size of *CandLists*, but they can be implemented with lists of bit arrays, thus accounting for a low memory overhead. We use the following notation: *MatchLists*[i][j] (or *NonMatchLists*[i][j]) is the flag indicating the match (non-match) between r_i and its candidate matching record r_j. Each element in the lists is initialized to zero, i.e., *false*. *MatchLists* and *NonMatchLists* keep track of

Algorithm 1: *BrewER* algorithm with $\alpha \equiv MAX(\cdot)$ and DESC ordering.

```
   Input: $\mathcal{D}$; $Q^c$; CandLists; MatchLists; NonMatchLists.
   Output: The incremental emission of the resolved entities.
 1 $Q^{seed} \leftarrow$ get the seed query for $Q^c$
 2 $Seeds \leftarrow \Pi_{id}\, Q^{seed}(\mathcal{D})$                       // seed record ids
 3 $\mathcal{I} \leftarrow Seeds \bigcup \{j \in CandLists[i] \mid i \in Seeds\}$
 4 $MatchSet \leftarrow \emptyset$                                        // empty set
 5 $EntityMap \leftarrow Map(\emptyset)$                            // empty hash table
 6 $PQueue \leftarrow maxHeap(\emptyset)$                            // priority queue
 7 forall $i \in \mathcal{I}$ do
 8     $r_i \leftarrow \sigma_{id=i}(\mathcal{D})$
 9     $val \leftarrow r_i[Q^c.orderByAttribute]$
10     PQueue.insert(i, val)
11 while $PQueue \neq \emptyset$ do
12     $i \leftarrow$ PQueue.extractHeadElement()
13     if EntityMap.hasKey(i) then
14         emit EntityMap.get(i)                        // then go to Line 11
15     if $i \in MatchSet$ then
16         continue                                           // go to Line 11
17     $E \leftarrow \emptyset$                   // initialize the entity cluster set
18     $\mathcal{R} \leftarrow \emptyset$                   // initialize the records to check
19     $onlySeeds \leftarrow True$                                 // consider seeds only
20     $recordID \leftarrow i$
21     matchingProcedure()
22     if $E \equiv \emptyset$ and $i \notin Seeds$ then
23         continue               // no matching seeds: go to Line 11
24     $E \leftarrow E \bigcup \{i\}$
25     $onlySeeds \leftarrow False$                     // consider also non-seeds
26     while $\mathcal{R} \neq \emptyset$ do
27         $recordID \leftarrow$ extract an id from $\mathcal{R}$
28         matchingProcedure()
29     $\varepsilon_i \leftarrow \widetilde{Q}^c(\sigma_{id \in E}(\mathcal{D}))$                        // a resolved entity
30     if $\varepsilon_i \neq \emptyset$ then
31         $EntityMap.add(i, \varepsilon_i)$
32         $PQueue.insert(i, \varepsilon_i.val)$
```

Procedure 1: matchingProcedure

```
   Input: Procedure 1 has visibility of all variables in Algorithm 1
 1 $Candidates \leftarrow CandLists[recordID]$
 2 for (p = 0; p++; p < len(Candidates)) do
       /* p is the position of the current candidate in the
          candidate list of recordID                            */
 3     $candidateID \leftarrow Candidates[p]$
 4     if $onlySeeds{=}{=}True \land candidateID \notin Seeds$ then
 5         break                       // continue only for non-seeds
 6     else if $candidateID \in E$ then
 7         continue                                         // go to Line 2
 8     else if $MatchLists[recordID][p]$ then
 9         $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{candidateID\}$
10         $E \leftarrow E \bigcup \{candidateID\}$
11     else if $NonMatchLists[recordID][p]$ then
12         continue                                         // go to Line 2
13     else if $\mu^Q(\mathcal{D}[i], \mathcal{D}[candidateID])$ then
14         $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{candidateID\}$
15         $E \leftarrow E \bigcup \{candidateID\}$
16         $MatchLists[recordID][p] \leftarrow True$
17         $p' \leftarrow$ get position of recordID in $CandLists[candidateID]$
18         $MatchLists[candidateID][p'] \leftarrow True$
19     else
20         $NonMatchLists[recordID][p] \leftarrow True$
21         $p' \leftarrow$ get position of recordID in $CandLists[candidateID]$
22         $NonMatchLists[candidateID][p'] \leftarrow True$
23 $MatchSet \leftarrow MatchSet \bigcup \{recordID\}$
```

matching/non-matching pairs among multiple query executions, avoiding the redundant comparisons.

As output, Algorithm 1 emits the resolved entities incrementally, according to the ORDER BY clause of Q^c. In the following, we describe the details of the algorithm.

First, the seed query Q^{seed} is derived from Q^c (Line 1), as defined in Section 4.1.1, and issued on $\mathcal{D}$ to obtain the seed records and to initialize the seed id set *Seeds* (Line 2). This set is then merged with all the candidate matches of the seeds in Line 3. *MatchSet* (Line 4) is an empty set used to keep track of the records that have already been positively matched. *EntityMap* (Line 5) is a *map* data structure (e.g., a *hash table*) that stores *key-value* pairs: the *key* is the *id* of a representative record of an entity, and the value is the resolved entity satisfying Q^c. In practice, *EntityMap* stores the entities that have been resolved and ready to be emitted when their turn comes.

A *max heap*—or *min heap* in the case of ASC ordering—is initialized in Line 6 and populated with the seeds and their candidates, serving as *priority queue* (Lines 7-10). The basic idea is to iteratively extract the head element from the priority queue, resolve its corresponding entity ε, and insert ε into the priority queue with its consolidated ordering value. Thus, every time that the head of the priority queue is a resolved entity, it can be emitted: all the other records and entities in the queue have an equal or lower value.

The iteration on the priority queue starts in Line 11. Notice that the priority queue stores only record *id*s associated to their ordering values. The head element i of the priority queue is extracted in Line 12 and then:

- (Line 13) If i is the representative *id* of an entity that was completely resolved in a previous iteration, that entity has the highest value of all possible remaining entities/records in the priority queue. Thus, it is emitted in Line 14.
- (Line 15) If i is not a representative record, but has already been matched in a previous iteration with at least one other record, the iteration continues and the next element in the priority queue is considered (Line 16).
- (Lines 17-32) Otherwise, the record r_i corresponding to i is compared to its candidates to completely resolve its entity or discarded, as explained in the following.

An entity should be emitted only if it is derived from at least one seed record (otherwise it does not satisfy Q^c); so, the first comparisons to be performed are those to ensure that r_i matches a seed record—if it is not a seed record itself. This is checked in Lines 19-23. If r_i matches a seed (or it is a seed itself), then all remaining candidates of r_i are considered, and the entity ε_i is completely resolved (Lines 25-29). *BrewER* tries to find also the matches of each match recursively, starting from r_i. To do so, it recursively inserts the *id*s of matches in $\mathcal{R}$.

The actual comparisons are verified by calling the matchingProcedure (Procedure 1), which also updates $\mathcal{R}$ with newly discovered matches. In the first call to matchingProcedure (Line 21), the flag *onlySeeds* is set to true, so to check only seed records. The set E collects the *id*s of the matches of r_i. After the first call to Procedure 1, if E is empty, then no seed matches r_i and the execution is interrupted (Line 23). The other calls to matchingProcedure have the *onlySeeds* flag equal to false instead.

So, at the end of the while loop on $\mathcal{R}$ of Line 26 of Algorithm 1, all the matching records of r_i are in E. The resolution functions can now be applied to that cluster of records. To do so, Algorithm 1 issues the query $\widetilde{Q}^c$ on the set of identified matching records, i.e., $\sigma_{id \in E}(\mathcal{D})$ (Line 29). $\widetilde{Q}^c$ denotes the query Q^c without the matching function (μ^Q) invocations: at this stage of the algorithm the query is performed against a cluster of known matching records E, i.e., it can assume that the GROUP BY ENTITY yields only one group. Thus, by issuing $\widetilde{Q}^c$ on the set of matching records, all aggregations are applied and all clauses of the query are verified in order to yield a single representative record ε_i. Depending on the clauses of $\widetilde{Q}^c$, ε_i can also be an empty set.

Finally, if not an empty set, ε_i is added to the *map* data structure as a value for the key i, and i is added back to the priority queue associated with the value of ε_i. The loop ends when the priority queue is empty, i.e., when all the entities satisfying the query have been emitted.

We now describe the matchingProcedure (Procedure 1) in detail. Given a record (*recordID*), it seeks for its matches iterating among its candidates. Lines 4-5 are needed to manage the first calls of Algorithm 1 mentioned above, which compare only the seed records—the following calls do not need this check. Firstly, for each candidate, matchingProcedure checks if it is already been assigned to the current entity cluster E (Line 6). This may happen when "following" the match: we do not want to execute a comparison with a record already assigned to the entity cluster of *recordID*. For example, in Figure 6 we want to avoid comparing r_2 and r_3 and vice-versa, since they already are in the entity cluster of r_1. Then, matchingProcedure checks whether the candidate pairs involving r_i have been already compared, in Lines 8 and 11, by exploiting *MatchLists* and *NonMatchLists*. If both *MatchLists*[*recordID*][p] and *NonMatchLists*[*recordID*][p] are equal to zero (i.e., false), this means that the comparison has not been executed yet. Hence, matchingProcedure invokes the matching function in Line 13 (denoted with the notation μ^Q). If the candidate record is a match, then it is inserted in $\mathcal{R}$ (Line 14). Finally, the candidate is added to E to avoid checking it again in further calls (Line 15), and the current *recordID* to the *MatchSet* (Line 23).

4.2.2 Special case: discordant ordering queries We present a variation of the *BrewER* algorithm called *Discordant BrewER*, which introduces an optimization for special yet frequent case of queries, namely queries that order the entities with the following predicates: *(i)* ORDER BY MIN(·) DESC and *(ii)* ORDER BY MAX(·) ASC (here we discuss only the former case; it is trivial to adapt for the latter). We call this case *discordant ordering* because the first entity to be emitted is the one with the maximum value, which is in turn the minimum value among the records that compose that entity.

Discordant BrewER is based on the observation that if a seed record matches with a non-seed record with a higher value, the value of the latter is not for certain the value of the final entity. Thus, the values of seed records belonging to an entity ε determine the maximum value that ε can assume: non-seed records can only lower that value, if they match. Hence, the heap in Algorithm 1 can be initialized by considering only seed records, i.e., by omitting the union with the candidates in Line 3. This significantly reduces the searching space and allows to achieve correct results with a

Table 1: Characteristics of the selected datasets.

Dataset	#D	#Matches	#Ent (AVG Size)	#Attr	OA
SIGMOD20	13.58k	12.01k	3.06k (4.4)	4	megapixels
SIGMOD21	1.12k	1.08k	190 (5.9)	4	price
Altosight	12.47k	12.44k	453 (27.534)	4	price
Funding	17.46k	16.70k	3.11k (5.6)	17	amount

fraction of the comparisons needed by the general algorithm, as we show with the experiments of Section 5.3.

5 Evaluation

This section aims to answer the following questions:

Q1. *What is the performance of BrewER when executing ER-on-demand?* (Section 5.1)

Q2. *How do ER-on-demand baselines derived from traditional batch and progressive ER methods perform?* (Section 5.2)

Q3. *What is the improvement in the case of discordant ordering queries introduced in Section 4.2.2?* (Section 5.3)

Q4. *How well does BrewER perform with different aggregate functions?* (Section 5.4)

Q5. *How does BrewER perform with blocking?* (Section 5.5)

Q6. *How fast is BrewER and how much time does it save without cleaning the entire dataset?* (Section 5.6)

Datasets We employ four real-world datasets from multiple domains with different sizes and characteristics, summarized in Table 1. For all of them the ground truth is known. The first dataset, SIGMOD20 [11, 44], is composed of camera specifications extracted from 24 e-commerce websites and has been employed for SIGMOD 2020 Programming Contest [12]. The second dataset is SIGMOD21, provided by Altosight [1] for SIGMOD 2021 Programming Contest [13], which contains well-curated specifications of electronic products (mainly USB pen drives) scraped from more than 20 websites. The third dataset, Altosight, is a superset of SIGMOD21, but differently from it, this larger set of entities is not well-curated and presents many noisy records with redundant values, missing values, and/or HTML tags. The last dataset is Funding [34], which reports financing requests addressed to the NYC Council Discretionary Funding. ER can be performed on it to identify the organizations presenting these requests as in [14].

We preprocess all the datasets by casting the ordering values to floats, lowercasing all the attributes, and filtering out records with a null value in the ordering attribute. The null values do not affect the final ordering of the entities (i.e., they are not considered by the aggregate functions), but slow down the computation for those entities that have a lot of them. An auxiliary experiment in Appendix C shows that the performance of *BrewER* is slightly affected when it involves a high number of entities with null values.

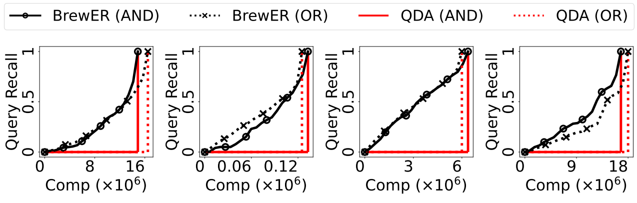
Experimental setup *BrewER* has been implemented in Python 3.7. Our experiments were performed on a server machine equipped with an Intel Xeon Silver 4116 CPU @ 2.10 GHz, a Nvidia Tesla T4 GPU, and 100 GB of RAM.

5.1 Performance of the *BrewER* algorithm

Here, we want to assess how *BrewER* performs in terms of comparisons required to progressively return the result sets of queries. As a baseline we employ *QDA* [2], which applies conflict resolution function directly to each comparison; thus, it can work only with MIN and MAX aggregate functions. For this reason, we consider only these two aggregate functions in this section. In the following Section 5.4, we will show the performance of *BrewER* with other aggregate functions.

Table 2: Minimum, maximum and average cardinality of the result sets of the considered batches of queries.

	Conjunctive Queries			Disjunctive Queries		
Dataset	#MIN	#MAX	#AVG	#MIN	#MAX	#AVG
SIGMOD20	27	172	55.63	368	567	440.55
SIGMOD21	5	15	7.43	28	85	55.45
Altosight	9	32	18.40	87	193	139.08
Funding	8	212	42.13	336	2297	1259.05



(a) SIGMOD20 (b) SIGMOD21 (c) Altosight (d) Funding

Figure 7: Progressive Recall with *BrewER*.

Since the goal of this experiment is to evaluate the *BrewER* algorithm (i.e., Algorithm 1), we do not employ any blocking strategy, which would influence the overall performance. We study how *BrewER* performs with blocking in Section 5.5. Finally, we are not interested in neither designing nor discovering the best matching functions for the task; hence, as a matcher, we employ an oracle that correctly labels all the comparisons—remember that the ground truth is known and *BrewER* is matcher-agnostic.

5.1.1 *Query generation*

We now describe how we generated the synthetic queries for our experiments.

We consider two basic types of queries. Firstly, *conjunctive queries*: queries with two selection predicates employing the LIKE operator in AND to express queries on related attributes (e.g., to select the prices of a series of specific models produced by a brand). Secondly, *disjunctive queries*: queries with two selection predicates, in OR, referring to the same attribute (e.g., to select all the models produced by two brands). The column OA of Table 1 indicates the ordering attribute employed for each dataset. Further technical details are provided in Appendix A.

For each dataset, we consider two batches of 20 queries: one for the conjunctive and one for the disjunctive case. Since the goal is to study the progressive emission of the resulting entities, each set is composed of the 20 queries emitting the highest number of entities out of a set of at least 50 randomly generated queries. Their characteristics are described in Table 2.

5.1.2 *Measures*

For each batch of conjunctive/disjunctive queries executed on a certain dataset, we compute the *progressive average macro-recall* (*progressive recall* for simplicity), as follows. The recall for a query Q_i is defined as $recall_{Q_i} = \frac{\#\{emitted\ entities\}}{\#Q_i^c(\mathcal{D})}$, where $\#Q_i^c(\mathcal{D})$ is the cardinality of the result set for the query Q_i. For each query Q_i in a batch of 20, we track the recall by steps of 5% of the total number of comparisons entailed by Q_i (i.e., a total of 20 steps). Thus, 20 values of $recall_{Q_i}$ are collected for each Q_i in the batch. For instance, if Q_1 entails one million of comparisons, we record the recall of its execution in *BrewER* every 50,000 comparisons (5% of one million). Then, to obtain aggregate values for each batch of queries: *(i)* we compute the average number of comparisons for each step among the queries

$$avg.\ num.\ comp. = \frac{\sum_{Q_i \in [Q_1,..,Q_N]} \#executed\ comparisons\ for\ Q_i}{N}$$

and *(ii)* the average value of recall corresponding at that step, i.e., the *macro-recall* for a batch of queries (or simply *Query Recall*):

$$Query\ Recall = \frac{\sum_{Q_i \in [Q_1,..,Q_N]} recall_{Q_i}}{N}$$

In our experiments, $N = 20$; thus, for each batch of queries, the *progressive recall* is represented by 20 points (one for each step) that can be reported in a single plot to summarize the performance of an ER algorithm on that batch.

5.1.3 Baseline We adapted *QDA* [2] to process queries that contain predicates on categorical attributes. *QDA* does not provide any mechanism to handle ORDER BY clauses, thus the result of its execution is a *batch* version of each query, i.e., performing the sorting of the entities at the end of the resolution process. Also, it supports only MIN and MAX as aggregate functions, since it merges (i.e., resolves) pairs of records as soon as they are found to be matching—this is not compatible with aggregate functions like VOTE and AVG, which take as input more than two values. In a nutshell, *QDA* tries to discard the entities that are not part of the result as soon as possible, by incrementally matching pairs of records that belong to those entities. In practice, by using our terminology, *QDA* tries to match all the seed records first—as in our Algorithm 1 we do by checking the match with the seed records. Hence, *BrewER* and *QDA* perform the same number of comparisons if enough time is given.

5.1.4 Results Figure 7 shows the average progressive recall obtained through the execution of the described randomly generated batches of queries, for each dataset and type of query.

QDA shows a typical step curve for this task due to the fact that it has to compare all the candidate pairs before starting emitting the results. On the other hand, *BrewER* exhibits a progressively increasing recall for all the four datasets as a function of the executed comparisons. We do not observe particular differences in performance among the datasets. Also, the kind of query (AND/OR) does not affect the performance. On SIGMOD20 (Figure 7a) and Funding (Figure 7d), disjunctive queries entail a higher number of comparisons than the conjunctive queries, on average (at most 15% more); vice versa, for SIGMOD21 (Figure 7b) and Altosight (Figure 7c), conjunctive queries need more comparisons (at most 10% more).

This is due to the generation of the seed records: as explained in Section 4.1.1, the seed records of a query are extracted with a disjunctive query and employed in Algorithm 1. Thus, the final number of comparisons depends on the selectivity of each predicate, and not on the result size.

5.2 Batch and Progressive ER shortcomings

We compare *BrewER* against two baselines that we derived from *(i)* a traditional batch ER workflow and *(ii)* an existing progressive ER method. We call the former *Batch-query-baseline* (*BBaseline*) and the latter *Progressive-query-baseline* (*PBaseline*). Our goal is to show that adapting existing ER methods to produce *correct* results for a query without cleaning the entire data or without designing a specific progressive algorithm is not trivial.

Table 3: *BrewER* vs. *Batch-query-baseline.*

	BrewER		***Batch-query-baseline***					
Dataset	R, P, F_1	Err@x	R	P	F_1	Err@1	Err@5	Err@20
SIGMOD20	1.00	0%	0.89	0.99	0.92	30%	13%	9%
SIGMOD21	1.00	0%	0.91	0.50	0.60	30%	40%	42%
Altosight	1.00	0%	0.89	0.20	0.31	60%	45%	57%
Funding	1.00	0%	0.71	0.86	0.77	100%	50%	70%

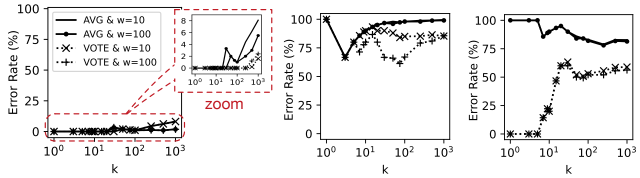
5.2.1 Batch-query-baseline Algorithm 1 guarantees that the results of a query issued on top of a dirty dataset are emitted as if the query were issued on the cleaned version of the dataset—i.e., $Q^c(\mathcal{D}) \equiv Q(\mathcal{D}^c)$. Yet, how good would the results be if we simply issue the query directly on the dirty dataset (i.e., $Q(\mathcal{D})$) and then perform ER on just that portion of the data? This question arises from observing that $Q(\mathcal{D}) \not\equiv Q(\mathcal{D}^c)$. In fact, by issuing the query Q (e.g., Q_1 in Figure 4a) directly on the dirty data, relevant records might be filtered out (e.g., records r_2 and r_4 in the example of Figure 5a). To investigate this effect, *BBaseline* first filters the dirty data $\mathcal{D}$ with Q and then performs ER on the result $Q(\mathcal{D})$. We compare it against *BrewER* by executing a batch of ten randomly generated AND queries (see Section 5.1.1) for each dataset. For each query q, we consider the set of matching pairs $\mathcal{M}_q$ needed to identify the entity set that satisfies q—we know $\mathcal{M}_q$ from the ground truth of each dataset—and the set of matches $\mathcal{M}_\varepsilon$ that the considered method identifies for producing the results. Then, we compute recall R_q, precision P_q and F_1-score F_{1q} as follows:

$$R_q = \frac{|\mathcal{M}_q \cap \mathcal{M}_\varepsilon|}{|\mathcal{M}_q|} \qquad P_q = \frac{|\mathcal{M}_q \cap \mathcal{M}_\varepsilon|}{|\mathcal{M}_\varepsilon|} \qquad F_{1q} = \frac{2 \cdot R_q \cdot P_q}{R_q + P_q}$$

The results of the comparison are shown in Table 3, where we report the average of recall, precision and F_1-score for each batch of queries. *BrewER* always returns the correct answer, and thus recall, precision and F_1-score are always 1.00. We also report the *Error Rate* (*Err@k*) of a method, which is the percentage of erroneously yielded entities in the *first k* emitted entities. For instance, *Err@*20 = 42% means that among the first 20 entities emitted according to the ORDER BY clause, 42% are incorrect according to the ground truth. These errors are introduced by filtering the dirty data with Q. For example, applying Q_1 directly to the dirty dataset of Figure 5a and considering AVG as resolution function for price, ε_1 ends up with a price of \$175 (since r_2 is filtered out), instead of \$155, which is ε_1 price value in the ground truth (Figure 5b).

Table 3 shows how the error rate is significant for all the datasets and for different values of k. On the other hand, *BrewER* (being an exact method) always has an error rate of 0%.

5.2.2 Progressive-query-baseline A common approach employed in state-of-the-art progressive ER methods [32, 38, 42] is based on the *Sorted Neighborhood* (SN). The basic idea is to sort all the records according to an attribute that can capture their similarity (e.g., price of products), then to slide a window from the head to the tail of the sorted list, and to progressively compare all the pairs of records that fall within that window, i.e., the *neighborhood.* The original method proposed in [42] starts with a window of size $w = 2$ and then, after each iteration over the whole list, increases the size of the window (to $w = 3$, $w = 4$, etc.). The main problem with this



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 8: *Progressive-query-baseline* Error Rate.

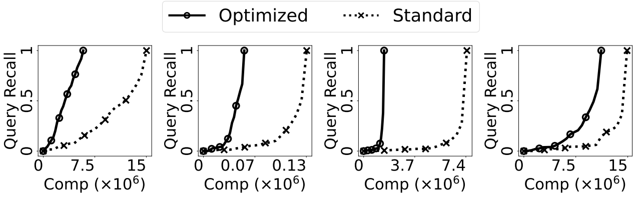
method is that it does not satisfy the *correctness* and *monotonicity* conditions of Definition 1: each time that w increases, new matches can be found for an entity, hence the final aggregate value may change. Yet, we can choose to set a value for w, performing a single iteration over the sorted list of records and avoiding this problem—the disadvantage is that we need to pre-specify w.

To measure the quality of the progressive entity emission in an *ER-on-demand* setting of the SN method, we sort all records by the attribute employed in the ordering clause (megapixels for SIGMOD20, price for Altosight, and amount for Funding). Then, we employ $w = 10$ and $w = 100$ to represent two opposite scenarios [30]: the former setting favors efficiency over recall, while the latter does the opposite. For the queries, we consider only the basic query with the GROUP BY ENTITY and ORDER BY predicates: queries with selection predicates could be simply applied to the progressively generated entities, but they would not affect the entity emission order—which is what we want to assess here. As resolution functions for the ordering attribute, we consider both AVG and VOTE.

As in the previous experiment, we report the error rate (*Err*@*k*) of *PBaseline* measured on the *first k* emitted entities. We mark an entity as erroneous only if the value of the ordering attribute is different from the ground truth; thus, if errors affect other attributes, we do not consider them. We did not notice any significant difference between ascending and descending ordering and report only the former. The results are shown in Figure 8; note that *BrewER* is an exact method, hence its error rate is always 0%. In SIGMOD20, the intra-cluster variance for the ordering attribute (megapixels) is very low, hence the error rate with AVG (VOTE) is under 4% (1%) for the first 100 emitted entities, rising up to 8% (2.5%) for the first 1000 emitted entities. On Altosight, *PBaseline* always fails to emit the first entity correctly (*Err*@1 = 100%); with VOTE on the first 100 (1000) entities, the error rate is at least 60% (75%); with AVG, near to 100% for $k \in (80, 1000)$. On Funding, *PBaseline* is correct only for $k \leq 10$ with VOTE, with high error rates (at least 50%) for all the other settings. All these errors occur because *PBaseline* (as all the progressive ER methods) does not keep track of the value of a resolved entity while resolving it (e.g., as *BrewER* does through the priority queue). Hence, *PBaseline* is not reliable for *ER-on-demand*.

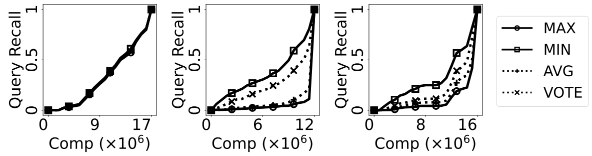
5.3 Discordant ordering queries

Algorithm 1 presents an optimized version to manage the special—yet frequent—case of discordant ordering (Section 4.2.2). To evaluate its performance, we employ the same settings of Section 5.1, with one major difference: the randomly generated queries have MAX-ASC or MIN-DESC as combinations of the aggregate function for the ordering value (i.e., MAX/MIN) and ordering mode (i.e., ASC/DESC).



(a) SIGMOD20 (b) SIGMOD21 (c) Altosight (d) Funding

Figure 9: Progressive Recall with discordant ordering queries.



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 10: Progressive Recall varying aggregate functions.

Figure 9 compares the progressive recall of both the standard Algorithm 1 and the optimized version for discordant ordering. On all the datasets, the optimized version terminates the queries by saving a significant amount of comparisons, up to four times compared to the standard Algorithm 1 on Altosight (Figure 9c).

We observe that for SIGMOD21 (Figure 9b), Altosight (Figure 9c) and Funding (Figure 9d), the recall curve is much more flat at the beginning of the plot and much steeper at the end, compared to the non-discordant-ordering case of the previous experiment (Figure 7). This is because with MAX-ASC and MIN-DESC queries (i.e., discordant ordering queries), when considering the head element of the priority queue, if a match is found it determines the re-insertion of the element in a lower position in the queue. This entails a higher average delay in the emission of the entities for SIGMOD21, Altosight and Funding (Figures 9b-d). Differently, SIGMOD20 is only marginally affected by that phenomenon. This can be explained by the fact that in SIGMOD20 the variance of the values for the megapixels attribute is very low within each cluster of entity records (i.e., most of the entities have records with similar ordering values). On the other hand, the price values within a single entity in Altosight may have a high variance (e.g., due to special offers).

Finally, no significant differences can be found between conjunctive (in Figure 9) and disjunctive queries, as when employing the standard Algorithm 1.

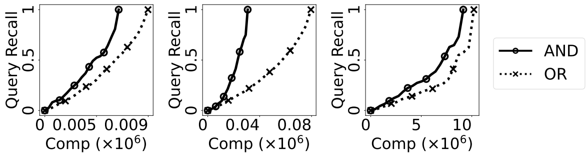
5.4 Experiments with aggregate functions

In Section 5.1 and Section 5.3 only MIN and MAX have been considered. The goal of this experiment is to evaluate *BrewER* with a set of different aggregate functions. We set ASC as ordering mode and we run each query of the batch with the following aggregate functions: MAX, MIN, AVG, VOTE. The batch of 20 AND queries is generated as explained in Section 5.1.1. In this experiment, MAX represents the discordant case and the optimized version is not employed.

The plots are in line with the previously observed results in Figures 7 and 9. SIGMOD20 does not present relevant variations: the performance by changing aggregate functions is almost unaltered (Figure 10a). Again, this can be explained by the little variance that the ordering attribute assumes among records belonging to the same entity. On the other hand, when the variance is high a

Table 4: Blocking characteristics.

Dataset	Recall	Precision	F_1
SIGMOD20	0.933	0.407	0.567
Altosight	0.999	0.056	0.107
Funding	0.966	0.014	0.028



(a) **SIGMOD20** (b) **Altosight** (c) **Funding**

Figure 11: Progressive Recall with blocking.

significant difference in the behavior of the aggregate function is observed, as shown for the other datasets in Figures 10b-c.

5.5 Performance with blocking

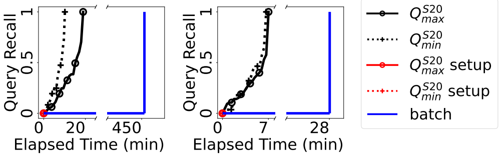
With this experiment, we want to evaluate if and how the performance of *BrewER* changes by employing blocking. Due to its small size, SIGMOD21 is not considered for this experiment. We employ JedAI [28, 31], which is based on a completely unsupervised blocking approach. We use its standard configuration based on *Token Blocking* and *Meta-blocking* [28]. Table 4 reports recall, precision and F_1-score of the produced candidate pairs. The goal of blocking is to reduce superfluous comparisons, while preserving as many true positives as possible; hence, it is typical to have high recall and low precision in this phase [8]. The final recall and precision of the ER process is determined ultimately by the quality of the matching function adopted, which is not evaluated here. We also evaluated manually-devised blocking strategies that yield lower recall (results available in Appendix B), for which we did not notice any significant difference in behavior when using *BrewER*.

The queries have been synthesized as explained in Section 5.1.1 and the results are presented in Figure 11. By employing blocking we see a huge reduction of required comparisons for all datasets compared to the all-pairs solutions of Figure 7 (up to 200 times for Altosight). As for the progressive recall, with SIGMOD20 and Altosight the curve for the AND queries is much steeper than the one for the OR queries. This happens because conjunctive queries allow to filter out blocks appearing in connected components whose records do not satisfy every predicate of the query, as explained in Section 4.1.1. Differently, with Funding, the difference between conjunctive and disjunctive queries is less evident. This is due to to the high intra-block variance of the selection attribute values, which limits the efficacy of the preliminary block filtering.

5.6 Runtime evaluation

We now want to assess the runtime performance of *BrewER* in a real-world scenario, by employing a state-of-the-art matching function and measuring the progressive recall as a function of the actual time needed for answering a query.

We design the experiment as follows. We consider a large dataset with blocking and a small one without blocking: the former is SIGMOD20, the latter is SIGMOD21. Then, from the batch of disjunctive queries of Section 5.1, we select for each dataset two queries: one yielding the largest result set, the other yielding the smallest result set (see Table 2 for the size of the query results). Thus, a total of four queries are considered: Q^{S20}_{max} and Q^{S20}_{min} for SIGMOD20, and Q^{S21}_{max} and Q^{S21}_{min} for SIGMOD21. As a matching function we employ a pre-trained deep learning classifier built with DeepMatcher [25]—by exploiting their Hybrid model, which we found to achieve good performances on our datasets.



(a) **SIGMOD20** (BL) (b) **SIGMOD21**

Figure 12: Query execution runtime in *BrewER*.

The results are shown in Figure 12, which reports for each query the average runtime of ten executions. The plots also report the runtime required for cleaning the entire dataset with a traditional batch ER method (blue line). For all datasets, the correct results start to be emitted after the first few minutes of execution. For instance, on SIGMOD20 (Figure 12a), with *BrewER* users receive 22 and 31 entities after only two minutes for Q^{S20}_{max} and Q^{S20}_{min}, respectively. Instead, with the complete ER process with a batch method users would wait 8 hours—even if the dataset has "only" thirteen thousand records and blocking is employed. A similar behavior is observable also with SIGMOD21 (Figure 12b).

Finally, the overhead time required by *BrewER* (i.e., for generating and executing the seed query and to initialize the priority queue) is negligible compared to the overall execution time of the query. In particular: *(i)* the startup time for *BrewER* is circa 4 and circa 0.1 seconds for SIGMOD20 and SIGMOD21, respectively; *(ii)* the average overhead introduced by *BrewER* for each comparison is 0.01 milliseconds, while the average runtime for comparison of the matching function is 2.7 milliseconds.

6 Conclusion

We introduced an *Entity Resolution (ER) on-demand* algorithm that tries to minimize the cleaning effort needed to evaluate an SQL SP query issued on dirty data, while progressively trying to maximize the results returned to the user, in a *pay-as-you-go* fashion. Our algorithm guarantees the correctness of the results: they are the same that would be obtained by issuing the query on the cleaned data (i.e., when ER is performed on the entire data before running the query). Also, our algorithm supports a wide class of *resolution functions*, which users can express directly in the SQL query to resolve the conflicts of attribute values of records belonging to the same real-world entity. We implemented our algorithm in a framework called *BrewER*, which is both *matcher*- and *blocking*-agnostic, meaning that it can operate with any binary matching function (e.g., state-of-the-art deep/transfer learning matchers) and can scale to large datasets with an existing blocking strategy of choice. Our experimental evaluation showed its efficacy on four real-world datasets with benchmark queries that we designed. Finally, we also showed that the overhead of *BrewER* is negligible when employed in real-world use cases. Many challenging problems remain to be explored, including how to support SQL SPJ queries for multi-table dirty datasets and additional features for ER pipeline debugging.

References

[1] Altosight. 2021. *Altosight Official Website.* https://altosight.com/

[2] Hotham Altwaijry, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2017. QDA: A Query-Driven Approach to Entity Resolution. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 402–417.

[3] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 120–131.

[4] Nils Barlaug and Jon Atle Gulla. 2021. Neural Networks for Entity Matching: A Survey. *ACM Trans. Knowl. Discov. Data* 15, 3 (2021), 52:1–52:37.

[5] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. 2009. Swoosh: a generic approach to entity resolution. *VLDB J.* 18, 1 (2009), 255–276.

[6] Jens Bleiholder and Felix Naumann. 2008. Data fusion. *ACM Comput. Surv.* 41, 1 (2008), 1:1–1:41.

[7] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.

[8] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Trans. Knowl. Data Eng.* 24, 9 (2012), 1537–1555.

[9] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2021. An Overview of End-to-End Entity Resolution for Big Data. *ACM Comput. Surv.* 53, 6 (2021), 127:1–127:42.

[10] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *Proc. VLDB Endow.* 2, 2 (2009), 1481–1492.

[11] Valter Crescenzi, Andrea De Angelis, Donatella Firmani, Maurizio Mazzei, Paolo Merialdo, Federico Piai, and Divesh Srivastava. 2021. Alaska: A Flexible Benchmark for Data Integration Tasks. *CoRR* abs/2101.11259 (2021).

[12] Database Research Group of the Roma Tre University. 2020. *SIGMOD 2020 Programming Contest Website.* http://www.inf.uniroma3.it/db/sigmod2020contest/index.html

[13] DBGroup of the University of Modena and Reggio Emilia and Database Research Group of the Roma Tre University. 2021. *SIGMOD 2021 Programming Contest Website.* https://dbgroup.ing.unimo.it/sigmod21contest/

[14] Dong Deng, Wenbo Tao, Ziawasch Abedjan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Guoliang Li, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Unsupervised String Transformation Learning for Entity Consolidation. In *ICDE.* IEEE, 196–207.

[15] AnHai Doan, Pradap Konda, Paul Suganthan G. C., Yash Govind, Derek Paulsen, Kaushik Chandrasekhar, Philip Martinkus, and Matthew Christie. 2020. Magellan: toward building ecosystems of entity matching solutions. *Commun. ACM* 63, 8 (2020), 83–91.

[16] Ivan P. Fellegi and Alan B. Sunter. 1969. A theory for record linkage. *J. Am. Stat. Assoc.* 64, 328 (1969), 1183–1210.

[17] Donatella Firmani, Barna Saha, and Divesh Srivastava. 2016. Online Entity Resolution Using an Oracle. *Proc. VLDB Endow.* 9, 5 (2016), 384–395.

[18] Luca Gagliardelli, Giovanni Simonini, and Sonia Bergamaschi. 2020. RulER: Scaling Up Record-level Matching Rules. In *EDBT.* OpenProceedings.org, 611–614.

[19] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2021. Efficient and effective ER with progressive blocking. *VLDB J.* 30, 4 (2021), 537–557.

[20] Guoliang Li. 2017. Human-in-the-loop Data Integration. *Proc. VLDB Endow.* 10, 12 (2017), 2006–2017.

[21] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE.* IEEE, 13–24.

[22] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proc. VLDB Endow.* 14, 1 (2020), 50–60.

[23] Michael Loster, Ioannis K. Koumarelas, and Felix Naumann. 2021. Knowledge Transfer for Entity Resolution with Siamese Neural Networks. *ACM J. Data Inf. Qual.* 13, 1 (2021), 2:1–2:25.

[24] Jayant Madhavan, Shirley Cohen, Xin Luna Dong, Alon Y. Halevy, Shawn R. Jeffery, David Ko, and Cong Yu. 2007. Web-Scale Data Integration: You can afford to Pay as You Go. In *CIDR.* www.cidrdb.org, 342–350.

[25] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD Conference.* ACM, 19–34.

[26] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989.

[27] Kevin O'Hare, Anna Jurek-Loughrey, and Cassio de Campos. 2019. A review of unsupervised and semi-supervised blocking methods for record linkage. *Linking and Mining Heterogeneous and Multi-view Data* (2019), 79–105.

[28] George Papadakis, Georgios M. Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas, and Manolis Koubarakis. 2020. Three-dimensional Entity Resolution with JedAI. *Inf. Syst.* 93 (2020), 101565.

[29] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and Filtering Techniques for Entity Resolution: A Survey. *ACM Comput. Surv.* 53, 2 (2020), 31:1–31:42.

[30] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *Proc. VLDB Endow.* 9, 9 (2016), 684–695.

[31] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2019. Domain- and Structure-Agnostic End-to-End Entity Resolution with JedAI. *SIGMOD Rec.* 48, 4 (2019), 30–36.

[32] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. 2015. Progressive Duplicate Detection. *IEEE Trans. Knowl. Data Eng.* 27, 5 (2015), 1316–1329.

[33] Alberto Pietrangelo, Giovanni Simonini, Sonia Bergamaschi, Felix Naumann, and Ioannis K. Koumarelas. 2018. Towards Progressive Search-driven Entity Resolution. In *SEBD (CEUR Workshop Proceedings)*, Vol. 2161. CEUR-WS.org.

[34] Qatar Computing Research Institute. 2018. *Data Civilizer System GitHub Repository - Address Dataset.* https://raw.githubusercontent.com/qcri/data_civilizer_system/master/grecord_service/gr/data/address/address.csv

[35] Anish Das Sarma, Xin Dong, and Alon Y. Halevy. 2008. Bootstrapping pay-as-you-go data integration systems. In *SIGMOD Conference.* ACM, 861–874.

[36] Enrico Sartori, Yannis Velegrakis, and Francesco Guerra. 2016. Entity-Based Keyword Search in Web Documents. *Trans. Comput. Collect. Intell.* 21 (2016), 21–49.

[37] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2018. Schema-Agnostic Progressive Entity Resolution. In *ICDE.* IEEE Computer Society, 53–64.

[38] Giovanni Simonini, George Papadakis, Themis Palpanas, and Sonia Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *IEEE Trans. Knowl. Data Eng.* 31, 6 (2019), 1208–1221.

[39] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. 2013. Data Curation at Scale: The Data Tamer System. In *CIDR.* www.cidrdb.org.

[40] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *Proc. VLDB Endow.* 14, 11 (2021), 2459–2472.

[41] Steven Euijong Whang and Hector Garcia-Molina. 2012. Joint Entity Resolution. In *ICDE.* IEEE Computer Society, 294–305.

[42] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. 2013. Pay-As-You-Go Entity Resolution. *IEEE Trans. Knowl. Data Eng.* 25, 5 (2013), 1111–1124.

[43] Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, and Saravanan Thirumuruganathan. 2020. ZeroER: Entity Resolution using Zero Labeled Examples. In *SIGMOD Conference.* ACM, 1149–1164.

[44] Luca Zecchini, Giovanni Simonini, and Sonia Bergamaschi. 2020. Entity Resolution on Camera Records Without Machine Learning. In *DI2KG@VLDB (CEUR Workshop Proceedings)*, Vol. 2726. CEUR-WS.org.

[45] Liang Zhu, Xu Du, Qin Ma, Weiyi Meng, and Haibo Liu. 2018. Keyword Search with Real-time Entity Resolution in Relational Databases. In *ICMLC.* ACM, 134–139.

Table 5: Manual blocking characteristics.

Dataset	Recall	Precision	F_1
SIGMOD20	0.993	0.014	0.028
Altosight	0.999	0.086	0.158
Funding	0.586	0.023	0.044

Appendices

A Query generation

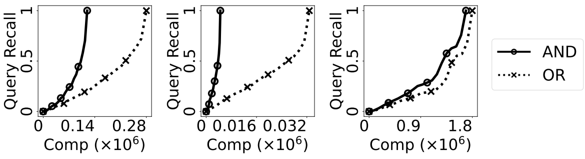
The definition of the batches of queries for the experimental evaluation was carried out according to the following strategies.

For the AND queries on SIGMOD20, the selection is based on a random value among the ten most frequent brands and on a random model series associated with the selected brand; for the OR queries, on two random values chosen from the ten most frequent brands. For the AND queries on SIGMOD21 and Altosight, the selection is based on the list of all nine brands in the dataset and a list of the most common USB stick sizes; for the OR queries, on two random brands. Finally, for the AND queries on Funding, the selection is based on a list of the most common areas in which the organizations are active and on a frequent substring of their names (e.g., *association*, *inc*, etc.); for the OR queries, on two random areas.

B Manually-devised blocking

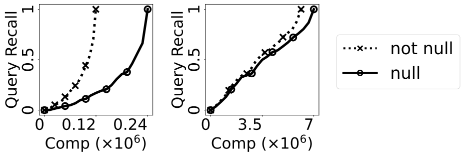
In addition to unsupervised blocking, we also evaluated manually-devised blocking strategies (for simplicity, *manual blocking*). We report in Table 5 the characteristics of the best configuration we could find for each dataset. Yet, for Funding, the blocking strategy yields a low recall. However, it represents a plausible real-world scenario, which gives the opportunity to study whether *BrewER* is affected by an aggressive blocking strategy. As depicted in Figure 13, no significant differences in performance can be detected, apart from the number of comparisons depending on the precision of the blocking function. Details about the adopted blocking strategies are illustrated below.

For SIGMOD20, for each record two keys are extracted from brand and model attribute values as follows. The first key is extracted by removing punctuation marks and numeric characters from the two attributes and concatenating the remaining strings. The second key is generated with the numerical characters of model. Then, these two keys are employed to build an inverted index of the records: each key corresponds to a block and two records are indexed together in the same block if they share at least one key. For Altosight, for each record a key is generated by concatenating brand and size values (if both attribute values are different from null) and removing white spaces. The key is then used for creating the blocks as for SIGMOD20. Of course, the same criterion could be applied to its subset SIGMOD21. For Funding dataset, two keys are considered for each record. Firstly, if name attribute value is not null, up to two tokens before the first comma are extracted (this is done to remove organization suffixes); then, punctuation marks, numeric characters and white spaces are removed to yield the final key. A second key is extracted from address value (if not null) by removing punctuation marks, numeric characters and white spaces. As for the other datasets, these keys are employed for creating the blocks.



(a) SIGMOD20 **(b) Altosight** **(c) Funding**

Figure 13: Progressive Recall with manual blocking.



(a) SIGMOD20 (BL) **(b) Altosight**

Figure 14: Progressive Recall with missing values.

C Performance with missing values

With this experiment, we want to evaluate the impact of null values in the ordering attribute. We consider only SIGMOD20 (with blocking) and Altosight, since SIGMOD21 and Funding do not present null values in their ordering attribute. Thus, 16.2 thousand and 1.5 thousand of additional records with null values have been added to SIGMOD20 and Altosight, respectively, for this experiment. In SIGMOD20 the new records are scattered among around 6 thousand of entities, while in Altosight only on 30 entities.

The experiment results are reported in Figure 14. For Altosight (Figure 14b), the progressive recall of the queries executed on the dataset with null values has a curve similar in shape to the one generated without considering the null values. As a matter of fact, only 30 entities have records with null values, but none of them has a final resolved ordering value equal to null: their priorities in the priority queue do not change. The only difference is the slightly higher number of comparisons needed for the additional records with null values.

Differently, Figure 14a showcases the negative effect of the high number of entities with null ordering value of SIGMOD20. Besides the higher number of comparisons needed for resolving the entire dataset (which has 16.2 thousand records more than the dataset without null values), the progressive recall curve is more flattened in presence of null values for the first circa 90% of the comparisons, and steeper at the end. This is due to the fact that many entities have an ordering value equal to null (i.e., all the ordering values of their records are null): they have the lowest priority in the priority queue, so they are compared, resolved and emitted only at the end of the processing.