Towards Progressive Search-driven Entity Resolution

Alberto Pietrangelo[1], Giovanni Simonini[1*], Sonia Bergamaschi[1], Ioannis Koumarelas[2], and Felix Naumann[2]

[1] Università degli Studi di Modena e Reggio Emilia
181067@studenti.unimore.it; giovanni.simonini@unimore.it; sonia.bergamaschi@unimore.it

[2] Hasso Plattner Institute, University of Potsdam, Germany
ioannis.koumarelas@hpi.de; felix.naumann@hpi.de

Abstract. Keyword-search systems for databases aim to answer a user query composed of a few terms with a ranked list of records. They are powerful and easy-to-use data exploration tools for a wide range of contexts. For instance, given a product database gathered scraping e-commerce websites, these systems enable even non-technical users to explore the item set (e.g., to check whether it contains certain products or not, or to discover the price of an item). However, if the database contains dirty records (i.e., incomplete and duplicated records), a pre-processing step to clean the data is required. One fundamental data cleaning step is *Entity Resolution*, i.e., the task of identifying and fusing together all the records that refer to the same real-word entity. This task is typically executed on the whole data, independently of: *(i)* the portion of the entities that a user may indicate through keywords, and *(ii)* the order priority that a user might express through an *order by* clause.
This paper describes a first step to solve the problem of *progressive search-driven Entity Resolution*: resolving all the entities described by a user through a handful of keywords, progressively (according to an *order by* clause). We discuss the features of our method, named *SearchER* and showcase some examples of keyword queries on two real-world datasets obtained with a demonstrative prototype that we have built.

Keywords: Keyword search · Entity Resolution · Data Cleaning · Pay-as-you-go · Query-driven Data Integration

1 Introduction

Entity Resolution (ER) is a fundamental task for *data cleaning* and *integration* [8][7][11][4]: it aims to identify different representations of the same real-world entity in a given dataset. Typically, an ER workflow is composed of three

* Corresponding author

main sub-tasks: *blocking*, *matching*, and *resolution*. The listed steps may depend on other tasks themselves; e.g.: *blocking* may require *schema-alignment* for the *blocking rules definition*; the *match function* may require the generation of a labeled training set to properly train a *classifier*, etc. We adopt this simplification for the sake of the presentation. Blocking is typically employed to avoid the quadratic complexity of the naïve solution (which compares all possible pairs of records) [6]. Basically, blocking provides the set of *candidate pairs* that are actually compared through a *match function*, i.e., a binary function that takes as input two records r_1 and r_2 and answers the question "*do* r_1 *and* r_2 *refer to the same real-world entity?*"—we say that r_1 and r_2 are *matching* ($r_1 \equiv r_2$) if the answer is *yes*, *non-matching* ($r_1 \not\equiv r_2$) otherwise. Finally, a *resolve function* takes as input all the records referring to a single entity (i.e., a set of matching records) and returns a single *representative* record, resolving conflicts of matches (e.g., it may result that $r_1 \equiv r_2$, $r_2 \equiv r_3$, but $r_1 \not\equiv r_3$), and inconsistent attribute values (e.g., $r_1 \equiv r_2$, but some attributes have different values). This task is also known as *Data Fusion*.

The Challenge

When the computational resources and/or the time are critical components for ER [9], *progressive* ER aims to yield record pairs progressively, trying to maximize the *recall* in case of early termination [18][14][16]. Yet, these techniques are specifically designed to yield *any* match as soon as possible, without considering any indication of the user (i.e., the query); adapting them to the progressive search-driven problem, in a keyword search system [5], is not trivial. This can be particularly useful for data exploration [17]. Consider the following example.

Example: *Given a dirty dataset of e-commerce products gathered from several sources on the Web, say that a user wants to find all the entities (i.e., resolved records) referring to* `Apple iPhone 8` *smartphones, ordered by decreasing price. (This is represented in the keyword-based query in Query 1.) Furthermore, say that the user has a limited time budget for such a task and wants to retrieve as many entities as possible within that budget.*

Keywords:	`Apple iPhone 8`
Ordering:	`Price`
Sort:	`DESCENDING`

Query. 1. Keyword query issued to retrieve all the "Apple iPhone 8" in the database, ordered by decreasing price.

Employing a traditional *batch* approach to ER, i.e., resolving the whole dataset and then executing the query, might be quite expensive on real datasets composed of millions of entities. Even existing progressive ER approaches are useless with such a problem, since they aim to approximate the optimal order of the record comparisons on the basis of the matching likelihood of the record pairs—the matching likelihood is assessed through heuristics, such as the similarity of some attributes. Thus, to generate the final result for Query 1, they

have to perform the whole ER process; this is because the most expensive `Apple iPhone 8` entity might correspond to records that have the lowest matching likelihood in the dataset.

Our approach

In this paper, we investigate the problem of the *progressive search-driven Entity Resolution*. We propose a first attempt to address this problem by envisioning an ER method, called *SearchER*, which enables users to express the *relevance* of the entities of their interest by means of a keyword query combined with the selection of an attribute that determines the ordering (descending or ascending).

From the user point of view, *SearchER* takes as input: *(i)* a dirty dataset, *(ii)* a blocking function, *(iii)* a user keyword query (which defines the entities of interest), *(iv)* an attribute for the ordering predicate, and *(v)* the ordering type (i.e., *ascending* or *descending*). Then, *SearchER* returns as output the solution for the user query, progressively. Under the hood *SearchER* exploits the blocking function to define the space of possible comparisons; then it identifies some initial candidate records to be resolved (i.e., those containing the keywords) and iteratively explores comparisons that involve these records. For ordering entities, *SearchER* considers not only the ordering attribute, but also the number of keywords that are associated to the entities. For example, considering Query 1, it may happen that an entity with a low price containing all the keywords is emitted before an entity with a high price, but that does not contain all the keywords. In other words, we assume that the keywords are more important than the ordering for the user. For this reason, we say that the final solution is *approximate*, since the final ordering might not be completely respected.

We also built a first experimental prototype of *SearchER* and tested it with some queries on two real-world datasets. This very preliminary result does not intend to show the efficacy of our method in general, yet it allows us to showcase that such an approach to ER is actually feasible and promising.

The reminder of this paper is organized as follows: Section 2 introduces the preliminaries; Section 3 describes the envisioned *SearchER* method; Section 4 reports our preliminary experiments on two real-world datasets; Section 5 describes main related work; finally, Section 6 concludes the paper and discusses the ongoing and future work.

2 Preliminaries

To perform ER, the naïve comparison of all possible pairs of records has a quadratic complexity, thus for scaling to large datasets *blocking* techniques are generally employed [6]: the records are indexed according to heuristics (called *blocking criteria*) into clusters (possibly overlapping) and the all-pair comparison is executed only within each cluster (a.k.a. *block*). Thus, the efficacy of blocking (i.e., how many matches are indexed in the blocks) strictly depends on the definition of the blocking criteria. Intuitively, large and overlapping blocks capture more matches than small and non-overlapping ones, but at the expense of efficiency.

An approach that has been shown to achieve high accuracy is *meta-blocking*, which operates on large and overlapping blocks, restructuring them to filter out non promising comparisons.

Meta-blocking relies on the assumption that the matching likelihood of any two records is analogous to their degree of co-occurrence in a block collection. This means that a block collection B has to be generated by a blocking method that yields redundancy-positive blocks, where the similarity of two records is proportional to the number of blocks they share.

Based on redundancy, which is common for blocking methods [12], meta-blocking represents the block collection as a *blocking graph*. This is an undirected weighted graph $\mathcal{G}_B(V_B, E_B)$, where V_B is the set of nodes, and E_B is the set of weighted edges. Every node $n_i \in V_B$ represents a record $r_i \in R$, while every edge $e_{i,j}$ represents a comparison $c_{i,j} \in B \subseteq R \times R$. A *weighting function* is employed to weight the edges, leveraging the co-occurrence patterns of records in B: each edge is assigned a weight that is derived exclusively from the (characteristics of the) blocks its adjacent records have in common. For example, the *ARCS* function sums the inverse cardinality of common blocks, assigning higher scores to pairs of records sharing smaller (i.e., more distinctive) blocks.

For the problem of progressive search-driven ER, we propose a revised and extended version of the blocking graph model, as explained in Section 3.

3 The SearchER Method

At its core, *SearchER* employs a *blocking function* to generate the block collection that is exploited for building blocking graph. In the following we describe the novel *node-weighting* and *edge-weighting* strategies employed by *SearchER* for building the blocking graph, and finally we describe how the blocking graph is employed by *SearchER* for the query evaluation.

Revised node weighting: The nodes are weighted according to the likelihood of appearing in the final solution. This introduces the concept of *Record-Relevance*; the intuition is explained with the following example:

Example: Consider Query 1, issued by a user that is looking for `iPhone` *entities and requires the results to be generated progressively, from the most expensive to the cheapest ones. Say that a record r_1 refers to entity ε_1 and has a high price. Say also that r_1 has many edges connecting it to many records with low prices, and that these edges have a high matching likelihood. So, it is likely that the final (resolved) price of ε_1 will not be high. This means that it is likely that ε_1 will not belong to the final solution. Hence, the Record-Relevance (i.e., the node-weight) of r_1 should be low.*

Record-Relevance computation: The Record-Relevance is assessed by adapting *tf-idf* [10], a widely employed information retrieval measure, in the following way: firstly, a weight is assigned to each node proportionally to the number of

In our preliminary experiment we employ *Token Blocking* [12], which considers each token as a blocking key, regardless of the attribute in which it appears.

keywords that the corresponding record contains (e.g., using tf-idf); secondly, a fraction of the weight (e.g., $fracweight/\#neighbors$) is equally propagated to its neighbours.

Revised edge weighting: The weight of the edges captures the likelihood of *affecting* the final solution. The weighting schema considers both the matching likelihood of an edge (as in meta-blocking [15]), and its *Edge-Relevance*.

Example: Consider Query 1. Say that a record r_1 is connected to only one record r_2 with its same price, and both belong to the real-world entity ε_1 (i.e., $r_1.year = r_2.year$ and $r_1 \equiv r_2$). Then, performing the comparison of r_1 and r_2 will not change the rank of ε_1. Hence, the Edge-Relevance (i.e., the edge-weight) of the edge connecting r_1 and r_2 should be low.

The *Edge-Relevance* also depends on the *resolve* function employed by the user. For the intuition consider the following example:

Example: Consider Query 1. Say that r_1 (with a high price $r_1.year$) is connected to another record r_2 (with $r_2.year \ll r_1.year$). Then, the Record-Relevance of r_1 (and r_2) should change on the basis of the resolve function: if the resolve function assigns `MAX`$(r_1.year, r_2.year)$ as final price of the entity, it is more likely for r_1 (and r_2) to be part of the answer for Query 1; which is not true if the resolve function assigns `MIN`$(r_1.year, r_2.year)$ as final price.

Edge-Relevance computation: The Edge-Relevance between two nodes is computed as the weight in a traditional blocking graph, normalized by the relative variation (e.g., $\Delta price = r_i.year - r_j.year$) of the attribute value that is employed for the ordering clause. The edge weight has to be stored also with a *sign* that indicates the direction of the variation: a positive value means that the value of the adjacent node r_i with the smaller id is greater than the other node r_j with a higher id (i.e., $i < j$)—this is just a convention, it could be the other way around. Thus, the edge weight can be interpreted differently on the basis of the *resolve* function (e.g., MIN/MAX price of the matching records).

Query evaluation: Processing a user's query, the blocking graph is not built in its entirety (i.e., considering all the records/nodes); instead, only the portion of the graph that is valuable for the query is considered. In fact, given a block collection, the *node-centric subgraph* of the blocking graph for any node r_i (i.e., the subgraph involving r_i and its neighbours only) can be efficiently built using algorithms described in [12] and [19]. Thus, *SearchER* considers only the node-centric subgraphs of the nodes that correspond to the records containing at least one of the given keywords, and merges subgraphs that share nodes. Notice that the resulting blocking graph can be disconnected.

When looking at the obtained blocking graph, we observe in preliminary experiments that for queries with a limited number of keywords (ideally the vast majority), many nodes tend to have the same weights. *SearchER* divides them into *levels* and resolves the records starting from the highest levels and sorts the comparisons within each level according to their edge-weights. As soon as a node is evaluated (i.e., all the comparisons involving it have been performed), the resulting entity is emitted.

4 Preliminary experimental results

We devised a prototype of *SearchER* that takes advantage of the proposed data model and performed preliminary experiments by issuing keyword queries on top of two well-known, real-world datasets (for which the ground-truth of the matching records is known). The first dataset is `CDDB` [13], which contains CD entities described with: name, artist, category, genre, and year fields along with the track titles. The second dataset is `CORA` [13], which contains computer science research articles described with: affiliation, author, location, title, venue and year.

In the following we report the results obtained for the following queries:

q1:	"Rock"	issued on `CDDB`
q2:	"Rock and Roll"	issued on `CDDB`
q3:	"Genetic Algorithms Applications"	issued on `CORA`
q4:	"Artificial Intelligence"	issued on `CORA`
q5:	"Jazz Music"	issued on `CDDB`
Ordering:	`Year`	
Sort:	`DESCENDING`	

For each query, the entities are sorted by descending values of the attribute `year` for both the datasets. As *resolve* function we employed $MAX(r_i.year, r_j.year)$, yet very similar results have been obtained employing $MIN(r_i.year, r_j.year)$. In our evaluation we consider as baselines: *Standard Blocking* (SB) and *Sorted Neighbour* (SN), with the same configurations of [6]: the ER process is executed employing these methods, and then we performed the queries on top of the resolved entity set (using tf-idf for ranking the results). The results for $q1$, $q2$, $q3$ and $q4$ are shown in Figure 2: it reports the number of emitted entities that satisfy the corresponding query (y-axis), as function of the number of pairwise comparisons performed (x-axis).

The results show that *SearchER* starts emitting entities much earlier (in terms of compared pairs) than the other methods; this is because traditional blocking techniques have to wait until the last comparison in order to evaluate the user query. Surprisingly, we observe that for some queries ($q3$ and $q4$ in Figure 2c,d), the baselines cannot identify as many matches as *SearchER*. This is due to the underlying blocking techniques that fail to yield some of the candidates that are actually matches, and which are correctly identified as such in the blocking graph.

Similarly, for query $q5$, we report the number of entities found, in function of the number of comparisons (Figure 3a) and in function of the time (Figure 3b). Showing that the advantage of *SearchER* is evident also considering the execution w.r.t. execution time.

5 Related Work

Query Driven Approach—To avoid the ER evaluation on the whole dataset before answering any query, recent works proposed a *Query Driven Approach* (QDA) to ER [2][1], which aims to resolve only the portion of the entities in a dataset that are

We report that the proposed method provides an approximate solution: some of the emissions are not in the corrected order and a tolerance range has been considered.

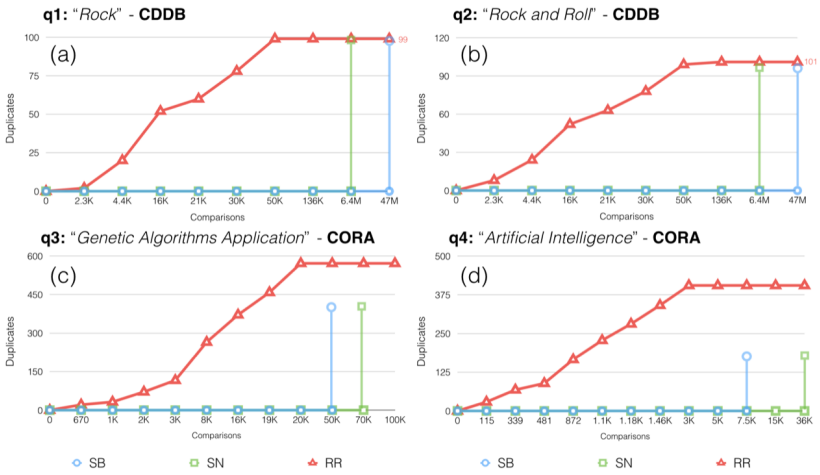

Fig. 2. Results for q1, q2, q3 and q4.

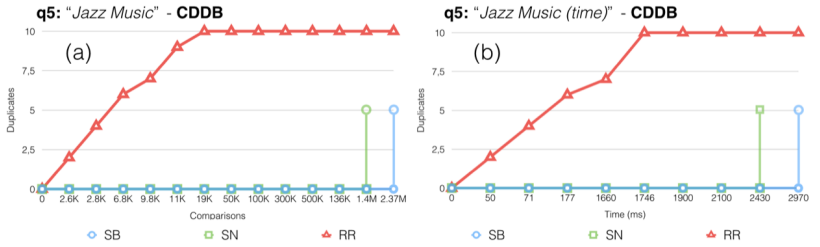

Fig. 3. Results for q5

actually relevant to the user, by exploiting `SPJ SQL` clauses. The `ORDER BY` clause is out of the scope of QDA.

Thus, Query 1, where the relevance ordering is expressed through an `ORDER BY` clause, will force QDA to resolve all the entities in the dataset and then sort them by price. Furthermore, QDA strictly relies on batch blocking, which is employed as initial step of the ER. Hence, adapting QDA to work in a pay-as-you-go fashion is not trivial.

6 Conclusion and Future Work

In this paper, we have presented a preliminary study of the problem of progressive search-driven Entity Resolution, namely the task of deduplicating records of a dirty dataset by following a user query that expresses: *(i)* the keywords representing the entities of interest (e.g., "Apple iPhone 8"); *(ii)* the ordering of interest (e.g., from the most expensive to the cheapest). We have proposed a first solution for solving this problem, and proposed an approximate method, called *SearchER*.

We believe that the proposed method paves the way for further investigation of the problem. In particular, we are currently devising a method to provide an exact

solution for a wide range of *resolution function*; e.g., w.r.t. Query 1: $minimum/-maximum/user-defined-function([price])$ for determining the final price of an entity. We are also investigating how to exploit different blocking techniques (other than Standard Blocking and meta-blocking) and advanced similarity functions for the matching phase [3]. Finally, we will compare approximate solutions (such as *SearchER*) and exact solutions, to thoroughly study their characteristics and defining the trade-offs to guide practitioners.

References

1. Altwaijry, H., Kalashnikov, D.V., Mehrotra, S.: Query-driven approach to entity resolution. PVLDB **6**(14), 1846–1857 (2013)
2. Altwaijry, H., Kalashnikov, D.V., Mehrotra, S.: QDA: A query-driven approach to entity resolution. IEEE TKDE **29**(2), 402–417 (2017)
3. Benedetti, F., Beneventano, D., Bergamaschi, S., Simonini, G.: Computing inter-document similarity with context semantic analysis. Information Systems (2018)
4. Bergamaschi, S., Ferrari, D., Guerra, F., Simonini, G., Velegrakis, Y.: Providing insight into data source topics. J. Data Semantics **5**(4), 211–228 (2016)
5. Bergamaschi, S., Guerra, F., Simonini, G.: Keyword search over relational databases: Issues, approaches and open challenges. In: Bridging Between Information Retrieval and Databases. pp. 54–73 (2013)
6. Christen, P.: A survey of indexing techniques for scalable record linkage and deduplication. IEEE TKDE **24**(9), 1537–1555 (2012)
7. Dong, X.L., Srivastava, D.: Big Data Integration. Morgan & Claypool (2015)
8. Getoor, L., Machanavajjhala, A.: Entity resolution: Theory, practice & open challenges. PVLDB **5**(12), 2018–2019 (2012)
9. Madhavan, J., Cohen, S., Dong, X.L., Halevy, A.Y., Jeffery, S.R., Ko, D., Yu, C.: Web-scale data integration: You can afford to pay as you go. In: CIDR. pp. 342–350 (2007)
10. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to information retrieval. Cambridge University Press (2008)
11. Naumann, F., Herschel, M.: An Introduction to Duplicate Detection. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2010)
12. Papadakis, G., Papastefanatos, G., Palpanas, T., Koubarakis, M.: Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. In: EDBT. pp. 221–232 (2016)
13. Papadakis, G., Svirsky, J., Gal, A., Palpanas, T.: Comparative analysis of approximate blocking techniques for entity resolution. PVLDB **9**(9), 684–695 (2016)
14. Papenbrock, T., Heise, A., Naumann, F.: Progressive duplicate detection. IEEE TKDE **27**(5), 1316–1329 (2015)
15. Simonini, G., Bergamaschi, S., Jagadish, H.V.: BLAST: a loosely schema-aware meta-blocking approach for entity resolution. PVLDB **9**(12), 1173–1184 (2016)
16. Simonini, G., Papadakis, G., Palpanas, T., Bergamaschi, S.: Schema-agnostic progressive entity resolution. In: IEEE ICDE. pp. 53–64 (2018)
17. Simonini, G., Zhu, S.: Big data exploration with faceted browsing. In: HPCS. pp. 541–544 (2015)
18. Whang, S.E., Marmaros, D., Garcia-Molina, H.: Pay-as-you-go entity resolution. IEEE TKDE **25**(5), 1111–1124 (2013)
19. Zhu, S., Fiameni, G., Simonini, G., Bergamaschi, S.: SOPJ: A scalable online provenance join for data integration. In: HPCS. pp. 79–85 (2017)