RulER: Scaling Up Record-level Matching Rules

Luca Gagliardelli
Università degli Studi
di Modena e Reggio Emilia
Italy
luca.gagliardelli@unimore.it

Giovanni Simonini
Università degli Studi
di Modena e Reggio Emilia
Italy
simonini@unimore.it

Sonia Bergamaschi
Università degli Studi
di Modena e Reggio Emilia
Italy
sonia.bergamaschi@unimore.it

ABSTRACT

Record-level matching rules are chains of similarity join predicates on multiple attributes employed to join records that refer to the same real-world object when an explicit foreign key is not available on the data sets at hand. They are widely employed from data scientists and practitioners that work with data lakes, open data, and data in the wild.

We present RulER, the first tool that allows to efficiently execute record-level matching rules on parallel and distributed systems—we developed that on top of Apache Spark to leverage on its vast ecosystem of libraries and wide adoption. In this demo, we show how RulER can be easily employed for data preparation tasks (i.e., to join data sets to be consumed by data analytic tasks) and to support the user in debugging record-level matching rules. Finally, we demonstrate how our execution strategy of the record-level matching rules—introduced by RulER—is up to 3 times faster than the naïve concatenation of similarity join predicates.

1 INTRODUCTION

Combining data sets that bare information about the same real-world objects is an everyday task for practitioners that work with structured and semi-structured data. Frequently (e.g., when dealing with data lakes or when integrating open data with proprietary data) data sets do not have explicit keys that can be used for a traditional *equi-join* [4, 8, 9]. When that happens, a common solution is to perform a *similarity join* [6], i.e., to join records that have an attribute value similar above a certain threshold, according to a given similarity measure, as in the following example:

Example 1.1 (Similarity Join)**.** *Given two product data sets, join all the record pairs with the Jaccard similarity of the product names above 0.8.*

A plethora of algorithms have been proposed in the last decades to efficiently execute the similarity join considering a single attribute, i.e., *attribute-level matching rules* (see [6] for a survey). At their core, all these algorithms try to prune the candidate pairs of records, on the basis of a single-attribute predicate—to alleviate the quadratic complexity of the problem.

Interestingly, only a few works had been focused on studying how to execute *record-level matching rules*, i.e., the combination of multiple similarity join predicates on multiple attributes (see section 2.1.1). Yet, this kind of rules permits to specify more flexible rules to match records, as in the following example:

Example 1.2 (Record-level matching rule)**.** *Given two product data sets, join all the record pairs that have a Jaccard similarity of the product names above 0.8, or that have a Jaccard similarity of the description that is above 0.6 and the edit distance of the manufacturer lower than 3.*

Furthermore, record-level matching rules can be used to represent *decision trees* [1], hence learned with machine learning algorithms when training data is available. As a matter of fact, a decision tree for binary classification (i.e., classification of *matching/not-matching* records) can be naturally represented with DNF (disjunctive normal form) predicates—the same consideration can be done for a forest of trees.

To the best of our knowledge, no techniques have been proposed to leverage on distributed and parallel computing for scaling record-level matching rules. The benefit is twofold: *(i)* distributed computation allows to scale to large data sets that cannot be handled with a single machine; *(ii)* parallel execution reduces the execution times (3 time faster in our experiments). As a matter of fact, being able to efficiently execute similarity join is crucial when time is a critical component, e.g., when users are involved in the process. For instance, in exploratory search in a data lake [7], users typically look for related data sets and low latency in performing similarity join is required for enabling the user's interactive exploration. Also, when debugging record-level matching rules, users typically try different configurations of similarity metrics, thresholds, and attributes. Hence, enabling fast execution of such rules can significantly save user' time.

Contribution. We present RulER, a tool that enables users to efficiently execute record-level matching rules to join large data sets on distributed parallel systems. In particular, we implemented RulER on top of Apache Spark[1], to leverage on its vast ecosystem of libraries and tools for data preparation, and machine learning. In this demonstration, we will showcase RulER on several real-world data sets; attendees will write their own matching rules through Jupyter notebooks, and explore and analyze the results. We will demonstrate how to employ RulER both in a data preparation pipeline and to debug record-level matching rules. We implemented state-of-the-art similarity join algorithms for Spark that can be employed to build chains of similarity join predicates (i.e., to mimic record-level matching rules). Attendees will run such similarity join chains and verify that RulER is actually significantly faster (up to an order of magnitude) and more convenient to program such rules thanks to its APIs.

2 TOOL ARCHITECTURE

2.1 Preliminaries

2.1.1 Record-level matching rules. In RulER, matching rules are written in Disjunctive Normal Form (DNF), i.e., as a *disjunction* (logical *OR*) of *conjunctions* (logical *AND*) of similarity join predicates on multiple attribute (i.e., at the *record level*). This design choice is driven by the fact that DNF matching rules are easy to read and thus to debug, in practice. Moreover, DNFs can be employed to represent the trained model of a decision tree (or

[1] https://spark.apache.org/

of a random forest), hence suitable for exploiting labelled data. In this demonstration, we focus on how to scale DNF matching rules and we do not investigate how to generate *good* DNFs (i.e., decision trees/random forests) starting from training data.

To the best of our knowledge, the only related work tackling this problem is [1], which focuses on single-node execution, borrowing optimization techniques from the traditional relational database approaches. Similarly, [5] focuses on how to optimize multi-attribute similarity join, but only for conjunctions of predicates (i.e., not for DNF).

2.1.2 Similarity join with prefix index. The naïve solution for similarity join (i.e., each predicate of a DNF) is to enumerate and compare every pair of records, i.e., highly inefficient and not feasible on large data sets. To reduce the task complexity, different approaches were proposed in literature [2, 10–12]. All these approaches adopt a filter-verification pattern: *(i)* first an index is used to obtain a set of pre-candidates (e.g., prefix filter); *(ii)* the pre-candidates are filtered using a set of filters that are fast to apply; *(iii)* the resulting candidate pairs are probed with the similarity function to generate the final result.

The most efficient technique to obtain the pre-candidates efficiently is the *prefix filter* [2], which works as follows. Given a set of strings (the values of an attribute in a table, for instance), a pre-processing function is applied to each string to obtain a set of elements (e.g., tokens or n-grams, etc.). These elements are then sorted according to a global order, usually by their not decreasing document frequency of the elements (i.e., $1/\#(strings\ containing\ that\ element)$)—typically infrequent elements yield fewer candidate pairs [2]. Then, for each sorted set of elements only the first π are considered, i.e., the *prefixes*. A pair of element $\langle r_i, r_j \rangle$ can be safely pruned if their prefixes have no common elements. The prefix size depends on the adopted similarity function and threshold. For example, the prefix filter for the overlap similarity is defined as follows: given two sets, r_i and r_j, and an overlap threshold t; if $|r_i \cap r_j| \geq t$, then there is at least one common element within the π_{r_i}-prefix of r_i and the π_{r_j}-prefix of r_j, where $r = |r_j| - t + 1$ and $s = |r_j| - t + 1$.

An example of prefix filtering is reported in Figure 1. The prefixes, assuming an overlap threshold $t = 4$ are highlighted in grey. Since the two prefixes do not share any element, the pair $\langle r_i, r_j \rangle$ can be pruned. The intuition behind this is that the 3 remaining elements to check can provide at most a similarity of 3, that is not enough to reach the requested threshold t.

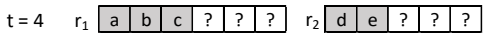

Figure 1: Prefix filtering example: The pair $\langle r_i, r_j \rangle$ can be pruned since the prefixes (in grey) have no common elements. The elements to check can provide at most an overlap similarity of 3 (or a Jaccard similarity of 3/8).

The prefix filter can be adapted to work with many similarity measures like Jaccard, Dice, Cosine, Overlap [11] and Edit Distance [10], and it is employed by best performing similarity join algorithms [6].

The state-of-the-art distributed and parallel similarity join algorithms [3] partition the candidate pair of records according to the entry in the prefix index, i.e., for each element in the index, all the corresponding pairs of candidates are assigned to a computational node—more optimizations can be performed, but at their core, this is how parallelization is achieved by existing

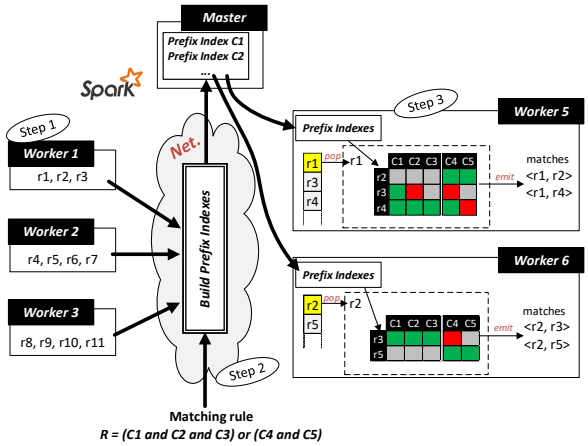

Figure 2: RulER execution model: green cells represents executed and passed rules; red cells executed that do not pass the rules; grey cells not executed rules.

algorithms. So, if two different similarity join conditions are considered (e.g., two different similarity measures on two different attributes), existing algorithms would create two different prefix indexes and generate two completely different parallelization strategies. Thus, given a fixed number of computational nodes, by employing existing algorithms, the only way to get the complete result of multiple similarity joins (i.e., the predicates of the matching rule) is to perform the joins in series and then combine the result sets.

2.2 The RulER execution model

The main intuition of RulER is to exploit the prefix indexes—one prefix index for each predicate of the matching rule—to build a graph structure, which is then employed to iterate over the records (the nodes of the graph), efficiently applying the rules and keep only the candidates (the edges of the graph) that pass the whole rule. In other words, RulER adopts a record-based parallelization approach; in contrast to the existing algorithms, which adopt a prefix-based parallelization approach on a single predicate at a time.

Example 2.1 (Distributed and parallel matching rule with RulER)**.** *Given a matching rule $R = (C1 \wedge C2 \wedge C3) \vee (C4 \wedge C5)$, in which each Cx is a similarity join predicate (e.g., Jaccard Similarity $title \geq 0.8$). An example of how RulER executes R is outlined in Figure 2. First, a prefix index is built on the basis of the record-level matching rules expressed in the main matching rule R. Then, the index is distributed to each worker. Each worker iterates over each record in its partition extracting the possible candidates from the prefix index. The rules are applied to each candidate. If more rules are in or it is possible to avoid computing the other rules when one of them is verified, e.g., with r1-r2 the rule ($C1 \wedge C2 \wedge C3$) is not verified since the pair passes the rule ($C4 \wedge C5$). Otherwise, if more rules are in and, it is possible to avoid the computation when one of them fails, for example for the pair r1-r3 C2 fails, so C3 has not to be computed.*

2.2.1 The algorithm. The RulER matching rule algorithm is outlined in Algorithm 1. The presented algorithm is the self-join version for sake of the presentation; adapting it for joining two different data sets is straightforward. The algorithm takes as

input a collection of records and a record-level matching rule $\mathcal{M}$ and gives as output the set of record pairs that satisfy $\mathcal{M}$. Recall that $\mathcal{M}$ is in DNF, i.e., it is composed of sets of predicates P_j in *logical or*, each set P_j contains predicates p_k in *logical and*. First of all, the values of attributes are converted into sets of elements (Line 1) according to the matching rule requirements (e.g., n-grams, trigrams, tokens, etc.); then the prefix indexes are built to find the candidate pairs (line 2)—one prefix index is needed for each predicate p_k of the matching rule. The prefix indexes are sent in broadcast to each node (line 3) to be available to each computational node (called *worker*). Then, each worker iterates over its portion of records (lines 5-6), and performs the following operations for each record r_i. First, a set of candidates for r_i is initialized as an empty set C_{r_i} (line 7). Second, for each set P_j, a set of candidates C_{P_j} is initialized as an empty set (lines 8-9) and for each $p_k \in P_j$ the candidates C_{r_i,p_k} that can match with r_i are extracted using the prefix indexes (lines 10-11). Third, the candidates C_{r_i,p_k} are pruned by removing those that already passed one of the previous P_j set of predicates (line 14), and those that did not passed previous $p_k \in P_j$ predicates (lines 15-16). Fourth, the retained candidates are probed with other filters that further improve the efficiency of the overall process (e.g., length filter, position filter, etc. [10, 11]) according to the rule (line 18). Since p_k is in *logical and* with the previous predicates, only the candidates that pass the filters are kept. Finally, the resulting candidates from P_j are added to C_{r_i} (line 20).

2.2.2 Difference operator. RulER implements a method to perform the difference between the result pairs generated by two matching rules, i.e., given two matching rules $\mathcal{M}_1$, $\mathcal{M}_2$ it efficiently performs the difference of the result sets $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$. Since the algorithm works at record level it is possible to perform a fine control on the application of the rules: when a record r_i is processed first $\mathcal{M}_1$ is checked, so to get $res(\mathcal{M}_1, r_i)$; then, $\mathcal{M}_2$ is applied only on the records $r_j \in res(\mathcal{M}_1, r_i)$, avoiding to compute it on the whole pairs again, retaining only the records r_j that do not satisfy $\mathcal{M}_2$ (i.e., $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$).

Algorithm 1 RulER core

Input: R collection of records to join
Input: $\mathcal{M}$ matching rule in DNF
Output: C, the pairs of records that can satisfy $\mathcal{M}$

1: $R_T \leftarrow getElements(R, \mathcal{M})$
2: $I \leftarrow buildPrefixIndexes(R_T, \mathcal{M})$
3: **broadcast**(I)
4: $C \leftarrow \{\}$ //Candidate pairs
5: **foreach partition** $part \in R_T$
6: | **for each** $r_i \in part$ **do**
7: | | $C_{r_i} \leftarrow \{\}$ //Candidates for r_i
8: | | **for each** $P_j \in \mathcal{M}$ **do** //For each set of predicates in logical or
9: | | | $C_{P_j} \leftarrow \{\}$ //Candidates that satisfy P_j
10: | | | **for each** $p_k \in P_j$ **do** //For each predicate in logical and
11: | | | | $C_{r_i,p_k} \leftarrow I(p_k, r_i)$ //Gets the candidates from the prefix index
12: | | | | /*Removes candidates that already passed previous predicates in *or*
13: | | | | and those that did not pass previous predicates in *and**/
14: | | | | $C_{r_i,p_k} \leftarrow C_{r_i,p_k} - C_{r_i}$
15: | | | | **if** $C_{P_j} \neq \emptyset$ **then**
16: | | | | | $C_{r_i,p_k} \leftarrow C_{r_i,p_k} \cap C_{P_j}$
17: | | | | /*Applies filters (length, positional, ...)*/
18: | | | | $C_{P_j} \leftarrow applyFilters(r_i, C_{r_i,p_k}, p_k)$
19: | | | $C_{r_i} \leftarrow C_{r_i} \cup C_{P_j}$
20: | | $C.append(C_{r_i})$

3 DEMONSTRATION SCENARIO

During the demonstration, participants will try RulER[2] on several scenarios and data sets by means of Jupyter Notebooks[3]. Users will be guided in: defining custom matching rules for a practical data preparation task (Section 3.1); debugging a matching rule (Section 3.2); and compare the efficiency of RulER w.r.t. existing state-of-the-art similarity joins (Section 3.3). Multiple scenarios and data sets (e.g., products, movies, books, finance, etc.) on which it is possible to try our RulER are available, but for sake of the presentation we describe only some of them in the following.

3.1 Data preparation scenario

In this scenario we use two movies data sets `Rotten Tomatoes`[4] and `Roger Erbert`[5]. The former gather users' ratings about movies, the latter critics' ratings. There is no foreign key between the two data sets. We ask the attendee to discover if there is a correlation between the ratings given by the users with the ones given by the critics. To do that, we need to define a matching rule to integrate the two data sets, then we compute the Pearson correlation on the obtained results. We ask to the attendee to write a record-level matching rule, for instance: ("*movie_name*", "*Title*", *JS*, 0.8) ∧ ("*actors*", "*Cast*", *JS*, 0.5), That means that the JS between the names of the movies must be greater or equal than 0.8 and the JS between the actors of the movies must be greater or equal than 0.5. After the matching, it is possible to obtain a scatter plot of the ratings, and compute the Pearsons correlation rating given by critics and the rating given by users on the same movie.

Figure 3 shows how simple is to use RulER. The user has just to: include RulER library (line 1), load the data as Spark Dataframe (lines 3-4), define the rules (lines 6-7) and combine them to obtain the final matching rule (line 9). Finally, the matching rule can be used to join the two dataframes with the *joinWithRules* method (line 11).

3.2 Matching rule debugging scenario

In this scenario we show how RulER can be used to efficiently debug different matching rules by using the difference operator. In particular, given two matching rules $\mathcal{M}_1$, $\mathcal{M}_2$ we will show how to use RulER to find the matches provided by the first rule that are not present in the matches provided by the second one, i.e. how to perform the difference between the two result sets $res(\mathcal{M}_1) \setminus res(\mathcal{M}_2)$. An example is shown in Figure 4. First, two matching rules are defined: in the first one the records are aligned by using the *title* and the *director* of the movies, while in the latter by using the *title* and the *cast*. Then, a new debugging rule is created using the difference operator defined in RulER. Finally, the debugging rule is applied to the dataset, obtaining the matches generated by $m1$ that are not generated by $m2$.

3.3 RulER efficiency demonstration

In this scenario we connect to a Spark cluster and use a subset of the IMDB data set[6] that contains records about movies, providing different fields that can be used to generate matching rules (e.g. movie title, cast, director, plot, etc.).

[2]We implemented RulER in Scala and made it open source: https://github.com/Gaglia88/ruler
[3]https://jupyter.org
[4]https://www.rottentomatoes.com
[5]https://www.rogerebert.com
[6]https://www.imdb.com

Figure 3: RulER usage example.

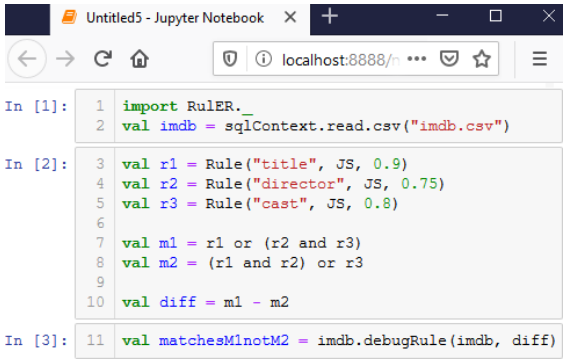

Figure 4: Rule's debugging example.

We show how difficult is to write a complex rule by using existing similarity join algorithms w.r.t. the use of RulER. Moreover, RulER is much more faster. Figure 6 presents this scenario. To execute the rule as a similarity join chain, we use EDJoin [10] to perform a similarity join based on Edit Distance and PPJoin [11] to perform a similarity join based on Jaccard Similarity. The matching rule written by chaining similarity joins is expressed in lines 13-16. Note that, each rule provides a partial result and then the partial results of each rule have to be combined (line 16). If two rules are in *and* the partial results have to be intersected; if they are in *or* they have to be merged. To merge two candidate sets and avoid duplicates, a *distinct* operation has to be performed. The *distinct* and the *intersection* are very expensive operations in Spark because they require a *shuffling* since the same pairs have to be computed by the same worker.

Figure 5: Execution time of RulER vs the execution time of the join chain in Figure 6.

Figure 5 shows the execution time of both the solutions. For the join chain, the prefix indexing and join times are computed as the sum of each join. The RulER indexing time is higher due to the time requested to broadcast the index, but the join time is faster. Moreover, RulER does not need to merge the partial results, that is the costly task of the join chain, which makes it highly inefficient.

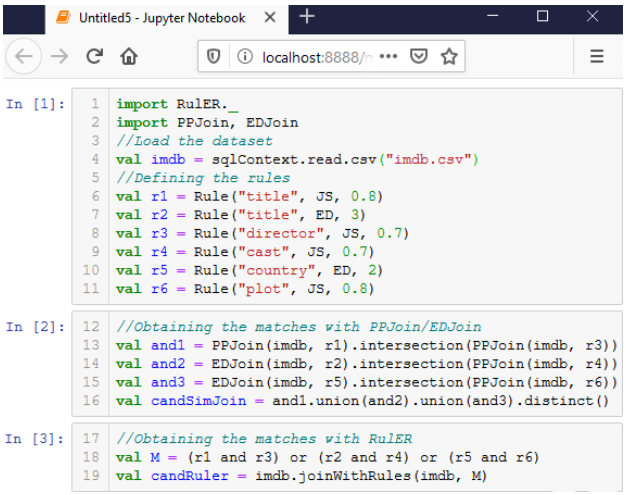

Figure 6: Join chain vs RulER execution.

REFERENCES

[1] Adel Ardalan, AnHai Doan, Aditya Akella, et al. 2018. Smurf: self-service string matching using random forests. *PVLDB* 12, 3 (2018), 278–291.

[2] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *ICDE*. 5–5.

[3] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set similarity joins on mapreduce: An experimental survey. *PVLDB* 11, 10 (2018), 1110–1122.

[4] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT 2019*. 602–605.

[5] Guoliang Li, Jian He, Dong Deng, and Jian Li. 2015. Efficient similarity join and search on multi-attribute data. In *SIGMOD*. 1137–1151.

[6] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An empirical evaluation of set similarity join techniques. *PVLDB* 9, 9 (2016), 636–647.

[7] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *PVLDB* 12, 12 (2019), 1986–1989.

[8] Giovanni Simonini, Luca Gagliardelli, Sonia Bergamaschi, and H. V. Jagadish. 2019. Scaling entity resolution: A loosely schema-aware approach. *Inf. Syst.* 83 (2019), 145–165.

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

[10] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

[11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM TODS* 36, 3 (2011), 15.

[12] Song Zhu, Giuseppe Fiameni, Giovanni Simonini, and Sonia Bergamaschi. 2017. SOPJ: A Scalable Online Provenance Join for Data Integration. In *HPCS 2017*. 79–85.