Scaling Up Record-level Matching Rules

Luca Gagliardelli, Giovanni Simonini, and Sonia Bergamaschi

University of Modena and Reggio Emilia, Modena, Italy
<name.surname>@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 by data scientists and practitioners that work with data lakes, open data, and data in the wild. In this work we present a novel technique that allows to efficiently execute record-level matching rules on parallel and distributed systems and demonstrate its efficiency on a real-wold data set.

Keywords: Data Integration · Entity Resolution · Parallel similarity join

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* [12,13,7,9]. When that happens, a common solution is to perform a *similarity join* [10], 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. (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 [10] 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). Yet, this kind of rules permits to specify more flexible rules to match records, as in the following example:

Example 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* [2], 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 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 times 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 [11], 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's time.

Contribution. In this work we present a technique that is able to use different similarity measures to apply record-level matching rules efficiently. Moreover, we present an algorithm, *RulER*, to efficiently run record-level matching rules on MapReduce-like systems, to take full advantage of a parallel and distributed computation.

The rest of the paper is organized as follow: Section 2 provides the preliminaries. Then, in Section 3 we present our novel technique. Section 4 shows the experimental results. Finally, in Section 5 we draw the conclusions.

2 Preliminaries and Related Work

This section describes the fundamental concepts and the related work.

2.1 Matching Rule

We define a matching rule $\mathcal{R}$ 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 of a random forest), hence suitable for exploiting labeled data. In this work, 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, which is the focus of Ardalan et al. [2]—Ardalan et al. do not investigate the parallel and distributed execution of matching rules.

2.2 Set Similarity Join

A record r_i is considered as a set of elements identified by a unique identifier. Different techniques can be employed to generate the elements from the values of a record, for example, each word can be considered as a token or it is possible to generate the n-grams, etc. Formally, given a collection of records, a similarity function *sim* and a similarity threshold t, the goal of set similarity join is to find all the possible pairs of records $\langle r_i, r_j \rangle$ such that $sim(r_i, r_j) \geq t$.

A naïve solution to perform the set similarity join is to enumerate and compare every pair of records, but this process is highly inefficient and not feasible in the Big Data context. To reduce the task complexity different approaches were proposed in literature [4,3,16,15]. All these approaches adopt a filter-verification approach: (1) first an index is used to obtain a set of pre-candidates; (2) the pre-candidates are filtered using a set of pre-defined filters; (3) the resulting candidate pairs are probed with the similarity function to generate the final results.

The most used filters are: prefix filter, length filter, and positional filter. All these filters can be adapted to work with different similarity measures: Dice, Cosine, Jaccard Similarity, Edit Distance and Overlap Similarity [10,15,16].

Prefix filter A key technique to perform the set similarity join efficiently is the *prefix filter* [4]. First of all, given a collection of records (i.e., sets of elements) their elements are sorted according to a global order $\mathcal{O}$, usually the document frequency of the tokens (i.e., how many documents contain that token) that is a heuristic that helps to reduce the number of comparisons [4]. Then, for each sorted set, only the first π elements are considered, i.e., the *prefixes.* A pair $\langle r_i, r_j \rangle$ can be safely pruned if their prefixes have no common elements. The prefix size depends on the similarity threshold and the similarity function. 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 token 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 how prefix filter works is reported in Figure 1. The prefixes for overlap threshold $t = 4$ are highlighted in grey. Since the two prefixes do not share any token, the pair $\langle r_i, r_j \rangle$ can be pruned. The intuition behind this is that the 3 remaining tokens to check can provide at most a similarity of 3, that is not enough to reach the requested threshold t.

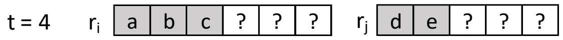

Fig. 1. Prefix filter.

Length filter A filter that is commonly used in conjunction with the prefix filter is the length filter [1]. Normalized similarity functions (e.g., Jaccard, Cosine, Dice, ED) depend on the set size, thus it is possible to exploit it to prune the pairs generated with the prefix filter. For the Jaccard Similarity the length filter is defined as: a set of elements r can reach Jaccard threshold t only with a set s of size $lb_r \leq |s| \leq ub_r$ ($lb_r = t \cdot |r|, ub_r = \frac{|r|}{|t|}$); for example, if $|r| = 10$ and $t = 0.8$, then $8 \leq |s| \leq 12$ is required.

Positional filter The positional filter [16] reasons over the matching position of tokens in the prefix. Given a pair of sets of sorted elements it checks the positions of their common tokens in the prefix, if the remain tokens to check are not enough to reach the threshold, it prunes the pair. Since it needs to scan the tokens in the prefix, this filter is more expensive than prefix and length filters, so usually it is applied only on the pairs that already passed them.

An example of how positional filter works is provided in Figure 2. The pair $\langle r_i, r_j \rangle$ passes both length and prefix filters. The first match in r_i occurs in position 1 (counting from 0), thus only 8 tokens of r_j are left to match tokens of r_i, and the pair can be filtered because it can never reach the requested threshold $t = 9$.

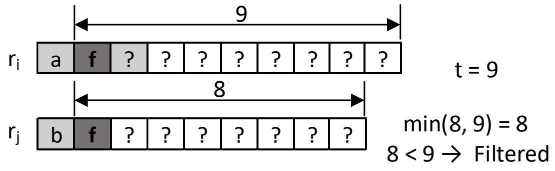

Fig. 2. Positional filter example.

Prefix filter based set similarity join An example of how a prefix filter based set similarity join works is outlined in Figure 3. Starting from a document collection, the documents are transformed in sets of elements (e.g., tokens, n-grams, etc.) and sorted according to a global order (1). Then, using the prefixes (highlighted in gray) an inverted index is built, i.e., the prefix index (2). From the prefix index, a set of pre-candidate pairs is built (3), i.e., each pair of profiles that appear together in at least one entry of the prefix index. The pre-candidate pairs are filtered using different filters (e.g., length filter, positional filter, etc.) that are fast to compute and let to discard the pairs that cannot reach the threshold (4). Finally, the pairs that pass all the filters (i.e., candidate pairs) are probed with the similarity function, and only those that have a similarity above the threshold are retained (5).

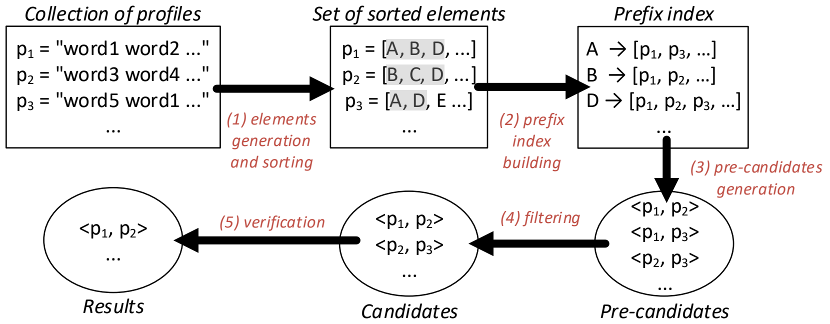

Fig. 3. Prefix filter based Similarity Join process.

3 RulER

In this section, we present our method *RulER* to efficiently scale record-level matching rules over big data sets. The presented algorithm is the self-join version for the sake of the presentation; adapting it for joining two different data sets is straightforward.

3.1 Baseline algorithm

Given a matching rule $\mathcal{R}$ a naïve solution to perform it is to process each predicate as a single similarity join and then intersect/merge the obtained results according to the requirements. In particular, we adopted two algorithms to perform the similarity joins: *PPJoin* [16] and *EDJoin* [15]. Both algorithms employ *prefix filter* (see Section 2) to find candidate pairs. *PPJoin* is considered one of the best performing similarity join algorithm [10], also its parallel implementation (i.e., Vernica Join) has demonstrated to be one of the best performing

Algorithm 1 PPJoin/EDJoin

```
Input: $R$ collection of records to join
Input: $\mathcal{P}$ predicate that contains the join attribute, the threshold, and the elements
    pattern (i.e., tokens, n-grams, etc.)
Output: $C$, the pairs of records that can satisfy $\mathcal{P}$
$R_T \leftarrow getSortedElements(R, \mathcal{P})$ //Transforms the records in set of sorted ele-
    ments (i.e., tokens, n-grams, etc.)
$I \leftarrow buildPrefixIndex(R_T, \mathcal{P})$ //Build prefix index
$C \leftarrow$ flatMap $B_i \in I$ //For each entry of the prefix index
    for each $\langle r_j, r_k \rangle \in B_i (r_j \neq r_k)$ do
        if $passLengthFilter(r_j, r_k, \mathcal{P})$ then
            if $passPositionalFilter(r_j, r_k, \mathcal{P})$ then
                $emit(\langle r_j, r_k \rangle)$
```

for distributed computing [6]. It can work with different similarity measures like Jaccard Similarity, Dice and Cosine. *EDJoin* adapts the *PPJoin* concepts to work with the Edit Distance. We adapted both algorithms to work on Spark as proposed in [14] for *PPJoin*(i.e., Vernica Join).

The distributed algorithm to perform PPJoin/EDJoin is presented in Algorithm 1. First, the records are transformed into sets of sorted elements according to the predicate requirements (line 1). The prefix index (see Subsection 2.2) is built generating an inverted index that groups all the records that share at least one token in their prefix. Then, the algorithm iterates over each entry of the prefix index B_i, probing each pair of records $\langle r_i, r_j\rangle \in B_i$ with the appropriate filters according to the predicate $\mathcal{P}$.

Algorithm 2 outlines the baseline algorithm (i.e., *JoinChain*). A rule in DNF format is composed of different blocks P_i of predicates that are in logical *OR*, each of these blocks contains one or more predicates p_i that are in logical *AND*. First, the algorithm iterates over the P_i blocks (line 2) and for each of them initializes a set of candidates C_{P_i} (line 3). Then, each simple predicate $p_j \in P_i$ is used to apply a similarity join on the record collection R according to requirements (lines 6-9). The result of a P_i block is given by the intersection of the results provided by each similarity join applied with the predicate $p_j \in P_i$ (lines 10-13). The final candidate set is computed by merge the results of each P_i block (line 14). In the end, the candidates are verified with a verify function that ensures that all predicates are respected (line 15).

Algorithm 2 JoinChain

```
Input: $R$ collection of records to join
Input: $\mathcal{M}$ matching rule in DNF form
Output: $M$, the pairs of records that satisfy $\mathcal{M}$
 1: $C \leftarrow \{\}$
 2: for each $P_i \in \mathcal{M}$ do //For each block of predicates in or
 3:     $C_{P_i} \leftarrow \{\}$ //Set of candidate pairs for $P_i$
 4:     for each $p_j \in P_i$ do //For each single predicate in and
 5:         $C_{p_j} \leftarrow \{\}$ //Set of candidate pairs for $p_j$
 6:         if $p_i.type = ED$ then
 7:             $C_{p_j} \leftarrow EDJoin(R, p_j)$ //Get candidate pairs with EDJoin
 8:         else
 9:             $C_{p_j} \leftarrow PPJoin(R, p_j)$ //Get candidate pairs with PPJoin
10:         if $C_{P_i}.isEmpty$ then //Intersects candidates with previous ones
11:             $C_{P_i} \leftarrow C_{p_i}$
12:         else
13:             $C_{P_i} \leftarrow C_{P_i} \cap C_{p_j}$
14:     $C \leftarrow C \cup C_{P_i}$ //Merge candidates with previous ones
15: return $verify(C, \mathcal{M})$
```

3.2 The RulER Algorithm

Algorithm 2 has three main drawbacks:

(i) the *intersect* operation (line 13) is expensive in MapReduce-like systems, because it generates *shuffle*;

(ii) a predicate is independently checked by the others. For example, given a matching rule $M = (C1 \wedge C2) \vee (C3 \wedge C4)$ in which each Cx is a similarity join predicate (e.g., Jaccard Similarity $title \geq 0.8$). A pair $\langle r_i, r_j \rangle$ is probed with all predicates even if it fails/passes one of them. For example, if the pair passes the predicate $(C3 \wedge C4)$ it is not necessary to probe it with $(C1 \wedge C2)$. Or, if it fails with the predicate $C1$ it is not necessary to probe it with $C2, C3$.

(iii) Vernica Join [14], employed in the implementation of *JoinChain* algorithm (Algorithm 2 lines 7, 9), produces duplicates [6] that have to be removed. If a pair of records appears in more prefix entries, it is processed and emitted multiple times (Algorithm 1 lines 3-7).

We solved these problems in our *RulER* algorithm. 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 to 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.

The *RulER* matching rule execution algorithm is outlined in Algorithm 3. 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. [16,15]) according to the rule (line 18). Since p_k is in *logical and* with the previous predicates, only the candidates

https://spark.apache.org/docs/2.1.0/programming-guide.html#shuffle-operations

Algorithm 3 RulER

```
Input: $R$ collection of records to join
Input: $\mathcal{M}$ matching rule in DNF
Output: $C$, the pairs of records that 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: map 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:         $C.append(C_{r_i})$
21: return $verify(C, \mathcal{M})$
```

that pass the filters are kept. The resulting candidates from P_j are added to C_{r_i} (line 20). Finally, the candidates are verified (line 21).

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* ≥ 0.8); an example of how *RulER* executes R is outlined in Figure 4. 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 $\langle r1, r2 \rangle$ 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 $\langle r1, r3 \rangle$, C2 fails, so C3 has not to be computed.

4 Experimental Evaluation

In this section we evaluate *RulER* with respect to *JoinChain* (see Section 3). In particular, the experimental evaluation aims to answer the following questions:

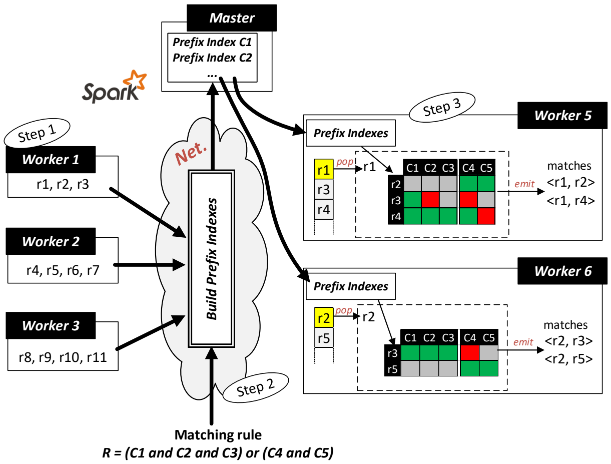

Fig. 4. *RulER* execution model: green cells represents executed and passed rules; red cells executed that do not pass the rules; grey cells not executed rules.

*Q*1: *What is the performance of RulER in terms of execution time compared to JoinChain (i.e., the naïve solution)?* (Section 4.2)

*Q*2: *How does RulER scale when varying the number of machines available for the record-level matching rule processing?* (Section 4.3)

4.1 Experimental Setup

All the experiments are performed on a ten-node cluster; each node has two Intel Xeon E5-2670v2 2.50 GHz (20 cores per node) and 128 GB of RAM, running Ubuntu 14.04. All the software is implemented in Scala 2.11.8 and available at [8]. To assess the performance of the state-of-the-art meta-blocking methods we re-implemented all of them for running on Apache Spark as well. We employ Apache Spark 2.1.0, running 3 executors on each node, reserving 30 GB of memory for

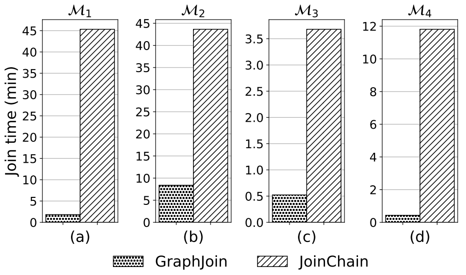

Fig. 5. Execution times of *RulER* and *JoinChain* with the rules reported in Table 1.

the master node. We set the default parallelism to twice the number of cores as suggested by best practice.

We employed the `ombd` data set [5] to evaluate the performance of *RulER* and *JoinChain*. It contains 2.3 millions of records about movies collected from *omdbapi.com*. This data set has a good variety of attributes (e.g., title, cast, director, writer, plot, etc.) on which it is possible to define a different kind of record-level matching rules. Moreover, it contains a sufficient number of records that make it suitable to test the performance with Spark.

The goal of our experiments is to compare the efficiency of the algorithms, not to find good matching rules. Moreover, with *RulER* it would be possible to define an order for the application of the predicates that minimize the execution time and the number of performed comparisons, but we do not explore this aspect in this work.

4.2 *RulER* vs *JoinChain*

The goal of this experiment is to compare the efficiency of *RulER* (Algorithm 3) and *JoinChain*(Algorithm 2). Both algorithms can be employed to apply a record-level matching rule $\mathcal{M}$. In this experiment we apply the rules presented in Table 1 on the `omdb` data set. Since both algorithms use the same functions to extract the elements from the records, to generate the prefix indexes and to verify the candidate pairs, we analyze here only the time requested to perform the *join* operation. All the experiments are performed on a single node.

Figure 5 reports the execution times of *RulER* and *JoinChain*with the rules reported in Table 1. *RulER* is always significantly faster than *JoinChain*: 24× with $\mathcal{M}_1$ (Figure 5(a)), 5× with $\mathcal{M}_2$ (Figure 5(b)), 7× with $\mathcal{M}_3$ (Figure 5(c)), 27× with $\mathcal{M}_4$ (Figure 5(d)). Moreover, Figure 6 shows the number of comparisons performed by both algorithms. Also in this case, *RulER* works better than *JoinChain* performing less comparisons: $21.6 \cdot 10^6$ less for $\mathcal{M}_1$ (Figure 6(a)), $23.0 \cdot 10^6$ less for $\mathcal{M}_2$ (Figure 6(b)), $3.8 \cdot 10^6$ less for $\mathcal{M}_3$ (Figure 6(c)), $45.1 \cdot 10^6$ less for $\mathcal{M}_3$ (Figure 6(d)).

We conclude that *RulER* is always faster than *JoinChain*.

https://spark.apache.org/docs/latest/tuning.html

	Rule	**Candidates**	**Matches**
$\mathcal{M}_1$	$(Title, 0.9, JS) \wedge (Cast, 0.8, JS)$	1725885	53023
$\mathcal{M}_2$	$(Title, 0.9, JS) \vee (Cast, 0.8, JS)$	990278774	253593213
$\mathcal{M}_3$	$((Cast, 0.9, JS) \wedge (Director, 0.9, JS) \wedge (Writer, 0.9, JS)) \vee (Title, 0.9, JS)$	61987445	34798235
$\mathcal{M}_4$	$(Director, 0.9, JS) \wedge (Title, 2, ED)$	1133134	252935

Table 1. Matching rules employed in the experiments. For each rule the number of candidate pairs obtained after the filters (i.e., prefix filter, length filter, positional filter, see Section 2) is reported, together with the number of final matching pairs.

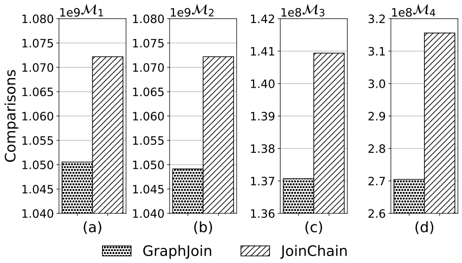

Fig. 6. Number of comparisons performed by *RulER* and *JoinChain* with the rules reported in Table 1.

4.3 *RulER* scalability

Finally, we assess the scalability of *RulER* by varying the number of nodes in the cluster (1, 3, 5, 7 and 10 nodes). For this experiment we apply the rules described in Table 1 on the `omdb` data set.

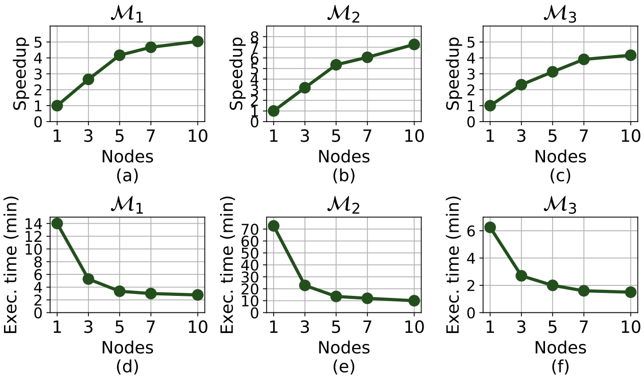

Fig. 7. Execution time and speedup of *RulER* with the rules reported in Table 1.

Figure 7 shows the scalability and the speedup of the whole process. For each step, we observe at least a 50% reduction of execution time from 1 to 3 nodes. Then, the execution times continuously decrease until reaching an overall speedup on 10 nodes of: 5.0× for $\mathcal{M}_1$, 7.3× for $\mathcal{M}_2$, and 4.16× for $\mathcal{M}_3$. $\mathcal{M}_2$ is the rule that takes more advantage in the increase of worker' nodes because it performs more comparisons than the others, as shown in Table 1.

5 Conclusion

In this work, We tackled the problem of performing record-level matching rules. We presented two solutions to perform them on parallel and distributed systems:

a baseline one (i.e. *JoinChain*) implemented by using existing solutions, and a novel approach (i.e. *RulER*) that optimizes the execution of the rules. We conducted a thorough experimental evaluation, demonstrating the efficiency of the proposed approach, which always outperforms the baseline solution in terms of execution time and number of comparisons. In the future, we plan to extend our system to automatically find the optimal execution order of the predicates that compose a rule.

References

1. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proceedings of the 32nd international conference on Very large data bases. pp. 918–929. VLDB Endowment (2006)
2. Ardalan, A., Doan, A., Akella, A., et al.: Smurf: self-service string matching using random forests. PVLDB **12**(3), 278–291 (2018)
3. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th international conference on World Wide Web. pp. 131–140. ACM (2007)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: 22nd International Conference on Data Engineering (ICDE'06). pp. 5–5. IEEE (2006)
5. Das, S., Doan, A., G. C., P.S., Gokhale, C., Konda, P.: The magellan data repository. https://sites.google.com/site/anhaidgroup/projects/data
6. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.C.: Set similarity joins on mapreduce: An experimental survey. Proceedings of the VLDB Endowment **11**(10), 1110–1122 (2018)
7. Gagliardelli, L., Simonini, G., Beneventano, D., Bergamaschi, S.: Sparker: Scaling entity resolution in spark. In: EDBT 2019. pp. 602–605 (2019)
8. Gagliardelli, L., Simonini, G., Zhu, S., Bergamaschi, S.: Sparker: an entity resolution tool for apache spark (2017), `https://github.com/Gaglia88/sparker`
9. Gagliardelli, L., Zhu, S., Simonini, G., Bergamaschi, S.: Bigdedup: a big data integration toolkit for duplicate detection in industrial scenarios. In: 25th International Conference on Transdisciplinary Engineering (TE2018). vol. 7, pp. 1015–1023 (2018)
10. Mann, W., Augsten, N., Bouros, P.: An empirical evaluation of set similarity join techniques. Proceedings of the VLDB Endowment **9**(9), 636–647 (2016)
11. Nargesian, F., Zhu, E., Miller, R.J., Pu, K.Q., Arocena, P.C.: Data lake management: challenges and opportunities. PVLDB **12**(12), 1986–1989 (2019)
12. Simonini, G., Gagliardelli, L., Bergamaschi, S., Jagadish, H.V.: Scaling entity resolution: A loosely schema-aware approach. Inf. Syst. **83**, 145–165 (2019)
13. Simonini, G., Papadakis, G., Palpanas, T., Bergamaschi, S.: Schema-agnostic progressive entity resolution. IEEE TKDE **31**(6), 1208–1221 (2019)
14. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 495–506. ACM (2010)
15. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
16. Xiao, C., Wang, W., Lin, X., Yu, J.X., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM TODS **36**(3), 15 (2011)