Information Systems 93 (2020) 101565

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Three-dimensional Entity Resolution with JedAI

George Papadakis [a,*], George Mandilaras [a], Luca Gagliardelli [b], Giovanni Simonini [b], Emmanouil Thanos [c], George Giannakopoulos [d], Sonia Bergamaschi [b], Themis Palpanas [e], Manolis Koubarakis [a]

[a] National and Kapodistrian University of Athens, Greece
[b] University of Modena and Reggio Emilia, Italy
[c] KU Leuven, Belgium
[d] NCSR "Demokritos", Greece
[e] University of Paris & French University Institute (IUF), France

ARTICLE INFO

Article history:
Received 6 May 2020
Received in revised form 22 May 2020
Accepted 23 May 2020
Available online 27 May 2020
Recommended by D. Shasha

Keywords:
Entity Resolution
Blocking
Matching
Clustering
Batch methods
Progressive methods
Massive parallelization

ABSTRACT

Entity Resolution (ER) is the task of detecting different entity profiles that describe the same real-world objects. To facilitate its execution, we have developed JedAI, an open-source system that puts together a series of state-of-the-art ER techniques that have been proposed and examined independently, targeting parts of the ER end-to-end pipeline. This is a unique approach, as no other ER tool brings together so many established techniques. Instead, most ER tools merely convey a few techniques, those primarily developed by their creators. In addition to democratizing ER techniques, JedAI goes beyond the other ER tools by offering a series of unique characteristics: (i) It allows for building and benchmarking millions of ER pipelines. (ii) It is the only ER system that applies seamlessly to any combination of structured and/or semi-structured data. (iii) It constitutes the only ER system that runs seamlessly both on stand-alone computers and clusters of computers — through the parallel implementation of all algorithms in Apache Spark. (iv) It supports two different end-to-end workflows for carrying out batch ER (i.e., budget-agnostic), a schema-agnostic one based on blocks, and a schema-based one relying on similarity joins. (v) It adapts both end-to-end workflows to budget-aware (i.e., progressive) ER. We present in detail all features of JedAI, stressing the core characteristics that enhance its usability, and boost its versatility and effectiveness. We also compare it to the state-of-the-art in the field, qualitatively and quantitatively, demonstrating its state-of-the-art performance over a variety of large-scale datasets from different domains.

The central repository of the JedAI's code base is here: https://github.com/scify/JedAIToolkit .

A video demonstrating the JedAI's Web application is available here: https://www.youtube.com/watch?v=OJY1DUrUAe8.

1. Introduction

Entity Resolution (ER) constitutes a core data integration task, with many applications that range from knowledge bases to question answering [1–3]. Its goal is to detect duplicate entity profiles that describe the same real-world objects. Due to the lack of a unique identifier per real-world object, ER can only be resolved by overcoming two main challenges: (i) the inherently quadratic computational cost, $O(n^2)$, as in the worst case, every entity profile should be compared with all others, and (ii) the noise and/or ambiguity in the attribute names and values that describe each entity profile, hampering the detection of duplicates.

Existing ER systems [3–6] attempt to tackle the above two challenges in a partial (unidimensional) way. In essence, the end-to-end pipelines they construct apply a batch, serialized processing that relies heavily on schema and domain knowledge to optimize two main steps [3]: (i) *Blocking*, which groups together similar entity profiles, restricting the computational cost to the comparison of a subset of the input entities, and (ii) *Matching*, which applies complex similarity measures and rules in order to distinguish between matching and non-matching entities. Each system, though, typically implements a few methods (primarily those proposed by its creators), and requires heavy user involvement. Yet, not all users are capable of configuring and using these ER systems. As a result, the potential user base of such systems is

* Corresponding author.
E-mail addresses: gpapadis@di.uoa.gr (G. Papadakis), gmandi@di.uoa.gr (G. Mandilaras), luca.gagliardelli@unimore.it (L. Gagliardelli), emmanouil.thanos@kuleuven.be (E. Thanos), ggianna@iit.demokritos.gr (G. Giannakopoulos), sonia.bergamaschi@unimore.it (S. Bergamaschi), themis@mi.parisdescartes.fr (T. Palpanas), koubarak@di.uoa.gr (M. Koubarakis).

https://doi.org/10.1016/j.is.2020.101565
0306-4379/

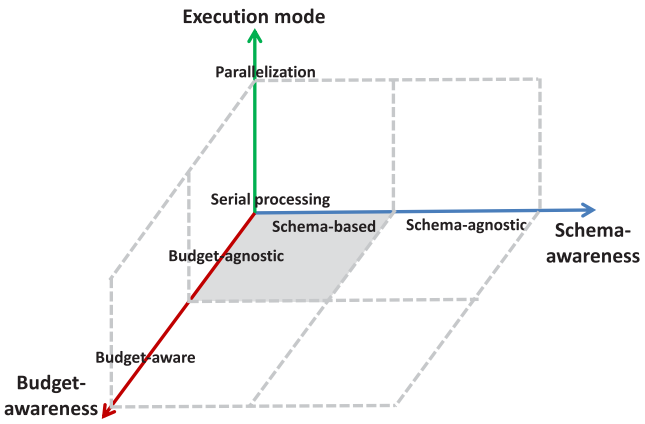

Fig. 1. The solution space of the end-to-end ER pipelines that can be constructed by JedAI.

restricted to experts, and even in that case, their capabilities and scope are rather limited.

In this paper, we present the *Java gEneric DAta Integration* (JedAI) system, an open-source ER system [7] that goes beyond the state-of-the-art in the field by covering a broad range of the main techniques in the literature and by supporting a large variety of use cases. In fact, JedAI can create any end-to-end pipeline that is defined by the following three dimensions:

1. **Schema-awareness.** JedAI supports both *schema-based* and *schema-agnostic* pipelines. The former rely on similarity join techniques, which efficiently detect near duplicates based on the noise-free, distinctive values of a specific attribute name. In contrast, the schema-agnostic workflows extract overlapping blocks from all attribute values and refine them through generic, efficient techniques that disregard any schema knowledge.
2. **Budget-awareness.** JedAI supports both *budget-agnostic* and *budget-aware* pipelines. The former are executed as a batch process that produces results upon its completion, whereas the latter operate in a pay-as-you-go manner that produces results progressively — their goal is to optimize performance within a specific budget of temporal or computational resources.
3. **Execution mode.** JedAI supports both the serialized execution of an end-to-end pipeline and its massive parallelization through Apache Spark [8].

Essentially, each pipeline category involves methods of two fundamentally different types. By allowing the methods of each category to be combined with those of all other dimensions, JedAI introduces the three-dimensional ER, which covers the entire solution space that is formed by the three axes in Fig. 1. This is a unique feature, given that all other tools merely cover the small, two-dimensional part of the solution space that is highlighted in gray.

Another unique feature of JedAI is its **generality**. JedAI supports both *Clean-Clean ER*, which resolves two individually duplicate-free, but overlapping data sources, and *Dirty ER*, which receives as input a single data source that contains duplicates in itself. This goes beyond top tools like Magellan [3], which exclusively supports Clean-Clean ER. Most importantly, JedAI applies seamlessly to data of any structuredness, supporting input formats that range from structured entities to semi-structured and un-structured ones (i.e., described by free text). As a result, its pipelines apply to any domain, as long as its entity profiles are described by textual values, regardless of the level of noise in attribute values and names. The only requirement is that the matching entities share parts of their attribute values. This is demonstrated by the experiments in Section 9, which involve a wide range of large datasets from various domains (e-commerce, bibliographic data, census data etc.).

An additional advantage of JedAI is its **high usability**. JedAI conveys non-learning methods that require minimal human intervention, as neither domain knowledge nor training sets are needed. Users are only required to select the methods that will form an end-to-end workflow. Optionally, the internal parameters of each method can be fine-tuned for optimal performance. In case of no relevant experience, the default configurations can be used, as they have been experimentally verified to consistently achieve high performance across various, diverse datasets [9,10]. In this way, JedAI allows non-experts to create complex pipelines of high performance with minimal human intervention, almost in a hands-off manner. This is made possible through an intuitive user interface that provides hints for building end-to-end solutions, while facilitating the observation of the input data as well as the intermediate and the final results. Notably, the large variety of resulting pipelines can be easily benchmarked through the GUI with respect to both effectiveness and time efficiency. This facilitates to identify the best performing baseline and to assess the impact of a particular method, workflow step or configuration parameter on the overall performance.

In summary, this work makes the following contributions:

- We analytically describe all important aspects of JedAI, delving into the types of solutions it creates and the corresponding end-to-end workflows. We explain the role of every component in its modular architecture, outlining the functionality of each method it includes. Thus, we facilitate not only the use of JedAI, but also its extension with more methods and modules.
- We perform an extensive experimental evaluation that involves 10 real-world and 7 synthetic datasets, whose sizes range from few thousand to few million entities. We evaluate the relative performance of all types of end-to-end pipelines created by JedAI (batch, progressive, schema-based, schema-agnostic, serialized and parallel ones), providing useful insights into their pros and cons.
- We compare JedAI with the state-of-the-art, qualitatively and quantitatively, highlighting the limitations of existing tools and explaining how we go beyond them.

The rest of the paper is structured as follows: Section 2 provides background knowledge, while Section 3 presents JedAI's modular architecture. The back-end is analytically described in Section 4, the front-end in Section 5 and the data model in Section 6. Section 7 presents the massively parallel operation of JedAI, Section 8 discusses applications employing JedAI, whereas Section 9 is devoted to the thorough experimental analysis. The paper concludes with a qualitative comparison with the state-of-the-art in Section 10 and a summary of the key points in Section 11.

2. Problem definition

We call the representation of a real-world object an *entity profile*, or *entity* for simplicity. More formally, an entity profile consists of a unique identifier and a set of textual *name–value* pairs. This simple model is versatile enough to accommodate both structured or semi-structured data. We say that two entities e_i and e_j are *matching* or *duplicates*, $e_i \approx e_j$, if they refer to the same real-world object.

In this context, *Entity Resolution* (ER) is the task of identifying the matching entities within a given set of entity profiles, $\mathcal{E}$.

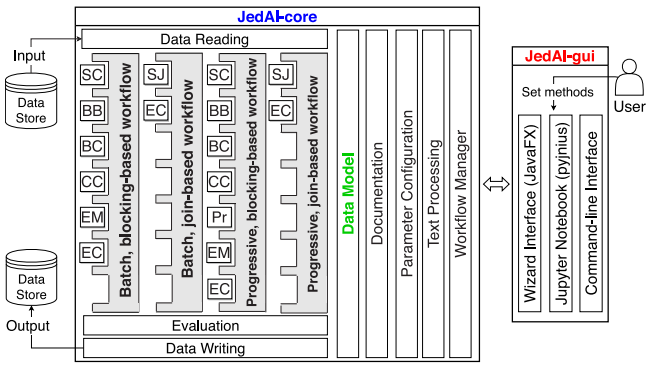

Fig. 2. JedAI's model-view-controller architecture.

When the entities come from two different data sources (i.e., $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$) and each data source is individually duplicate-free, we have a *Clean-Clean* ER problem. When the entities comes from the same data source, which contains duplicates, we have a *Dirty* ER problem.

We call *matching function* the binary function μ that takes as input two entities and determines the likelihood that they are duplicates: $\mu : \mathcal{E} \times \mathcal{E} \rightarrow (0, 1)$. Usually, matching functions require the computation of similarity measures, which are prohibitively expensive to apply on all possible pairs of entities|the complexity of this naïve approach is $\mathcal{O}(|\mathcal{E}|^2)$. The goal of *blocking* methods is to alleviate this complexity by indexing similar entities into *blocks*, so as to restrict the actual comparisons to entities co-occurring in at least one block. The indexing functions employed for blocking are called *blocking functions*. A blocking function $\beta(e_i) \rightarrow \{k_1, \ldots, k_n\}$ takes as input an entity and returns one or more blocking keys, which are used to place the entity into one or more buckets (i.e., the blocks).

After applying the matching function to all pairs in a set of blocks B, a clustering algorithm leverages the results of the matching function to produce the final outcome of ER. This consists of a set of *equivalence clusters*, such that every cluster corresponds to a distinct real-world object and contains all entities that describe it. Note that for Clean-Clean ER, all equivalence clusters have a cardinality up to two.

Let $c_{i,j}$ stand for an individual comparison between entities e_i and e_j, $\mathcal{C}_\mathcal{B}$ for the set of pairwise comparisons in the set of blocks B, $\mathcal{D}$ for the set of matching pairs after the clustering phase, and $\mathcal{M}$ for the real set of matching entities (i.e., ground-truth). To assess the quality of a set of blocks B, we employ the blocking recall, which is called *Pairs Completeness* and is defined as $PC = |\mathcal{C}_\mathcal{B} \cap \mathcal{M}|/|\mathcal{M}|$, and the blocking precision, which is called *Pairs Quality* and is defined as $PQ = |\mathcal{C}_\mathcal{B} \cap \mathcal{M}|/|\mathcal{C}_\mathcal{B}|$. To assess the quality of the overall ER process, we employ *recall* and *precision*, which are respectively defined as $Re = |\mathcal{D} \cap \mathcal{M}|/|\mathcal{M}|$ and $Pr = |\mathcal{D} \cap \mathcal{M}|/|D|$. We also consider their harmonic mean, *F-Measure* ($F1$). Time efficiency is measured through the *running time* (RT) that intervenes between receiving the input entities and producing the equivalence clusters.

3. JedAI overview

JedAI aims to address the following goals:

- **(G1)** *Broad data coverage.* JedAI should apply seamlessly to most types of structured and semi-structured data.
- **(G2)** *Broad literature coverage.* JedAI should serve as a library of the main, established techniques in the literature.
- **(G3)** *Broad scenario coverage.* JedAI should support both academic and commercial applications.
- **(G4)** *High usability.* JedAI should accommodate a broad user base that includes both lay and expert users. The former should be able to build complex, high performing end-to-end pipelines for the data at hand without necessarily knowing all details about the functionality of their methods. The latter should be able to intervene in all aspects of JedAI's functionality so as to tailor it to their special needs.
- **(G5)** *Extensibility.* JedAI should facilitate its enrichment with new techniques or even workflow steps by power users.
- **(G6)** *High time efficiency.* JedAI should process large datasets quickly, not only in commodity, stand-alone systems, but also in powerful computer clusters.

Goal G1 is accomplished through JedAI's flat entity model, which consists of a string-valued entity id (e.g., URI) and a set of textual name–value pairs. This simple model is capable of accommodating the main structured and semi-structured data formats, while supporting noisy attribute names or values, tag-style values (which are not associated with any attribute name) and entity links, where the URI of an entity is given as an attribute value to the associated entity. See Section 6 for details.

To meet G2, JedAI comprises numerous methods that support four different end-to-end ER workflows (cf. Section 4).

For G3, JedAI's code is released under Apache License V2.0, which supports both academic and commercial applications. The former are further facilitated through JedAI's benchmarking functionality; its intuitive GUI allows every user to easily evaluate the relative performance of a large variety of end-to-end pipelines, in case a ground-truth is available (as is common in academic applications). To additionally support commercial applications, any workflow built by JedAI can operate independently of a ground-truth, producing its own detected matches.

To address G4, JedAI equips novice users with a wizard-like GUI, with documentation and with default parameters for every implemented method. In case a ground-truth is available, they can also use two ways of automatic parameter fine-tuning (see Section 4.5). For power users, JedAI offers manual configuration for each method as well as a modular architecture, where every workflow step corresponds to a separate component that implements a simple and clear interface. Every new class (algorithm) implementing a particular interface can be seamlessly integrated into the corresponding component, thus facilitating extensibility (G5), too.

Finally, goal G6 is met for stand-alone systems through GNU Trove [11], which provides high performance data structures that operate on primitive data types instead of objects, restricting their time and space complexity to a large extent [12]. For cluster systems, JedAI supports massive parallelization of all methods and workflows through Apache Spark. This is actually accomplished through the same GUI as the serialized execution.

The above six objectives are accomplished through JedAI's model-view-controller architecture, which is depicted in Fig. 2. JedAI-gui provides the interfaces for user interaction (view), JedAI-core implements the plethora of methods and workflows (controller), and the Data Model component provides the data structures that lie at its core (model). We elaborate on these three parts in the next three sections.

4. Back-end: JedAI-core

This component implements four different end-to-end ER workflows that are formed by two of JedAI's dimensions: budget- and schema-awareness (note that the execution mode does not alter the form of the end-to-end workflows — only the way they are carried out). For each workflow, we briefly describe the role of each step and the functionality of the available methods so as to facilitate their understanding and use by researchers and

practitioners. Typically, any method in a workflow step can be combined with any method of the same or the other steps. Thus, the more steps a workflow involves, the higher is the number of valid combinations, which raises up to several millions for the largest workflows. This is a unique feature among all ER systems.

4.1. Budget- & schema-agnostic workflow

Fig. 3 depicts this end-to-end pipeline along with the available methods per workflow step. All methods are inherently crafted for highly noisy and heterogeneous data, despite their learning-free functionality. They rely on a schema-agnostic functionality that leverages all attribute values in each entity rather than employing a particular set of attributes. Thus, they are resilient to errors in attribute values. Excluding the input and output steps, which are described in Section 6, the processing steps are the following:

(1) Schema Clustering (SC). This is an optional step, suitable for highly heterogeneous datasets with a schema comprising a large diversity of attribute names. In these settings, it significantly improves the overall precision at a limited cost on recall by grouping together attributes that are *syntactically* similar, but are not necessarily *semantically* equivalent [12,13]. Attribute Name Clustering groups together attributes with similar names, Attribute Value Clustering does the same for attributes with similar values, and Holistic Attribute Clustering is a hybrid method that considers both attribute values and names.

All methods can be combined with any similarity measure and representation model from the Text Processing component (see Section 4.5). They produce a set of attribute clusters, which lay the ground for improving the next steps in various ways: Block Building leverages them to break large blocks into smaller ones, without missing duplicates [12], while Comparison Cleaning extracts the entropy per blocking key for a-priori weighting candidate matches [13].

(2) Block Building (BB). This step clusters similar entities into blocks so as to drastically reduce the candidate match space, cutting down on the overall ER running time. It includes most of the state-of-the-art blocking methods [14] using their *schema-agnostic* adaptation [9], which extracts multiple blocking keys from each entity. In this way, every entity participates into several blocks, reducing the likelihood of missed matches, i.e., duplicates having no block in common. In other words, high recall is achieved by producing *overlapping* blocks with high levels of *redundancy*. This comes, however, at the cost of low precision, due to the large number of unnecessary comparisons [10] - the *redundant* ones, which are repeated across different blocks, and the *superfluous* ones, which involve non-matching entities.

The core approach is Token Blocking (**TB**) [15], which uses as blocking keys every token in any attribute value. It is the only parameter-free method in the literature, but is inappropriate for sparse entity profiles with character-level errors.

To cover such cases, Suffix Arrays (SA) [16] extends TB by converting its blocking keys into their suffixes that consist of at least l_{min} characters. Then, it considers only the suffixes appearing in at most b_{max} entities, i.e., maximum block size. Extended Suffix Arrays [9,14] alters SA by converting TB's blocking keys into all substrings (not just suffixes) with more than l_{min} characters that occur in less than b_{max} entities.

A similar approach, independent of frequency thresholds, is Q-Grams Blocking [14,17], which transforms every TB blocking key into all substrings of q characters, i.e., q-grams. Extended Q-Grams [9,14] improves Q-Grams by transforming every TB blocking key into combinations of N q-grams.

All these *hash-based* methods create a separate block for every distinct key such that two matches co-occur in a block if they share at least one key. Duplicates with all their keys differing in at least one character are not placed in any common block, thus being undetectable. To overcome this issue, other methods rely on the *similarity* of keys.

The main similarity-based method is Sorted Neighborhood (**SN**) [18], which sorts TB's keys alphabetically and orders the corresponding entities accordingly; then, it slides a window of fixed size w over the sorted list of entities. In every iteration, the last entity in the current window is compared with all other entities in the same window. Extended Sorted Neighborhood [9, 14] improves SN by sliding the window over the sorted list of blocking keys, rather than the list of entities. This means that each block combines w TB blocks.

Finally, LSH MinHash [19] and LSH Superbit Blocking [20] create blocks with entities whose sets of keys exceed a certain threshold on Jaccard or cosine similarity, respectively.

Note that any combination of the above methods is possible. Usually, this is necessary for highly noisy datasets, e.g., those including both character- and token-level errors.

(3) Block Cleaning (BC). This is an *optional* step that cleans the original blocks from those dominated by the redundant and the superfluous comparisons. Removing these comparisons improves precision at a minor cost in recall [10].

The core assumption in BC is that the larger a block is, the less likely it is to contain unique duplicates, i.e., matches co-occurring in no other block (e.g., a TB block corresponding to a stop word). In this context, Size-based Block Purging [21] discards all blocks exceeding a certain number of entities, Cardinality-based Block Purging [15] discards all blocks exceeding a certain number of comparisons, Block Filtering [22] retains every entity in a subset (r%) of its smallest blocks, and Block Clustering [23] ensures that all blocks remain within a user-specified range of sizes.

These methods are complementary and can be combined for higher performance gains. The larger the set of input blocks is, the more BC methods should be applied to it.

(4) Comparison Cleaning (CC). This *optional* step also targets redundant and superfluous comparisons, but operates at the level of individual comparisons, achieving higher accuracy than BC at the cost of a higher time complexity. It includes primarily Meta-blocking techniques [10], of which only one can be added in an end-to-end pipeline.

The simplest approach is Comparison Propagation [24], which eliminates all redundant comparisons from a set of overlapping blocks. Instead of hashing all executed comparisons in memory, an approach that does not scale to large datasets, it performs a pairwise comparison $c_{i,j}$ in block b_k only if k is the least common block index of e_i and e_j.

All other methods of this step extend Comparison Propagation so that it discards superfluous comparisons, as well. To this end, they rely on block co-occurrence patterns, as they are captured by the *Meta-blocking weighting schemes*. These associate every non-redundant comparison $c_{i,j}$ with a normalized score that depends on the blocks the entities e_i and e_j share: the more blocks they have in common and the smaller these blocks are, the higher is the overall score.

Based on these schemes, Weighted Edge Pruning [25] discards all comparisons with a weight lower than the average one across all distinct comparisons in the input blocks. Cardinality Edge Pruning [25] retains the overall top-K weighted comparisons. Cardinality Node Pruning (CNP) [25] keeps the top-k weighted comparisons per entity. Reciprocal CNP [22] retains comparisons that are among the top-k weighted ones for both involved entities. Weighted Node Pruning (WNP) [25] estimates the average comparison weight for every entity and retains only those comparisons that exceed it. Reciprocal WNP [22] keeps comparisons that exceed the average weight for both involved entities. BLAST [13]

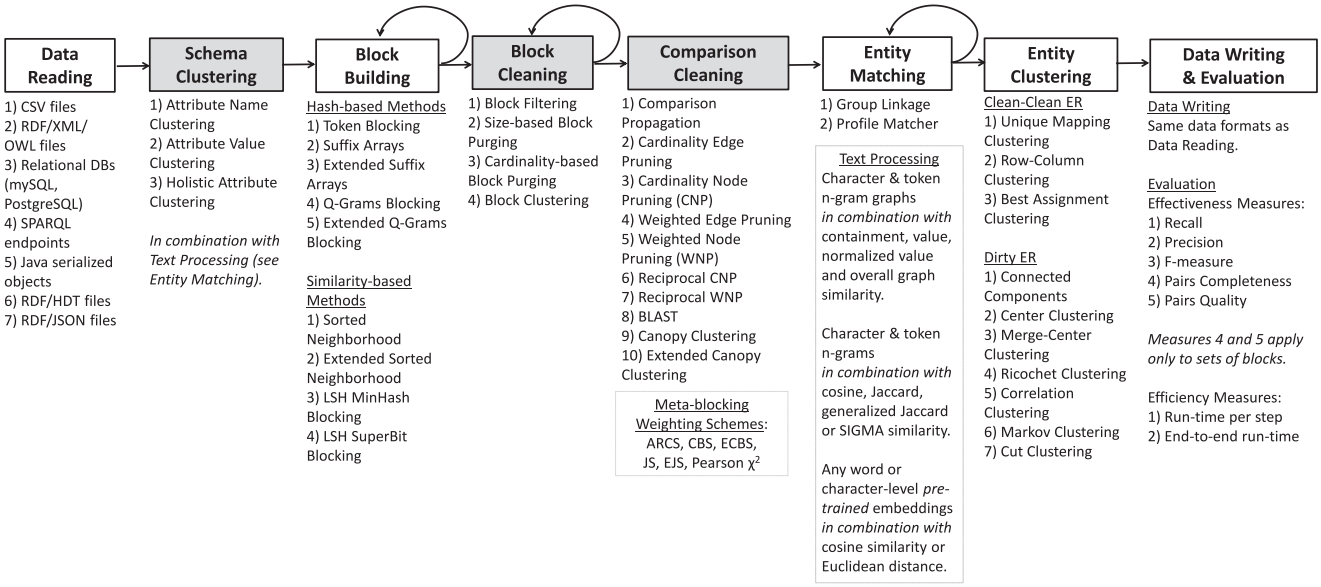

Fig. 3. JedAI's budget- & schema-agnostic end-to-end workflow along with the available methods per step. Self-loops indicate steps that can be repeated, whereas gray rectangles designate optional steps.

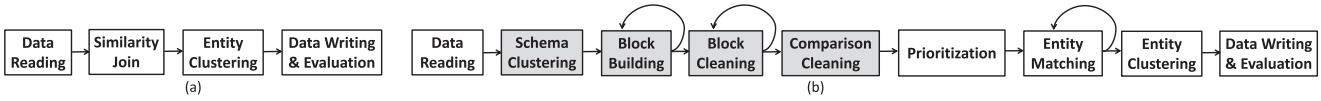

Fig. 4. (a) The schema-based workflow, regardless of budget-awareness; (b) the budget-aware, schema-agnostic one.

retains those weighted higher than the average maximum weight of the two involved entities.

Note that Comparison Cleaning also includes the schema-agnostic adaptation of Canopy Clustering [26], which iteratively selects a random entity from the input blocks and creates a new block that contains all co-occurring entities with a *comparison weight* higher than t_{in}; all entities with a comparison weight higher than t_{ex} ($> t_{in}$) are not placed in any other block. Extended Canopy Clustering [9,14] replaces the weight thresholds with cardinality ones: each new block contains the n_{in} co-occurring entities with the highest comparison weights, while the n_{ex} ($< n_{in}$) most similar entities are excluded from all other blocks.

(5) Entity Matching (EM). This step involves schema-agnostic methods for assessing the value similarity of all entity pairs in the input blocks. Profile Matcher aggregates all attribute values in each entity into a representation model and compares it with the models of the other entities according to a specific similarity measure. Group Linkage [27] matches a pair of entities based on bipartite graph matching: every value from the one entity is linked with its most similar value from the other entity; if the similarity of these links is high enough and there is a large fraction of such links, the two entities are considered duplicates. Both methods can be combined with any similarity measure and representation model from the Text Processing component (see Section 4.5). It is also possible to combine different configurations of these methods into a single workflow, leveraging evidence from multiple representations and similarity measures. In all cases, the resulting similarity scores are normalized in [0, 1].

(6) Entity Clustering (EC). This step partitions the comparisons executed by EM into equivalence clusters. Its functionality depends on the type of the ER task at hand.

For Clean-Clean ER, Unique Mapping Clustering [28] is typically applied. It sorts all pairwise comparisons in decreasing similarity score and iteratively considers the top one as a match, if its score exceeds a predetermined threshold and none of the involved profiles has already been matched. Row-Column Clustering implements an efficient approximation of the Hungarian Algorithm [29], while Best Assignment Clustering implements an efficient, heuristic solution to the assignment problem in unbalanced bipartite graphs [30].

For Dirty ER, the simplest approach is Connected Components [31,32], which sets a cut-off threshold t and considers as matches all comparisons with a similarity score higher than t; then, it estimates the transitive closure of the matches. For higher robustness to noise, more advanced algorithms build clusters around selected entities that operate as centers. Center Clustering [33] defines as centers the nodes with the highest average similarity score, while Merge-Center Clustering [31] unites clusters with centers similar to the same node. Ricochet Clustering [34] defines as centers the entities with the highest number of comparisons and iteratively re-assigns every entity to its closer cluster center, similar to K-Means. Other techniques amplify the strength of *intra-links*, i.e., the similarity scores inside each equivalence cluster, while abating the strength of *inter-cluster links*, i.e., the similarity scores across different equivalence clusters. This approach is treated as an optimization problem by Correlation Clustering [35], whereas Markov Clustering [36] relies on random walks and Cut clustering [37] on the minimum cuts of maximum flow paths.

4.2. Budget-agnostic, schema-based workflow

This type of workflows leverages domain knowledge to achieve both high effectiveness and high efficiency. This is usually the case in datasets where a single attribute contains values that are distinctive enough to identify matching entities. As an example, consider the title attribute in bibliographical data. In such cases, the user needs to define a *matching rule* that consists of two parameters: the distinctive attribute, and a similarity threshold,

above which two values are considered to indicate duplicate entities.

The steps of this end-to-end budget-agnostic, schema-based workflow appear in Fig. 4(a). After data reading, JedAI allows users to detect the most reliable attribute in terms of noise and distinctiveness through the data exploration functionality (see Section 5). A similarity join algorithm is then used to accelerate the detection of pairs of entities that satisfy the user-specified matching rule, while a clustering algorithm leverages the resulting similarity scores to identify implicit matches or remove wrong ones.

To implement the *Similarity Join* (**SJ**) step, JedAI conveys a library of the state-of-the-art techniques. They are listed in Fig. 5 and can be distinguished in two broad categories according to the similarity measures they support.

The token-based methods are crafted for the Overlap, Jaccard, Cosine and Dice similarity measures [38,39]. To compute them, these methods transform every textual value into the set of its tokens. AllPairs [40] sorts the tokens of every attribute value in increasing order of frequency across all values. Then, it forms the prefix of each value by selecting the n first tokens, i.e., the n rarest ones. Subsequently, *Prefix Filtering* demands that two values exceed the user-specified similarity threshold if their prefixes share at least one token. The size of the prefix depends on this threshold and the selected similarity measure. In general, the higher the similarity threshold, the shorter the prefix and the less candidate matches are produced. PPJoin [41] extends Prefix Filtering with *Positional Filtering*, which estimates a tighter upper bound for the overlap between the two sets of tokens, based on the positions where the common tokens in the prefix occur. SilkMoth goes beyond these methods by enabling fuzzy joins, i.e., allowing slight variations in the matching tokens.

The character-based methods are crafted for Edit Distance, which essentially estimates the minimum number of edit operations (i.e., insertions, deletions and substitutions) that are required to transform one attribute value to another [38]. For short textual values, FastSS [42] provides the most efficient filtering [38]. Every value is associated with the set of substrings that are produced after deleting a certain number of characters, and every other value that shares one or more substrings is considered a candidate match. PassJoin [43] partitions a value into a set of non-overlapping character q-grams and, based on the pigeon-hole principle, it considers as candidate matches the values that share at least one of these q-grams. The same principle lies at the core of PartEnum [44], which is however crafted for the Hamming Distance, i.e., the minimum number of substitutions required to change one value to the other. Ed-Join [45] adapts Prefix Filtering to Edit Distance, similar to the character-based AllPairs, and optimizes it by removing unnecessary q-grams from the prefix and by adding Position Filtering.

4.3. Budget-aware, schema-agnostic workflow

This workflow is suitable for applications with limited computational or time resources, which can only be addressed in a *pay-as-you-go* way that provides the best possible *partial solution* in the context of the available resources. To this end, it applies the workflow in Fig. 4(b). Even though it seems identical to the budget- and schema-agnostic workflow in Fig. 3, there are several key differences: (i) Data Reading also receives as input the user-specified *budget* in terms of the maximum running time or the maximum number of executed comparisons. (ii) Block Building is now an optional step, as some progressive methods can be applied directly to the input entities. (iii) Entity Matching executes one comparison at a time. (iv) Evaluation primarily focuses on the rate of detected duplicates per comparison, i.e., the evolution of recall as more comparisons are executed. The resulting diagram is used for estimating the area under curve, which is analogous to the effectiveness of the progressive methods. (v) A new step, called *Prioritization* (**Pr**), is applied before Entity Matching to schedule the processing of entities, comparisons or blocks.

Similarity Join Methods		Prioritization Methods
Token-based	Character-based	1) Local Progressive Sorted Neighborhood
1) AllPairs	1) FastSS	2) Global Progressive Sorted Neighborhood
2) PPJoin	2) PassJoin	3) Progressive Block Scheduling
3) SilkMoth	3) PartEnum	4) Progressive Entity Scheduling
	4) EdJoin	5) Progressive Global Top Comparisons
	5) AllPairs	6) Progressive Local Top Comparisons

Fig. 5. The available methods for SJ and Pr.

In more detail, Prioritization consists of two phases. First, the *initialization phase* associates all entities, comparisons or blocks with a weight that is proportional to the likelihood that they involve duplicates. Then, it orders accordingly part of the pairwise comparisons in decreasing weight. Second, the *emission phase* emits iteratively the next pair of entities to be compared by Entity Matching, stopping when the available budget runs out. If all prioritized comparisons are emitted before the end of the budget, the initialization phase is repeated to schedule the processing of the next ones.

Prioritization incorporates the techniques presented in Fig. 5. *Local Schema-agnostic Progressive SN* [46] applies directly to the input entities, sorting them according to schema-agnostic Sorted Neighborhood. Then, it slides a window w=1 along the sorted list of entities to compare all profiles in consecutive positions. The window size is iteratively incremented ($w = 2, 3, \ldots$) until reaching the user-defined budget. In each window size, the initialization phase orders non-redundant comparisons in decreasing frequency. This approach is extended by *Global Schema-agnostic Progressive SN* [46] so that the initialization phase operates for a predetermined range of windows.

The rest of the methods operate on blocks, leveraging Meta-blocking weighting schemes to define an ordering of comparisons. *Progressive Block Scheduling* [46] orders the input blocks in ascending number of comparisons and then prioritizes all comparisons in the current block in decreasing matching likelihood. *Progressive Entity Scheduling* [46] orders entities in decreasing average comparison weight and then prioritizes all comparisons involving the current entity by ordering them in decreasing matching likelihood. *Progressive Global Top Comparisons* simply orders all comparisons in the input blocks in descending matching likelihood. *Progressive Local Top Comparisons* extracts the k top-weighted comparisons per entity from the input blocks and orders all of them in decreasing matching likelihood.

4.4. Budget-aware, schema-based workflow

This pipeline implements the same workflow as its budget-agnostic counterpart, which is depicted in Fig. 4(a). The only difference is that it implements a single method, namely the *Top-k Similarity Join* [47]. This algorithm is crafted for token-based joins and applies seamlessly to both Dirty and Clean-Clean ER. During the initialization phase, it constructs an index similar to that of Prefix Filtering. During the emission phase, it iteratively emits pairs of candidate matches in non-increasing order of estimated similarity (usually Jaccard). This is carried out in two ways: globally, by considering all comparisons across the entire dataset, or locally, by considering the top-k comparisons per entity.

4.5. Auxiliary components

We now describe three components that do not implement any Entity Resolution algorithm, but play a crucial role in all end-to-end workflows supported by JedAI.

Documentation. This component is crucial for the usability of JedAI. It is applied through the `IDocumentation` interface, which conveys a series of abstract functions, with each one returning a textual description about a core characteristic of an algorithm: (i) its name, (ii) a short explanation of its functionality, (iii) the names of its internal parameters, (iv) a brief description of each parameter, and (v) the configuration of the current instance of the algorithm, i.e., the specified value for every internal parameter. Another method facilitates the manual configuration of any implemented technique by providing all necessary information in `JSON` format: the type (i.e., Java class) of each parameter, its default value and, in case of numeric parameters, the range of values that are typically used by experts in practice. This range is determined through a minimum and a maximum value as well as a step for automatically searching the best value in this interval. All techniques in JedAI implement this interface, laying the ground for the how-to guide that is offered by its various front-ends (cf. Section 5).

Parameter configuration. A major task when applying end-to-end ER pipelines is to configure properly the parameters of their methods. This is non-trivial, due to the strong dependency between the successive workflow steps. Given that the output of each step is the input to the next one, a step malfunctioning because of poor parameterization has a devastating effect on the overall performance of the end-to-end workflow. To address this issue, this component offers four solutions that apply to every method:

(1) *Default configuration.* Every parameter is a-priori set to a value that has been verified to achieve high performance through an extensive experimental analysis over a series of established benchmark datasets [9,10]. This parameterization is the default choice in JedAI and requires no input by the user, thus being ideal for lay users.

(2) *Manual configuration.* Expert users are able to leverage their deep ER knowledge by determining the value of any parameter with the help of the Documentation component.

(3) *Grid search.* In case a ground-truth is available for the data at hand, this approach automatically detects the best parameterization in a brute-force way. For categorical parameters (e.g., the weighting scheme in Meta-blocking), it considers all possible values, while for numeric parameters, it considers all values specified by the `Documentation` component. The configuration corresponding to the maximum overall effectiveness is selected as the best one.

(4) *Random search.* To automatically fine-tune a method when the ground-truth is available, this approach draws random values from the domain of every parameter and evaluates their effectiveness. For numerical parameters, it selects arbitrary values in the interval [minimum, maximum] that is specified by the `Documentation` component, while for categorical parameters, it simply picks among the available values. The configuration maximizing effectiveness after a limited number of iterations is selected as the best one.

In practice, JedAI associates every configuration parameter in every method with two classes: one implementing grid search and one implementing random search. This allows for seamlessly applying both approaches to all methods in an end-to-end pipeline in two different ways:

(i) During *holistic configuration*, the parameters of all methods in the selected end-to-end workflow are simultaneously optimized. In every iteration, a new value is assigned to at least one internal parameter and the iteration achieving the highest F-Measure is selected as optimal.

(ii) In *step-by-step configuration*, the performance of each workflow step is optimized with respect to F-Measure independently of those following it.

Note that there is a trade-off between the efficiency and the effectiveness of these two approaches [48]: step-by-step configuration is typically much faster, as it gradually minimizes the computational cost of every workflow step. In contrast, holistic configuration might involve a workflow step with high computational cost, as long as the overall F-Measure is high. However, step-by-step configuration is prone to confining itself in local maxima for each workflow step, with the overall effectiveness lying very far from the optimal one. Instead, holistic configuration is crafted for identifying the global maximum. Combining it with grid search, though, might lead to an exponential computational cost.

Text processing. As explained above, JedAI is crafted for integrating datasets that are dominated by textual values. This component provides the techniques that are necessary for identifying similarities between sets of textual values that represent individual attributes (in Schema Clustering) or individual entities (in Entity Matching).

In more detail, this component comprises a series of established, state-of-the-art representation models, which transform a set of textual values in a format that is suitable for applying various similarity measures. The cornerstone approach is the *vector model* [49], which converts every value into a set of character or token n-grams. For character n-grams, $n \in \{2, 3, 4\}$, while for token n-grams, $n \in \{1, 2, 3\}$. The resulting vector representations are weighted in two ways [49]: TF associates every n-gram with its frequency in the current set of values, while TF-IDF multiplies the frequency of every n-gram with its inverse document frequency $IDF = \log |N|/|N_i|$, where $|N|$ stands for the total number of sets of values and $|N_i|$ for the number of sets that contain the current n-gram.

To assess the similarity of two vector representations, several established measures are available [49]: the cosine similarity, the Jaccard similarity, which assumes binary weights in the vectors, variations of the Jaccard similarity that support TF and TF-IDF weights, the SIGMA similarity [28], and the ARCS similarity $ARCS = \sum_{N_i \in N_C} 1/log|N_i|$, where $|N_C|$ stands for the common n-grams between two vectors.

However, the vector model disregards the order of n-grams in a textual value; e.g., "Bob sues Jim" and "Jim sues Bob" have identical token unigram representations, despite their significantly different meaning. To address this issue, *n-gram graphs* [50] enrich the vector model with contextual information: every n-gram is connected with an edge with every other n-gram that co-occurs in the same textual value within a window of size n; the weight of each edge is inversely proportional to the distance of the neighboring n-grams. This approach supports both character and token n-grams, with $n \in \{2, 3, 4\}$ and $n \in \{1, 2, 3\}$, respectively, as in the vector model.

To assess the similarity between the resulting graphs, several measures are supported [50,51]: the containment similarity estimates the proportion of edges that are shared by two graphs, regardless of the associated weights; the value similarity considers the weights of common edges in the graphs such that higher scores correspond to graphs with more similar weights; the normalized value similarity enhances value similarity so that it is insensitive to the relative size of the compared graphs; the overall similarity computes the average of these 3 measures.

JedAI offers another way of enriching the vector model with contextual information: the *pre-trained embeddings* [52–54], which transform every n-gram into a real-valued vector of low

dimensions (e.g., 50 or 200). Unlike *n*-gram graphs, which exclusively consider the internal context in textual values, the embeddings rely on external context, extracting co-occurrence patterns from large textual corpora such as Wikipedia. JedAI actually supports any form of pre-trained word- or character-level embeddings, like GloVe [53], word2vec [54] and fastText [52]. The user only needs to provide the path of the file containing them. To compare the vector models, the cosine similarity or the Euclidean distance are used in this case.

Note that all similarity measure scores are normalized in [0, 1], with higher values corresponding to higher similarity.

5. Front-end: JedAI-gui

JedAI offers the following interfaces for user interaction:

(1) **Web application.** This is the main GUI of JedAI, allowing for constructing any combination of the available methods in the context of the aforementioned four end-to-end ER workflows. To facilitate this process, it displays all explanatory information provided by the Documentation module for each method and internal parameter. It also facilitates the benchmarking of different workflows or configurations through the *Workbench window*, which summarizes the outcomes of all runs and maintains details about the effectiveness, the time efficiency as well as the configuration of every step. Another crucial functionality is *Data Exploration*, which provides users with a comprehensive overview of the input and output data, allowing them to delve into the peculiarities of their data in order to form more fitting workflows. This is particularly useful in the case of the schema-based workflows, where users should select the most appropriate attribute for applying similarity join. Note that this GUI provides a unified access to both serial and parallel processing (using Apache Livy [55]) and can be easily deployed through a Docker image [56].

Part of the screens of this new GUI are depicted in Fig. 6. The left image depicts the screen for selecting among the available methods in each workflow step (Comparison Cleaning in this case), the middle one shows the presentation of the effectiveness and time efficiency of a particular workflow, and the right one illustrates the benchmark screen, which allows for inspecting the performance per workflow step and for comparing the performance of different workflows or configurations.

The code of the Web application is available here: https://github.com/GiorgosMandi/JedAI-WebApp.

(2) **Command-line interface.** This interface implements the basic functionalities of JedAI. First, it asks users to select one of the four end-to-end workflows and then to select one or more methods per workflow step. This can be repeated multiple times in the context of benchmarking different workflows, with a screen summarizing the experimental results so far. It also provides access to the Documentation component, allowing users to retrieve information about individual methods or specific parameters.

(3) **Jyputer notebook.** To integrate JedAI with Python's data analysis ecosystem, we augmented JedAI-core with a Python wrapper based on `pyjnius` [57]. Thus, JedAI can be seamlessly used in a Jupyter Notebook, following the guidelines that are available in its code repository [7].

6. Data flow

We now describe the data structures that are used in JedAI's workflow steps with the aim of achieving high time efficiency and low memory footprint in then serialized execution. JedAI leverages the data structures of GNU Trove, which operate on primitive types, instead of the default ones in `java.util.Collections`, which rely on the wrapper classes; e.g., GNU Trove uses a 4-byte `int`, rather than a 16-byte `java.lang.Integer`. In this way, the memory footprint is minimized across all operations of JedAI.

Input data. The first step in all pipelines is *Data Reading*, which loads from disk into main memory the dataset(s) to be processed along with the ground-truth, if available. The corresponding entities are converted into a flat model that represents them as sets of property-literal and relation-URI pairs. In this way, JedAI supports the main structured data formats (relational databases and CSV) along with the main semi-structured data formats, i.e., SPARQL endpoints and RDF, XML, OWL, HDT and JSON files. Any mixture of those formats is possible in the case of Clean-Clean ER. No additional contextual or domain knowledge (e.g., ontology) is required as part of the input. Data Reading also assigns to every entity a unique `integer` id so that all subsequent steps avoid using its original, textual URI.

Output data. JedAI allows for storing final or intermediate results (for debugging purposes) in any of the supported data formats through the *Data Writing* workflow step.

Block building. This step converts the input entities into an *inverted index* (hash table), which points from `String` blocking keys to `int` entity ids. Every posting list with more than two entity ids is transformed into a block, which consists of a unique `int` id and an array of the corresponding entity ids. The set of all blocks is returned as output.

Block cleaning. This step modifies the contents of the input block, without creating any sizeable data structure.

Comparison cleaning. To weight pairs of entities according to their co-occurrence patterns and detect redundant comparisons, an *Entity Index* [21] is created to associate every entity id with the block ids that contain it. This information is then used to modify the input blocks.

Entity matching. This step converts every input entity into a representation model that facilitates the computation of similarity measures. This model constitutes an n-gram graph, a dense embeddings vector or a sparse n-gram vector (the last one is actually a hash table with `String` n-grams as keys and `int` frequencies as values). Using the resulting model, all pairwise comparisons in the input blocks are executed. The output comprises the set of entity id pairs along with the corresponding similarity scores.

Entity clustering. This step converts the executed comparisons into an undirected *similarity graph*, where the nodes correspond to entities, and the edges connect the compared entities. Every edge is weighted in [0, 1] according to the similarity score of the corresponding entity profiles. A clustering algorithm extracts from the similarity graph the set of equivalence clusters, with each one containing the array of entity ids that are deemed as duplicates.

Similarity join. The structures used in this step depend on the functionality of each technique. Most of them, though, rely on signatures, similar to Blocking. Thus, they employ an Inverted Index that maps every textual signature to the ids of the entities that are represented by it.

Prioritization. If the selected method operates on a set of blocks, this step employs the Entity Index to weight the candidate matches. If the selected method applies directly to the input entity profiles, this step employs the array of entity ids that is formed by schema-agnostic Sorted Neighborhood. In any case, the initialization phase populates a priority queue with a small number of the most promising comparisons, sorted in decreasing matching likelihood. The emission phase pulls the first element from this queue; whenever the queue gets empty, it is repopulated with the next group of promising comparisons.

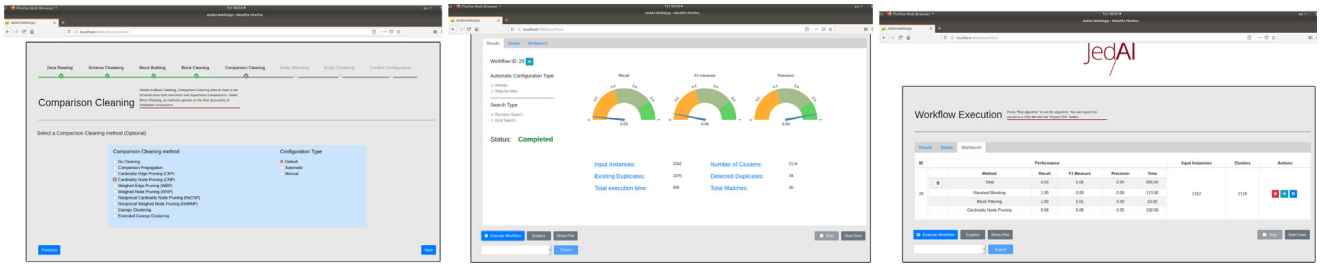

Fig. 6. Screenshots of JedAI's new graphical user interface.

7. Parallel execution

We now describe the methods and the data structures we used for adapting all algorithms implemented by JedAI to the parallel execution on top of Apache Spark. The implementation in Scala is available here: https://github.com/scify/JedAI-Spark.

Budget- and schema-agnostic workflow. Block Building receives as input an RDD of entity profiles and essentially builds an inverted index that points from `String` blocking keys to `int` entity ids. The adaptation to the MapReduce paradigm is straightforward for most methods, like Token Blocking: the Map phase extracts the blocking keys from each entity, while the Reduce phase aggregates all entity ids that correspond to the same blocking key. Each key with more than two associated entities creates a separate block. For Sorted Neighborhood, we use the MapReduce algorithm proposed in [58]. Sorting keys are extracted in parallel from the profiles and then are sorted. The sorted profiles are split into overlapping partitions, whose overlap is equal to the selected window size. Blocks are then generated from each partition using a sliding window, as in serialized SN. For LSH, we employ the algorithm proposed in [59] to extract MinHash signatures from each profile in parallel.

The Block Cleaning methods are also easy to adapt to MapReduce, because they simply modify the blocks that are already distributed on an RDD after Block Building. In fact, they remove or shrink the largest blocks with the aim of reducing the unnecessary comparisons they involve [22].

For Comparison Cleaning, we adopt the approach described in [60]. First, the Entity Index I_E is created in the form of an RDD, mapping each profile id to the ids of the blocks that contain it. Then, an inverted index I_B is built, associating each block id with the ids of the profiles it contains. I_B is broadcasted to all worker nodes so that they can build a profile's neighborhood locally in combination with I_E. For each block id contained in I_E, it is possible to obtain all the entity (neighbor) ids contained in that block from I_B. The pruning is finally performed inside each profile neighborhood. The main advantage of this approach is that the blocking graph is not materialized in its entirety. Only a portion is materialized by each worker, thus restricting the memory consumption to manageable levels.

Entity Matching receives two RDDs as input: one containing the input entity profiles and another one containing the candidate matches that have been identified by the previous steps. A core requirement is to re-distribute the entity profiles, such that the ones needed for carrying out each pairwise comparison coexist in the same nodes. This is implemented by performing two left-outer joins. This process places all entity profiles in the right nodes, allowing for efficiently applying Profile Matching or Group Linkage.

In more detail, we reduce the set of candidate matches into an RDD of key–value pairs. The key corresponds to id of an entity, while the value consists of an array of all entity ids that are likely to match with the key entity. Using a left outer join, the ids of the keys are replaced with the respective profiles. We repeat the same procedure for all value ids, constructing an RDD that contains entity profiles as keys and arrays of profiles to compare as values.

JedAI also offers an alternative implementation that is based on broadcasted variables. First, it forms tuples with entity ids as keys and values comprising arrays of all ids that are potential matches of the key id. The resulting RDD is collected and broadcasted to all executors so that it can be used as an inner variable to Spark actions and transformations in combination with the RDD that contains the original profiles. If the RDD is too large to fit in main memory, it is processed gradually; smaller parts are sequentially broadcasted to avoid exceeding the broadcast size limitation.

Entity clustering receives as input an RDD of matches, i.e., pairs of entity ids along with a weight that represents their similarity score. The corresponding similarity graph is built using the GraphX library[1] of Apache Spark. The same library is used to split the graph into its connected components in parallel. This is required by most clustering methods for Dirty ER, as they typically go on to refine the original clusters [31]. Adapting them to work on top of Apache Spark is straightforward, because they process each connected component independently of the others.

Budget-agnostic, schema-based workflow. To execute similarity joins in a distributed way, we adapted the algorithm in [61] to work with Spark. Starting from an RDD of profiles and a similarity threshold t, each profile is parallelly transformed into a set of signatures (character n-grams or tokens) that are sorted by their entity frequency. Then, a Prefix Index [62] is built. For each entity, we probe the posting lists of the index that contain it, gathering the candidate matches. A series of filters (e.g., length filter, prefix filter) removes those candidates that cannot reach the requested threshold. We optimized this functionality by using the *LeCoBI* condition [21] to avoid emitting duplicate pairs, due to the parallel execution of this step. *LeCoBI* essentially checks whether the current posting list that contains two entities is their first common one, i.e., it corresponds to their least common list id. Only in this case is the pair of candidate matches emitted. Otherwise it is skipped. Finally, the pairs that pass the filters are analytically compared with the selected similarity function. Only those with a similarity greater or equal than the desired threshold t are kept.

Budget-aware workflows. The schema-agnostic workflows of this type rely on the distributed implementation of Sorted Neighborhood for the prioritization algorithms that apply directly to the input entity profiles. The rest of the prioritization algorithms, which operate on a set of blocks, employ the distributed algorithms of Comparison Cleaning. The schema-based workflows rely on the aforementioned distributed implementation of similarity joins. For both workflow types, only the initialization phase is executed in parallel, as the emission phase simply returns the pre-computed next best pair of entities.

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

8. Applications

Given that there is no clear winner among the available ER techniques, extensive experimentation is required to identify the best end-to-end workflow for each ER task at hand. The main goal of JedAI is to facilitate this process, offering a library of the state-of-the-art techniques and a practical GUI for building, testing and visualizing the results of end-to-end ER pipelines. In this context, JedAI is ideal for the *development phase* of ER solutions, simplifying the identification of the best end-to-end pipeline for a particular application.

Regarding the *production phase*, JedAI is currently used in some commercial data integration projects (e.g., see http://www.datariver.it/en/sparker). It is also used in academic applications that involve data integration tasks, like the research projects *OpenAIRE* (https://www.openaire.eu), *Copernicus App Lab* [63] and *ExtremeEarth* (http://earthanalytics.eu). Even though JedAI's codebase is in development for the last 10 years (since [15]), having incorporated several optimizations (e.g., the use of GNU Trove's primitive collections for Java), further optimizations are possible, depending on the industrial use case and the pipeline that is selected during the development phase.

JedAI in practice. Based on our experience with JedAI in practical data integration tasks, we provide high-level guidelines for deciding which optional steps should be included in a schema-agnostic end-to-end workflow among Schema Clustering, Block Cleaning and Comparison Cleaning.

The use of the Schema Clustering depends on the input data. For homogeneous data sources, which involve a limited number of attributes, there is no need to apply Schema Clustering. For heterogeneous data sources, though, it is indispensable for reducing the computational cost of the blocks to a large extent. By grouping together attributes with similar values and/or names, it yields a larger number of smaller blocks than applying a completely schema-agnostic blocking method [13,21]. It also provides useful information for comparison weighting in case Meta-blocking is used in the same pipeline [13].

The use of Block Cleaning depends on the selected Block Building techniques. If the resulting blocks exhibit little variation in their sizes, with most blocks involving few entities, Block Cleaning should be avoided. Such blocks are usually derived from blocking methods that apply size constraints, like (Extended) Suffix Arrays Blocking. Instead, Block Cleaning is indispensable if the resulting blocks exhibit a Zipf distribution, where the frequency of blocks is inversely proportional to their size (i.e., most blocks are small and few are excessively large).

Finally, Comparison Cleaning should be used in all cases. For blocks with low levels of redundancy, such as those produced by drastic Block Cleaning or proactive Block Building, Comparison Propagation is necessary to remove all redundant comparisons at no cost in recall. In most cases, though, the blocks involve high levels of redundancy, requiring one of the Meta-blocking approaches to reduce the computational cost of the pipeline to manageable levels.

In any case, the best method per optional step depends on the data at hand, as there is no clear winner among them.

9. Experimental analysis

The goal of our experimental study is manifold: (i) to evaluate the relative performance of the two types of budget-agnostic (i.e., batch) end-to-end workflows, (ii) to assess the benefits of the budget-aware (i.e., progressive) end-to-end workflows over the corresponding budget-agnostic ones, (iii) to demonstrate the scalability of JedAI's parallelization of all methods and workflows over Apache Spark, and (iv) to quantitatively compare JedAI with state-of-the-art systems.

Experimental setup. All methods and experiments are implemented in Java 8. The code for the experiments is publicly available in JedAI's code repository [7]. The experiments were ran on a server with Intel Xeon E5-4603 v2 (2.2 GHz, 16 physical cores), 128 GB RAM, running Ubuntu 14.04.5 LTS. For all time measurements, we repeated the experiments 10 times and report the mean values. We also report memory requirements per experiment and dataset.

Datasets. The technical characteristics of the 17 datasets we use in our experiments are reported in Tables 1 and 2 for Dirty ER and Clean-Clean ER, respectively. All of them have been widely used in the literature [1,9,10,14,64] and are publicly available through JedAI's code repository [7].

The Dirty ER datasets in Table 1 include two real-world collections: D_{cddb}, which contains entity profiles describing CDs randomly extracted from `freeDB.org`, and D_{cora}, which contains entity profiles with bibliographical information for scientific papers. The rest of the datasets are synthetic, involving census data artificially generated by Febrl [65] through the following procedure: duplicate-free entity profiles were initially formed based on frequency tables for real names (given and surname) and addresses. Then, duplicates were randomly generated based on real error characteristics and modifications (e.g., inserting, deleting or substituting characters or words). Each entity was subject to at most 10 modifications, of which up to 3 modifications for the same attribute value. 40% of the resulting entities are duplicates, with less than 10 matches per entity.

The Clean-Clean ER datasets in Table 2 exclusively contain real-world collections. D_{c1} includes entity profiles that describe restaurants from the Fodor's and Zagat restaurant guides. D_{c2} contains product entities from the online retailers `Abt.com` and `Buy.com`. D_{c3} matches products from Amazon with those from Google Base. D_{c4} involves publication entities from DBLP and ACM Digital Library. D_{c5} matches products from Walmart and Amazon. D_{c6} aligns curated publication entities from DBLP with noisy publication entities from Google Scholar. D_{c7} conveys a collection of movie entities shared by DBPedia and IMDB. D_{c8} matches two versions of the DBPedia Infobox Data Set, which chronologically differ by two years. D_{c8} is actually the only dataset with high schema heterogeneity, as in all other cases, any discrepancies in the schema can be manually fixed.

Budget- and schema-agnostic workflow. This workflow consists of the following methods: Block Building, Block Purging, Block Filtering, CNP, Profile Matcher and Connected Components or Unique Mapping Clustering in case of Dirty and Clean-Clean ER, respectively. The first four methods are exclusively considered with their default configurations, because the pipeline they form consistently exhibits an excellent performance across the diverse datasets we have considered, as shown in Table 4(a): blocking recall (PC) exceeds 90% in most cases, while blocking precision (PQ) is very high, since the number of pairwise comparisons is bounded by the number of input entities. On average, CNP retains at most 11 comparisons per entity. As a result, this workflow has three degrees of freedom, i.e., parameters that need to be fine-tuned: the representation model and the similarity measure used by Profile Matcher for executing the pairwise comparisons as well as the similarity threshold *simTh* of the Entity Clustering method.

Using grid search, we estimated the configuration that maximizes the F-Measure for each dataset to yield the corresponding *best configuration*. The resulting parameters per dataset appear in Table 3(a). We also determined the *default configuration* of this workflow as the parameter settings that achieve the maximum average F-Measure across all datasets. Recall that both schema-agnostic workflows used the default configuration for the methods in the first three workflow steps, i.e., Token Blocking, Block Purging, Block Filtering and CNP. For Entity Matching, we

Table 1
Technical characteristics of the real and synthetic datasets for Dirty ER. $|E|$ stands for the number of entity profiles, NVP for the total number of name–value pairs in the dataset, $|N|$ for the number of distinct attributes, $|\bar{p}|$ for the average profile size (in terms of name–value pairs), $|D(E)|$ for the number of duplicate pairs, and $\|E\|$ for the comparisons executed by the brute-force approach.

	D_{cora}	D_{cddb}	D_{10k}	D_{50k}	D_{100k}	D_{200k}	D_{300k}	D_{1M}	D_{2M}
$\|E\|$	1295	9763	10,000	50,000	100,000	200,000	300,000	1,000,000	2,000,000
NVP	7166	183,072	106,108	530,854	1,061,421	2,123,728	3,184,885	10,617,729	21,238,252
$\|N\|$	12	106	12	12	12	12	12	12	12
$\|\bar{p}\|$	5.53	18.75	10.61	10.62	10.61	10.62	10.62	10.62	10.62
$\|D(E)\|$	17,184	299	8705	43,071	85,497	172,403	257,034	857,538	1,716,102
$\|\|E\|\|$	$8.38 \cdot 10^5$	$4.77 \cdot 10^7$	$5.00 \cdot 10^7$	$1.25 \cdot 10^9$	$5.00 \cdot 10^9$	$2.00 \cdot 10^{10}$	$4.50 \cdot 10^{10}$	$5.00 \cdot 10^{11}$	$2.00 \cdot 10^{12}$
Type	Real	Real	Synthetic	Synthetic	Synthetic	Synthetic	Synthetic	Synthetic	Synthetic

Table 2
Technical characteristics of the real datasets for Clean-Clean ER.

	D_{c1}	D_{c2}	D_{c3}	D_{c4}	D_{c5}	D_{c6}	D_{c7}	D_{c8}
$Dataset_1$	Rest.1	Abt	Amazon	DBLP	Walmart	DBLP	DBPedia	DBPedia 3.0rc
$Dataset_2$	Rest.2	Buy	Google Pr.	ACM	Amazon	Scholar	IMDB	DBPedia 3.4
NVP_1/NVP_2	1130/7519	2568/2308	5302/9110	10,464/9162	14, 143/$1.1 \cdot 10^5$	10, 064/$2 \cdot 10^5$	$1.6 \cdot 10^5/8.2 \cdot 10^5$	$1.69 \cdot 10^7/3.50 \cdot 10^7$
$\|E_1\|/\|E_2\|$	339/2256	1076/1076	1354/3039	2616/2294	2554/22,074	2516/61,353	27,615/23,182	$1.19 \cdot 10^6/2.16 \cdot 10^6$
$\|N_1\|/\|N_2\|$	7/7	3/3	4/4	4/4	6/6	4/4	4/7	30,688/52,489
$\|\bar{p}_1\|/\|\bar{p}_2\|$	3.33/3.33	2.39/2.14	3.92/3.00	3.99/4.00	5.54/5.18	3.23/3.26	5.63/35.20	14.19/16.18
$\|D(E_1 \cap E_2)\|$	89	1076	1104	2224	853	2308	22,863	892,579
$\|\|E_1 \times E_2\|\|$	$7.65 \cdot 10^5$	$1.16 \cdot 10^6$	$4.11 \cdot 10^6$	$6.00 \cdot 10^6$	$5.64 \cdot 10^7$	$1.54 \cdot 10^8$	$6.40 \cdot 10^8$	$2.58 \cdot 10^{12}$

Table 3
Parameter configuration of JedAI's budget-agnostic workflows. C2GG stands for character big graph graphs, while the suffixes TF and TFIDF denote TF and TF-IDF weights, resp., and the prefixes C2G, C3G, T1G and T2G stand for character bigrams, character trigrams, token unigrams and token bigrams, resp.

	D_{c1}	D_{c2}	D_{c3}	D_{c4}	D_{c5}	D_{c6}	D_{c7}	D_{c8}	D_{cora}	D_{cddb}
(a) Best configuration of the schema-agnostic workflow										
Representation Model	C2G TF	C2G TFIDF	T2G TFIDF	T1G TFIDF	T1G TFIDF	C3G TFIDF	T1G TFIDF	T1G TFIDF	T1G TFIDF	C2GG
Sim measure *SimThr*	Cosine 0.90	Cosine 0.30	Cosine 0.05	Sigma 0.55	Cosine 0.60	Sigma 0.45	Cosine 0.10	Sigma 0.45	Gen. Jaccard 0.45	Graph overall 0.75
(b) Configuration of the schema-based workflow										
Attribute *SimThr*	phone no. 0.90	name 0.40	title 0.45	title 0.8	modelno 0.90	title 0.80	title 0.45	name 0.80	title 0.70	track08 0.80

employ character 4-grams with TF-IDF weights with cosine similarity and $simTh = 0.15$ in the case of Clean-Clean ER, while for Dirty ER, we use character 2-gram graphs with graph value similarity and $simTh = 0.65$.

Looking into Table 4(b), we observe that the default configuration achieves a very high effectiveness, with an F-Measure well above 0.8 in 7 out of 10 datasets. Two of the datasets with low effectiveness, namely D_{c3} and D_{c5}, contain so high levels of noise that F-Measure remains below 0.8 for all other approaches we consider. We also observe that most datasets are processed in few seconds with much less than 1 Gb of main memory — even D_{c6} which contains than 60,000 entities. Overall, this workflow combines high effectiveness with high efficiency, despite requiring no parameter fine-tuning. Its high effectiveness of should be attributed to its consistently high recall, which stems from its schema-agnostic functionality: by considering all attribute values during Block Building and Entity Matching, it successfully overcomes the typical levels of noise.

By optimizing its parameters, the best configuration of this workflow yields much higher precision, which leads to a significantly higher F-Measure, as shown in Table 4(c). The running time and the memory consumption are also significantly reduced, in most cases. Even D_{c7} is now processed in less than a minute with less than 1 Gb of main memory.

Budget-agnostic, schema-based workflow. For the join-based workflow of Fig. 4(a), we performed grid search to identify the best matching rule. We applied Jaccard similarity in combination with all thresholds in $[0.05, 0.95]$ with a step of 0.05 to all attributes. The only exception is D_{c8}, where we exclusively considered the attribute "*name*", which has the highest coverage for both individual datasets (43,34%). Note also that edit distance is not suitable for most datasets, as they involve a large number of tokens per attribute. For each dataset, we consider the matching rule that maximizes F-Measure and use PPJoin to accelerate it. Unique Mapping Clustering or Connected Components is then applied for Clean-Clean or Dirty ER, respectively, using the same similarity threshold as the matching rule. The resulting performance is reported in Table 4(d), while the exact matching rules are reported in Table 3(b).

We observe that this workflow underperforms the best schema-agnostic (blocking-based) one in all cases. Its F-Measure is also significantly lower than the default blocking-based workflow for 8 datasets. However, it reduces the memory footprint up to 50% and the running time even by a whole order of magnitude. For example, it process the 3.35 million entities of D_{c8} within just 15 min. The reason is that it reduces the search space for duplicates to the values of a single attribute, unlike the blocking-based workflow that considers all attribute values per dataset.

Overall, we can conclude that the schema/join-based workflow achieves excellent performance in some cases, with its F-Measure exceeding 0.8 for minimum running time and memory footprint. However, it is not robust, as its effectiveness often remains very low, despite making the most of schema knowledge. In contrast, the schema-agnostic workflow consistently achieves excellent effectiveness after fine-tuning and allows for a default, parameter-free configuration. This is impossible with the join-based workflow, as it depends on the schema of the dataset at hand.

Table 4
Performance of JedAI's budget-agnostic workflows and the main baseline methods over the real data.

	D_{c1}	D_{c2}	D_{c3}	D_{c4}	D_{c5}	D_{c6}	D_{c7}	D_{c8}	D_{cora}	D_{cddb}
(a) Blocks' performance for the schema-agnostic workflows										
PQ	0.372	0.085	0.018	0.120	0.008	0.013	0.025	0.025	0.776	0.002
PC	1.000	0.910	0.882	0.998	0.991	0.987	0.948	0.863	0.498	0.993
(b) Default configuration of the schema-agnostic workflow										
Pr	0.473	0.902	0.538	0.975	0.310	0.888	0.908	0.806	0.876	0.874
Re	1.000	0.836	0.645	0.989	0.878	0.952	0.834	0.819	0.816	0.856
F1	0.643	0.867	0.586	0.982	0.458	0.919	0.869	0.813	0.845	0.865
RT	1.1 s	1.3 s	12.0 s	2.0 s	8.3 s	23.5 s	91.0 s	14.5 h	5.5 s	65 s
Memory	50 Mb	50 Mb	200 Mb	200 Mb	300 Mb	750 Mb	1.5 Gb	100 Gb	150 Mb	1.4 Gb
(c) Best configuration of the schema-agnostic workflow										
Pr	0.788	0.949	0.576	0.993	0.590	0.946	0.905	0.841	0.912	0.858
Re	1.000	0.856	0.641	0.992	0.753	0.949	0.875	0.821	0.819	0.886
F1	0.881	0.900	0.607	0.993	0.662	0.947	0.889	0.831	0.863	0.872
RT	1.0 s	1.1 s	4.5 s	1.3 s	5.3 s	30.0 s	46.0 s	12.7 h	850 ms	65.7 s
Memory	50 Mb	50 Mb	100 Mb	100 Mb	150 Mb	750 Mb	750 Mb	100 Gb	50 Mb	1.4 Gb
(d) Best configuration of the schema-based workflow										
Pr	0.755	0.884	0.663	0.978	0.829	0.953	0.931	0.833	0.751	0.278
Re	0.933	0.438	0.423	0.932	0.552	0.775	0.499	0.370	0.859	0.719
F1	0.834	0.585	0.517	0.954	0.663	0.855	0.649	0.512	0.802	0.401
RT	200 ms	367 ms	499 ms	608 ms	478 ms	14 s	7.7 s	15.2 min	328 ms	566 ms
Memory	50 Mb	50 Mb	50 Mb	50 Mb	50 Mb	100 Mb	300 Mb	38 Gb	50 Mb	50 Mb
(e) Best configuration of Magellan [3]										
F1	1.000	0.436	0.491	0.984	0.791	0.923	0.826	–	–	–
(f) Best configuration of DeepMatcher [66]										
F1	1.000	0.628	0.693	0.984	0.693	0.947	0.872	–	–	–

Baseline systems. We now compare JedAI with the state-of-the-art in the field, Magellan [3] and DeepMatcher [66]. Neither of them is applicable to Dirty ER, as both require two duplicate-free datasets as input. For the Clean-Clean ER datasets $D_{c1}, \ldots, D_{c6}$, we consider the top performance that is reported in [66] among all configurations and dataset versions (we could not reproduce it ourselves, due to the lack of necessary details and the human-in-the-loop approach of both systems). We could not apply either system to D_{c8}, due to the extreme schema heterogeneity, but we were able to apply both of them to D_{c7}. The resulting performance appears in Table 4(e) and (f).

We observe that for D_{c1}, D_{c3} and D_{c4}, at least one of the baseline systems outperforms the best configuration of JedAI's budget- and schema-agnostic workflow to a significant extent. In the last two cases, though, the F-Measure remains below 0.8. For the remaining datasets, JedAI outperforms them to a lesser or a greater extent. This means that there is no clear winner among the three systems in terms of effectiveness (their overall difference is statistically insignificant). In terms of time efficiency, though, JedAI is much faster, since Magellan and DeepMatcher spend a considerable time for training their matching models — after performing Blocking with the help of an expert and after labeling a considerable number of comparisons [66], two operations that are quite time consuming. For D_{c7}, for instance, they were trained over 16,800 pairs of entities (1:10 match/non-match), with Magellan requiring 21 s for training and DeepMatcher taking ∼40 min per epoch, i.e., >3.5 h in total for 10 epochs.

Scalability analysis. To examine JedAI's scalability, we applied the default budget- and schema-agnostic (block-based) end-to-end workflow to the seven synthetic Dirty ER datasets in Table 1. We also applied a default matching rule, *JaccarSim*(*all_tokens*_1, *all_tokens*_2) > 0.4, executed by PPJoin and followed by Connected Components with the same similarity threshold (no other rule achieved reasonable effectiveness across all datasets). The resulting performance appears in Fig. 7. In the legends, the prefixes *B* and *J* indicate the block- and the join-based workflow, respectively.

In Fig. 7(a), we observe that the blocking-based workflow excels in recall, which consistently exceeds 95%. Precision fluctuates between 84% and 94% and F-Measure between 96% and 89%. The larger the dataset is, the lower both measures get, since the ER task becomes harder — the candidate matches increase quadratically, while the duplicates increase linearly. In contrast, the join-based workflow consistently achieves perfect precision, while recall and F-Measure remain practically stable at 60% and 75%, respectively. These patterns verify the capabilities of the schema-agnostic workflow and the limitations of the schema-based one.

The corresponding running times appear in Fig. 7(b). We have considered both the serial execution of each pipeline, which uses a single core, as well as the parallel, Spark-based execution, which employs all available cores of our server (i.e., 32 executors). We observe that most approaches exhibit similar run-times for the smaller datasets, requiring just ∼3 min for processing D_{300k}. The only exception is the serialized schema-based workflow, whose run-time increases quadratically with the input data size, due to the low similarity threshold it employs (the similarity join techniques like PPJoin are crafted for Jaccard thresholds higher than 0.7 [38,39]). We can conclude that the overhead of Apache Spark does not pay-off for limited workloads. For the larger datasets, though, the parallel implementations take the lead, involving significantly lower run-times. In particular, the parallel schema-agnostic implementation scales *sublinearly* with the increase in input data size, unlike the serial implementation, which scales *superlinearly*.

The linear scalability is demonstrated in Fig. 7(c), which depicts speedup $s(n) = n_{min} \times RT(n_{min})/RT(n)$ as we vary the number of cores (i.e., executors) in $n \in \{2, 4, 8, 16\}$ when processing the two synthetic largest datasets, D_{1M} and D_{2M}. We observe that the speedup is very close to the ideal, linear one in all cases, demonstrating the high scalability of JedAI's Spark-based implementation. For both workflows, the speedup is slightly larger for D_{2M} than for D_{1M}, which implies that it is affected by the size of the workload.

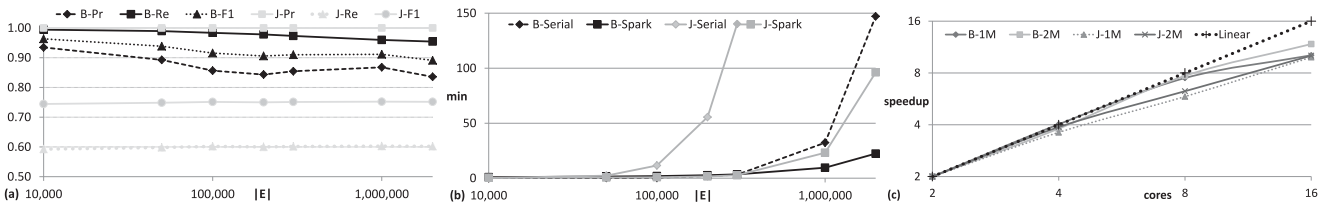

Fig. 7. Performance of the default budget-agnostic workflows over the synthetic Dirty ER datasets w.r.t. (a) effectiveness, (b) running time, and (c) speedup.

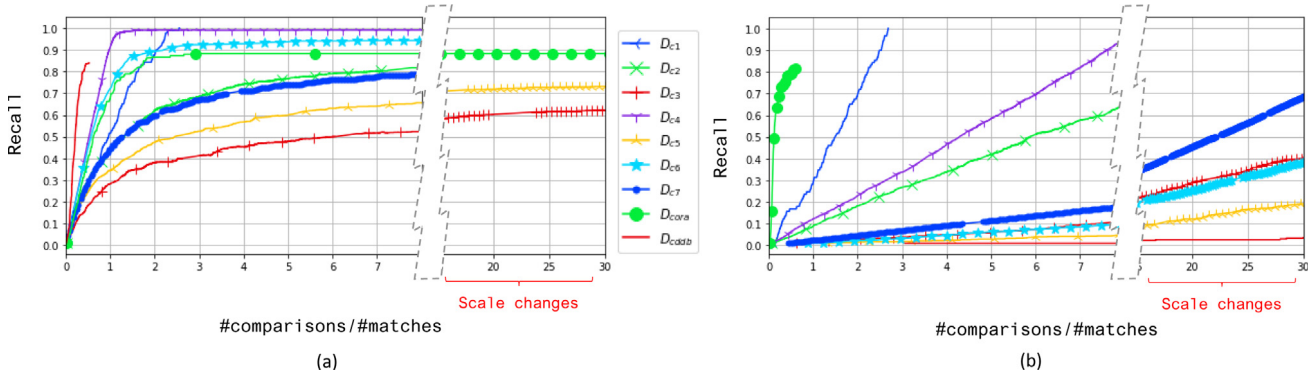

Fig. 8. Performance of the (a) budget-aware, schema-agnostic workflow, and (b) its budget-agnostic counterpart with respect to progressive recall.

Budget-awareness experiments. We now evaluate the benefits of budget-aware end-to-end workflows. To this end, we estimate *progressive recall*, i.e., the rate at which recall increases (on the vertical axis) as the number of executed comparisons increases (on the horizontal axis). The larger the resulting area under curve (AUC), the more duplicates are detected early on and the better is the progressive functionality.

We applied Progressive Global Top Comparisons to the blocks produced by the schema-agnostic workflow that is used in Section 9, i.e., Token Blocking, Block Purging, Block Filtering and CNP. This approach essentially orders all comparisons in the final set of blocks in decreasing Jaccard weight. Its performance across all real datasets appears in Fig. 8(a). The performance of the corresponding batch process, which executes the pairwise comparisons in arbitrary order appears in Fig. 8(b). As the AUC suggests, the budget-aware workflow consistently outperforms its budget-agnostic counterpart to a large extent. Similar results are obtained when comparing the schema-based budget-aware workflow, which relies on the top-k similarity join algorithm, with its budget-agnostic counterpart, with relies on token-based similarity joins executed by PPJoin (see Section 9).

The only exceptions are the smallest datasets for Clean-Clean and Dirty ER, i.e., D_{c1} and D_{cora}, where the budget-agnostic workflow exhibits a performance equivalent to the budget-aware one, despite not using prioritization. The reason is that in both cases, the precision of the final set of blocks (*PQ* in Table 4) is so high that most comparisons involve duplicates without the need to prioritize them. Another reason is that D_{cora} involves very large equivalence clusters so the transitive closure raises recall to a significant extent even with a small number of identified duplicates.

Overall, the outcomes demonstrate that progressive recall increases significantly faster for the latter ones, as we execute more comparisons. This applies to both the schema-agnostic and schema-based end-to-end workflows.

10. Related work

We now present an overview of the main ER systems, explaining how JedAI goes beyond the current state-of-the-art. In Table 5, we report the methods that are implemented by each ER tool for the two main steps of their end-to-end workflows: Blocking and Entity Matching. The latter step incorporates filtering methods, which accelerate the execution of similarity joins, and techniques for estimating similarity measures. The main technical characteristics per ER tool are presented in Table 6. Note that we do not report the supported output formats, as not all tools facilitate the storage of final or intermediate results. Some tools merely present their results through their GUI, while others return custom data structures that require further processing by the user.

We focus on open code systems, as these fulfill one of the main challenges arising in data integration [67], namely the development of extensible, open-source tools. However, few of these open code systems fulfill the second challenge [67], which requires them to process data of any structuredness. In fact, we can distinguish these systems into two broad categories according to the data structuredness they handle.

The first category includes the open-source tools that are crafted for structured data, namely Magellan [3], Dedupe [68], DuDe [69], Febrl [65], FRIL [70], OYSTER [71], Record Linkage [72] and FAMER [32]. All of them apply a budget-agnostic, schema-based end-to-end workflow that typically consists of two steps: Blocking and Matching. For Blocking, each tool provides few custom or established methods, except for Febrl, which offers the schema-based implementation of the main hash- and similarity-based techniques. For Matching, each tool provides various similarity measures, with Magellan offering the main similarity join techniques, too, to accelerate the execution of the matching rules it has learned. Unlike JedAI, all tools disregard Block and Comparison Cleaning, while Dedupe and FAMER are the only tools that apply Clustering. The former applies a simple hierarchical method to enhance its entity matching process, whereas the latter implements various established techniques, focusing on Multi-source ER. Note also that only Febrl and FRIL offer a GUI for ease of use and that only Dedupe, Febrl and FAMER support parallelization for at least one workflow step.

The second category includes open-source link discovery frameworks, which are crafted for semi-structured data. Similar

Table 5
Methods per workflow step for the main open-source ER systems. Minoan-ER and Dedupe are the only systems that offer Block Processing and Clustering techniques, respectively, together with JedAI. See Fig. 3 for the methods implemented by JedAI.

Tool	Blocking	Entity Matching	
		Filtering	Similarity measures
(a) Systems for structured data			
Magellan [3]	SB, SN (also allows user-specified blocking patterns)	Overlap, Size, Prefix, Position, Suffix	Cosine, Dice, edit distance, Jaccard, overlap and overlap coefficient
Dedupe [68]	SB with learning-based techniques	–	Affine Gap Distance
DuDe [69]	SB, SN, Sorted blocks	–	BlockDistance, Cosine, DiceCoefficient, EuclideanDistance, Jaccard, JaroDistance, Jaro–Winkler, Levenshtein, MatchingCoefficient, MongeElkan, NeedlemanWunsch, OverlapCoefficient, SmithWaterman
Febrl [65]	SB, SN, Sorted blocks, Suffix Arrays, Extended Q-Grams, Canopy Clustering, StringMap	–	Bag-Dist, Dam-Le-Edit-Distance, EditDist,Editex, Jaro, Long-Common-Seq, Q-Gram, S-Gram, Smith-Water-Dist, Syll-Align-Dist, Winkler, stringEquality, Token-Set, TIME, Key-Difference, Numeric
FRIL [70]	SB, SN	–	Edit distance, Soundex, Q-gram, Equality(0–1)
OYSTER [71]	SB	–	
Record Linkage [72]	SB (with SOUNDEX)	–	Uses statistics (ML) for different attributes' equivalence metrics to attain patterns-probabilities for false match rates
CODI [73]	Logic-based constraints to exclude comparisons	–	Threshold-based edit-distance
LogMap [74]	Logic-based constraints to exclude comparisons	–	ISUB [75]
FAMER [32]	SB, SN, Q-Grams	–	Jaro–Winkler, TruncateBegin, TruncateEnd, EditDistance, MongeElkan, Jaccard, DICE, Overlap ExtendedJaccard, Longest Common Substring, Numerical Similarity Max Distance, Numerical Similarity Max Percentage
(b) Systems for semi-structured data			
KnoFuss [76]	Literal Blocking	–	Edit-distance (DATE, DiceCoefficient, Jaccard, Jaro, Jaro–Winkler, Overlap, MongeElkan, SmithWaterman, TokenBased, TokenWise)
SERIMI [77]	Logic-based constraints to exclude comparisons	–	n-gram based
Silk [6]	Multiblock	–	Jaro, jaro–Winkler, qGram, stringEquality, num, date, uriEquality, taxonomic, maxSet
LIMES [4]	Custom methods	PPJoin+, EdJoin, HR3, HYPPO, ORCHID	Cosine, ExactMatch, Jaccard, Jaro, Jaro–Winkler, Levenshtein, MongeElkan, Overlap, QGram, RatcliffObershelp, Soundex, Trigram
Winte.r [78]	SB, SN, Custom methods	–	Jaccard, N-Grams, Levenshtein EditDistance, Levenshtein, Maximum Of Token Containment, Numerical (Absolute-Differences, Deviation, Unadjusted deviation, percentage), DATE (custom based, user specified)
(c) Systems for both structured and semi-structured data			
JedAI	SB, (Extended) SN, (Extended) Suffix Arrays, MinHash/Superbit LSH, (Extended) Q-Grams	AllPairs, PPJoin, FastSS, PassJoin, PartEnum, EdJoin, SilkMoth	Group Linkage & Profile Matcher in combination with character & token n-gram graphs and containment, (normalized) value & overall graph similarity, or character & token n-grams and cosine, (generalized) Jaccard & SIGMA similarity, or pretrained embeddings and cosine similarity or Euclidean distance
Minoan-ER [79]	SB	–	Cosine, Jaccard

to the systems of the first group, they all implement a budget-agnostic, schema-based workflow that consists of Blocking and Matching. For Blocking, they generally offer custom approaches: KnoFuss [76] and SERIMI [77] apply Token Blocking to the literal values of RDF triples, whereas Silk [6] implements MultiBlock [80] and LIMES [4] its homonymous technique that relies on the triangle inequality in metric spaces. Only Winte.r [78] complements its custom methods with established ones, namely Standard Blocking and Sorted Neighborhood. All systems offer the main similarity measures, with LIMES further providing a set of established and custom similarity join techniques to accelerate their execution. No system implements Block Cleaning, Comparison Cleaning or Clustering. Silk and LIMES are the only systems that provide a GUI and support parallelization.

Unlike these tools, JedAI is capable of processing data of any structuredness. This is also accomplished by MinoanER [79], but it offers exclusively a budget- and schema-agnostic end-to-end workflow that runs on top of Apache Spark. Instead, JedAI fully realizes the three dimensional Entity Resolution of Fig. 1. Another advantage is that JeAI applies seamlessly to both Clean-Clean and Dirty ER, whereas tools like Magellan and the link discovery frameworks are restricted to Clean-Clean ER. Finally, JedAI offers a *learning-free* functionality that can operate with default configurations, independently of a ground-truth. It needs the ground-truth only for fine-tuning the parameters of its end-to-end pipelines and for benchmarking purposes.

In contrast, the *learning-based* functionality of systems like Magellan, Silk and LIMES requires a labeled dataset for its supervised operation. Without such a dataset, they cannot learn

Table 6
Technical features of the main open-source ER systems. LB stands for Learning-based, LF for learning-free, C-C for Clean-Clean ER and D for Dirty ER. The prioritization methods offered by JedAI for Budget-aware ER are listed in Fig. 5.

Tool	Input formats	Learning	GUI	Language	Parallelization	Task	Budget-aware ER
(a) Systems for structured data							
Magellan [3]	CSV	LB	Yes	Python	–	C-C	×
Dedupe [68]	CSV, SQL	LB	No	Python	Multi-core	C-C, D	×
DuDe [69]	CSV, JSON, XML, BibTex, Database (Oracle, DB2, MySQL and PostgreSQL)	LF	No	Java	–	C-C	×
Febrl [65]	CSV, text-based	LB, LF	Yes	Python	Multi-core	C-C, D	×
FRIL [70]	CSV, Excel, COL, Database	LB, LF	Yes	Java	–	D	×
OYSTER [71]	text-based	LF	No	Java	–	D	×
Record Linkage [72]	Database	LB	No	R	–	C-C, D	×
CODI [73]	RDF, OWL	LF	No	Java	–	C-C	×
LogMap [74]	RDF, OWL	LF	Yes	Java	–	C-C	×
FAMER [32]	JSON	LF	No	Java	Apache Flink	C-C	×
(b) Systems for semi-structured data							
KnoFuss [76]	RDF, SPARQL	LB	No	Java	–	C-C	×
SERIMI [77]	SPARQL	LF	No	Ruby	–	C-C, D	×
Silk [6]	RDF, SPARQL, CSV	LB	Yes	Scala	Apache Spark	D	×
LIMES [4]	RDF, SPARQL, CSV	LB	Yes	Java	Multi-core	C-C	×
Winte.r [78]	CSV, JSON, XML	LB	No	Java	–	C-C, D	×
(c) Systems for structured and semi-structured data							
JedAI	CSV, RDF/XML, RDF/HDT, RDF/JSON, OWL, Database (mySQL, PostgreSQL), SPARQL endpoint, Java serialized object	LF	Yes	Java	Apache Spark	C-C, D	✓
Minoan-ER [79]	RDF	LF	No	Java	Apache Spark	C-C, D	×

any blocking or matching model. Note that a labeled dataset is fundamentally different from the ground-truth: the latter simply comprises positive instances, i.e., the existing pairs of matches, while the former should also include a carefully selected sample of negative instances (i.e., non-matching entities). The relative number and representativity of positive and negative instances affects significantly the performance of the learned model. No such restrictions apply to JedAI's learning-free functionality.

Note also that heavy human intervention is usually required in order to define domain and/or dataset-specific features for every supervised method in a learning-based pipeline. This is especially true for the recent crowd-sourced systems like Corleone [81] and Falcon [82] as well as human-in-the-loop systems SystemER [5]. This is not the case, though, with the learning-free approaches, which merely require users to fine-tune their generic (i.e., domain and/or dataset-agnostic) parameters. Generic supervised features are only employed by Deep Learning-based approaches, i.e., DeepER [83] and DeepMatcher [66], which are based on embeddings that are also supported by JedAI (see Section 4.5).

Overall, JedAI addresses successfully the four main challenges in building ER systems [3]: it requires no coding from its users, it provides guidelines for creating effective solutions, it covers the entire end-to-end pipeline and it exploits a wide range of techniques. The last two challenges are actually overcome in a way that allows for building millions of high-end end-to-end pipelines. Additionally, JedAI can be easily extended with new methods or even workflow steps and achieves high time efficiency, both for stand-alone and cluster systems.

Note that JedAI has already been presented in a short journal paper [48] and as a demo in past conferences [12,84,85]. The first releases, i.e., version 1 [84], version 2 [12] and version 2.1 [48], cover exclusively the serialized execution of the budget- and schema-agnostic workflow that is presented in Section 4.1, while providing a rather limited experimental analysis of its performance [48]. The serialized implementation of the batch schema-based workflow and of the budget- and schema-agnostic workflow are briefly presented in [85], without evaluating their relative performance. In this work, we provide a comprehensive experimental evaluation of the different types of workflows supported by JedAI along with a detailed qualitative and quantitative comparison with the state-of-the-art in the literature. We also facilitate the use and extension of JedAI by describing in more detail the implemented methods (Section 4) and their technical details (Section 6). Technical details are also provided for the new, parallel implementation of all approaches on top of Apache Spark (Section 7), while Section 8 offers practical guidelines for putting JedAI into practice.

11. Conclusions

We have presented in detail all important aspects of JedAI so as to facilitate practitioners, researchers and developers to integrate it into their own applications, making the most of it. We have also juxtaposed it with state-of-the-art tools in the field and performed an extensive experimental analysis of all types of end-to-end pipelines it produces.

In the future, we will extend JedAI in various ways. We plan to enrich it with supervised techniques, like Supervised Meta-blocking [86] and BLOSS [87], taking special care to facilitate the active learning process that might be required. We also intent to include constraints in a way that accommodates both generic, schema-agnostic features and rules (e.g., e_1 and e_2 can never be the same entity) as well as schema-based ones (e.g., entities with identical ASIN number are duplicates). Finally, we will extent JedAI with a library of techniques for integrating geospatial entities.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by the EU H2020 project ExtremeEarth (Grant agreement No. 825258).

References

[1] V. Christophides, V. Efthymiou, K. Stefanidis, Entity Resolution in the Web of Data, in: Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers, 2015.

[2] X.L. Dong, D. Srivastava, Big Data Integration, Morgan & Claypool Publishers, 2015.

[3] P. Konda, S. Das, G.C. Paul Suganthan, A. Doan, A. Ardalan, J.R. Ballard, H. Li, F. Panahi, H. Zhang, J.F. Naughton, S. Prasad, G. Krishnan, R. Deep, V. Raghavendra, Magellan: Toward building entity matching management systems, PVLDB 9 (12) (2016) 1197–1208.

[4] A.N. Ngomo, S. Auer, LIMES - A time-efficient approach for large-scale link discovery on the web of data, in: IJCAI, 2011, pp. 2312–2317.

[5] K. Qian, L. Popa, P. Sen, Systemer: A human-in-the-loop system for explainable entity resolution, PVLDB 12 (12) (2019) 1794–1797.

[6] J. Volz, C. Bizer, M. Gaedke, G. Kobilarov, Silk-a link discovery framework for the web of data, LDOW 538 (2009).

[7] G. Papadakis, L. Tsekouras, E. Thanos, Jedai code repository, 2017, https://github.com/scify/JedAIToolkit.

[8] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.

[9] G. Papadakis, G. Alexiou, G. Papastefanatos, G. Koutrika, Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data, PVLDB 9 (4) (2015) 312–323.

[10] G. Papadakis, J. Svirsky, A. Gal, T. Palpanas, Comparative analysis of approximate blocking techniques for entity resolution, PVLDB 9 (9) (2016) 684–695.

[11] E. Friedman, R. Eden, Gnu trove: High-performance collections library for java, 2013, http://trove4j.sourceforge.net/html/overview.html.

[12] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, M. Koubarakis, The return of jedai: End-to-end entity resolution for structured and semi-structured data, PVLDB 11 (12) (2018) 1950–1953.

[13] G. Simonini, S. Bergamaschi, H.V. Jagadish, BLAST: a loosely schema-aware meta-blocking approach for entity resolution, PVLDB 9 (12) (2016) 1173–1184.

[14] P. Christen, A survey of indexing techniques for scalable record linkage and deduplication, IEEE TKDE 24 (9) (2012) 1537–1555.

[15] G. Papadakis, E. Ioannou, C. Niederée, P. Fankhauser, Efficient entity resolution for large heterogeneous information spaces, in: ACM WSDM, 2011, pp. 535–544.

[16] A.N. Aizawa, K. Oyama, A fast linkage detection scheme for multi-source information integration, in: International Workshop on Challenges in Web Information Retrieval and Integration, 2005, pp. 30–39.

[17] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: VLDB, 2001, pp. 491–500.

[18] M.A. Hernández, S.J. Stolfo, The merge/purge problem for large databases, in: SIGMOD, 1995, pp. 127–138.

[19] A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: VLDB, 1999, pp. 518–529.

[20] J. Ji, J. Li, S. Yan, B. Zhang, Q. Tian, Super-bit locality-sensitive hashing, in: NIPS, 2012, pp. 108–116.

[21] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, W. Nejdl, A blocking framework for entity resolution in highly heterogeneous information spaces, IEEE TKDE 25 (12) (2013) 2665–2682.

[22] G. Papadakis, G. Papastefanatos, T. Palpanas, M. Koubarakis, Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking, in: EDBT, 2016, pp. 221–232.

[23] J. Fisher, P. Christen, Q. Wang, E. Rahm, A clustering-based framework to control block sizes for entity resolution, in: KDD, 2015, pp. 279–288.

[24] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, W. Nejdl, Eliminating the redundancy in blocking-based entity resolution methods, in: JCDL, 2011, pp. 85–94.

[25] G. Papadakis, G. Koutrika, T. Palpanas, W. Nejdl, Meta-blocking: Taking entity resolutionto the next level, IEEE TKDE 26 (8) (2014) 1946–1960.

[26] A. McCallum, K. Nigam, L.H. Ungar, Efficient clustering of high-dimensional data sets with application to reference matching, in: KDD, 2000, pp. 169–178.

[27] B. On, N. Koudas, D. Lee, D. Srivastava, Group linkage, in: ICDE, 2007, pp. 496–505.

[28] S. Lacoste-Julien, K. Palla, A. Davies, G. Kasneci, T. Graepel, Z. Ghahramani, Sigma: simple greedy matching for aligning large knowledge bases, in: KDD, 2013, pp. 572–580.

[29] H.W. Kuhn, The hungarian method for the assignment problem, Nav. Res. Logist. Q. 2 (1–2) (1955) 83–97.

[30] L. Ramshaw, R.E. Tarjan, On Minimum-Cost Assignments in Unbalanced Bipartite Graphs, Tech. Rep. HPL-2012-40R1, HP Labs, Palo Alto, CA, USA, 2012.

[31] O. Hassanzadeh, F. Chiang, R.J. Miller, H.C. Lee, Framework for evaluating clustering algorithms in duplicate detection, PVLDB 2 (1) (2009) 1282–1293.

[32] A. Saeedi, M. Nentwig, E. Peukert, E. Rahm, Scalable matching and clustering of entities with FAMER, Complex Syst. Inform. Model. Quart. 16 (2018) 61–83.

[33] T.H. Haveliwala, A. Gionis, P. Indyk, Scalable techniques for clustering the web, in: Proceedings of the 3rd International Workshop on the Web and Databases (WebDB), 2000, pp. 129–134.

[34] D.T. Wijaya, S. Bressan, Ricochet: A Family of Unconstrained Algorithms for Graph Clustering, Brisbane, Australia, 2009, pp. 153–167.

[35] N. Bansal, A. Blum, S. Chawla, Correlation clustering, Mach. Learn. 56 (1–3) (2004) 89–113.

[36] S.M. Van Dongen, Graph Clustering by Flow Simulation (Ph.D. thesis), Utrecht University, 2000.

[37] G.W. Flake, R.E. Tarjan, K. Tsioutsiouliklis, Graph clustering and minimum cut trees, Internet Math. 1 (4) (2003) 385–408.

[38] Y. Jiang, G. Li, J. Feng, W. Li, String similarity joins: An experimental evaluation, PVLDB 7 (8) (2014) 625–636.

[39] W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, PVLDB 9 (9) (2016) 636–647.

[40] R.J. Bayardo, Y. Ma, R. Srikant, Scaling up all pairs similarity search, in: WWW, 2007, pp. 131–140.

[41] C. Xiao, W. Wang, X. Lin, J.X. Yu, Efficient similarity joins for near duplicate detection, in: WWW, 2008, pp. 131–140.

[42] T. Bocek, E. Hunt, B. Stiller, Fast Similarity Search in Large Dictionaries, Tech. Rep. ifi-2007.02, Department of Informatics, University of Zurich, 2007, http://fastss.csg.uzh.ch/.

[43] G. Li, D. Deng, J. Wang, J. Feng, PASS-JOIN: A partition-based method for similarity joins, PVLDB 5 (3) (2011) 253–264.

[44] A. Arasu, V. Ganti, R. Kaushik, Efficient exact set-similarity joins, in: VLDB, 2006, pp. 918–929.

[45] C. Xiao, W. Wang, X. Lin, Ed-join: an efficient algorithm for similarity joins with edit distance constraints, PVLDB 1 (1) (2008) 933–944.

[46] G. Simonini, G. Papadakis, T. Palpanas, S. Bergamaschi, Schema-agnostic progressive entity resolution, in: ICDE, 2018, pp. 53–64.

[47] C. Xiao, W. Wang, X. Lin, H. Shang, Top-k set similarity joins, in: ICDE, 2009, pp. 916–927.

[48] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, M. Koubarakis, Domain- and structure-agnostic end-to-end entity resolution with jedai, SIGMOD Rec. 48 (4) (2019) 31.

[49] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[50] G. Giannakopoulos, V. Karkaletsis, G.A. Vouros, P. Stamatopoulos, Summarization system evaluation revisited: N-gram graphs, TSLP 5 (3) (2008) 5:1–5:39.

[51] G. Papadakis, G. Giannakopoulos, G. Paliouras, Graph vs. bag representation models for the topic classification of web documents, World Wide Web 19 (5) (2016) 887–920.

[52] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, TACL 5 (2017) 135–146.

[53] J. Pennington, R. Socher, C.D. Manning, Glove: Global vectors for word representation, in: EMNLP, 2014, pp. 1532–1543.

[54] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: NIPS, 2013, pp. 3111–3119.

[55] Apache livy, 2020, https://livy.incubator.apache.org.

[56] G. Mandilaras, Jedai docker image, 2020, https://hub.docker.com/repository/docker/gmandi/jedai-webapp.

[57] A.F.M. Gad, Building Android Apps in Python Using Kivy with Android Studio, Springer, 2019, https://pyjnius.readthedocs.io.

[58] L. Kolb, A. Thor, E. Rahm, Multi-pass sorted neighborhood blocking with mapreduce, Comput. Sci.-Res. Dev. 27 (1) (2012) 45–63.

[59] A.S. Das, M. Datar, A. Garg, S. Rajaram, Google news personalization: scalable online collaborative filtering, in: Proceedings of the 16th International Conference on World Wide Web, 2007, pp. 271–280.

[60] G. Simonini, L. Gagliardelli, S. Bergamaschi, H.V. Jagadish, Scaling entity resolution: A loosely schema-aware approach, Inf. Syst. 83 (2019) 145–165, http://dx.doi.org/10.1016/j.is.2019.03.006.

[61] R. Vernica, M.J. Carey, C. Li, Efficient parallel set-similarity joins using MapReduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 495–506.

[62] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: 22nd International Conference on Data Engineering (ICDE'06), IEEE, 2006, p. 5.

[63] K. Bereta, H. Caumont, E. Goor, M. Koubarakis, D. Pantazi, G. Stamoulis, S. Ubels, V. Venus, F. Wahyudi, From copernicus big data to big information and big knowledge: A demo from the copernicus app lab project, in: CIKM, 2018, pp. 1911–1914.

[64] H. Köpcke, A. Thor, E. Rahm, Evaluation of entity resolution approaches on real-world match problems, PVLDB 3 (1) (2010) 484–493.

[65] P. Christen, Febrl-: an open source data cleaning, deduplication and record linkage system with a graphical user interface, in: KDD, 2008, pp. 1065–1068.
[66] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, Deep learning for entity matching: A design space exploration, in: SIGMOD, 2018, pp. 19–34.
[67] B. Golshan, A.Y. Halevy, G.A. Mihaila, W. Tan, Data integration: After the teenage years, in: PODS, 2017, pp. 101–106.
[68] M. Bilenko, R.J. Mooney, Adaptive duplicate detection using learnable string similarity measures, in: KDD, 2003, pp. 39–48.
[69] U. Draisbach, F. Naumann, Dude: The duplicate detection toolkit, in: 8th International Workshop on Quality in Databases, 2010.
[70] P. Jurczyk, J.J. Lu, L. Xiong, J.D. Cragan, A. Correa, Fine-grained record integration and linkage tool, Birth Defects Res. A 82 (11) (2008) 822–829.
[71] E. Nelson, J. Talburt, Entity resolution for longitudinal studies in education using oyster, in: IKE, 2011.
[72] M. Sariyar, A. Borg, K. Pommerening, Controlling false match rates in record linkage using extreme value theory, J. Biomed. Inform. 44 (4) (2011) 648–654.
[73] J. Huber, T. Sztyler, J. Nößner, C. Meilicke, CODI: combinatorial optimization for data integration: results for OAEI 2011, in: Proceedings of the 6th International Workshop on Ontology Matching, 2011.
[74] E. Jiménez-Ruiz, B.C. Grau, Logmap: Logic-based and scalable ontology matching, in: ISWC, 2011, pp. 273–288.
[75] G. Stoilos, G.B. Stamou, S.D. Kollias, A string metric for ontology alignment, in: ISWC, 2005, pp. 624–637.
[76] A. Nikolov, V. Uren, E. Motta, Knofuss: a comprehensive architecture for knowledge fusion, in: K-CAP, 2007, pp. 185–186.
[77] S. Araújo, D.T. Tran, A.P. de Vries, D. Schwabe, SERIMI: class-based matching for instance matching across heterogeneous datasets, IEEE TKDE 27 (5) (2015) 1397–1410.
[78] O. Lehmberg, C. Bizer, A. Brinkmann, Winte. r-a web data integration framework., in: International Semantic Web Conference (Posters, Demos & Industry Tracks), 2017.
[79] V. Efthymiou, G. Papadakis, K. Stefanidis, V. Christophides, MinoanER: Schema-agnostic, non-iterative, massively parallel resolution of web entities, in: EDBT, 2019, pp. 373–384.
[80] R. Isele, A. Jentzsch, C. Bizer, Efficient multidimensional blocking for link discovery without losing recall, in: Proceedings of the 14th International Workshop on the Web and Databases (WebDB), 2011.
[81] C. Gokhale, S. Das, A. Doan, J.F. Naughton, N. Rampalli, J.W. Shavlik, X. Zhu, Corleone: hands-off crowdsourcing for entity matching, in: C.E. Dyreson, F. Li, M.T. Özsu (Eds.), SIGMOD, 2014, pp. 601–612.
[82] S. Das, G.C. Paul Suganthan, A. Doan, J.F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, Y. Park, Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services, in: SIGMOD, 2017, pp. 1431–1446.
[83] M. Ebraheem, S. Thirumuruganathan, S.R. Joty, M. Ouzzani, N. Tang, Distributed representations of tuples for entity resolution, PVLDB 11 (11) (2018) 1454–1467.
[84] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, M. Koubarakis, Jedai: The force behind entity resolution, in: ESWC, 2017, pp. 161–166.
[85] G. Papadakis, L. Tsekouras, E. Thanos, N. Pittaras, G. Simonini, D. Skoutas, P. Isaris, G. Giannakopoulos, T. Palpanas, M. Koubarakis, Jedai^3 : beyond batch, blocking-based entity resolution, in: EDBT, 2020, pp. 603–606.
[86] G. Papadakis, G. Papastefanatos, G. Koutrika, Supervised meta-blocking, PVLDB 7 (14) (2014) 1929–1940.
[87] G.D. Bianco, M.A. Gonçalves, D. Duarte, BLOSS: effective meta-blocking with almost no effort, Inf. Syst. 75 (2018) 75–89.