Schema-agnostic Progressive Entity Resolution

Giovanni Simonini*, George Papadakis†, Themis Palpanas‡, Sonia Bergamaschi*

*University of Modena and Reggio Emilia, Italy †University of Athens, Greece ‡Paris Descartes University, France
*{name.surname}@unimore.it †gpapadis@di.uoa.gr ‡themis@mi.parisdescartes.fr

***Abstract*—Entity Resolution (ER) is the task of finding entity profiles that correspond to the same real-world entity. Progressive ER aims to efficiently resolve large datasets when limited time and/or computational resources are available. In practice, its goal is to provide the best possible partial solution by approximating the optimal comparison order of the entity profiles. So far, Progressive ER has only been examined in the context of structured (relational) data sources, as the existing methods rely on schema knowledge to save unnecessary comparisons: they restrict their search space to similar entities with the help of schema-based blocking keys (i.e., signatures that represent the entity profiles). As a result, these solutions are not applicable in Big Data integration applications, which involve large and heterogeneous datasets, such as relational and RDF databases, JSON files, Web corpus etc.**

To cover this gap, we propose a family of schema-agnostic Progressive ER methods, which do not require schema information, thus applying to heterogeneous data sources of any schema variety. First, we introduce a naïve schema-agnostic method, showing that the straightforward solution exhibits a poor performance that does not scale well to large volumes of data. Then, we propose three different advanced methods. Through an extensive experimental evaluation over 7 real-world, established datasets, we show that all the advanced methods outperform to a significant extent both the naïve and the state-of-the-art schema-based ones. We also investigate the relative performance of the advanced methods, providing guidelines on the method selection.

I. Introduction

When dealing with heterogeneous data, real-world entities may have different representations; for instance, they can be records in a relational database, sets of RDF triples, JSON objects, text snippets in a web corpus, etc. We call *entity profile* (or simply *profile*) each representation of a real-world *entity* in data sources. The task of identifying different profiles that refer to the same real-world entity is called Entity Resolution (ER) and constitutes a critical process that has many applications in areas such as Data Integration, Social Networks, and Linked Data [1], [2], [3].

ER can be distinguished into two broad categories [4], [5]: *(i)* Off-line or Batch ER, which aims to provide a *complete solution*, after all processing is terminated, and *(ii)* On-line or Progressive ER, which aims to provide the best possible *partial solution*, when the response time, or the available computational resources are limited. The latter is driven by modern *pay-as-you-go* applications that do not require the complete solution to produce useful results.

Progressive ER is becoming increasingly important [4], [5], as the number of data sources (e.g., on the Web), and the amount of available data (e.g., shopping catalogs) multiply, while at the same time the relevant applications have strict time requirements. (e.g., updating catalogs every few hours for large online retailers[1]). In this paper, we propose novel, schema-agnostic Progressive ER methods that significantly outperform the current state-of-the-art approach, even when this approach makes use of schema information.

Progressive Methods. A core characteristic of the existing methods for Progressive ER is that they rely on *blocking* in order to scale to large datasets [4], [5]. Blocking is a typical pre-processing step for Batch ER that aims to index together *likely-to-match* profiles into buckets (called *blocks*), according to an indexing criterion (called *blocking key*). Thus, comparisons are limited to pairs of profiles that co-occur in at least one block, avoiding the quadratic complexity of the naïve ER solution, which compares every profile with all others. In this way, progressive methods generate on-line the most promising *pairs of profiles* to be compared by a *match function*, i.e., a (usually) binary function that takes as input two profiles and decides if they are matching, or not.

In fact, progressive methods use blocking to generate on-line pairs of profiles in decreasing order of matching likelihood. So far, however, they have been exclusively combined with *schema-based* blocking [4], [5], which is specifically crafted for structured (relational) data. That is, they rely on schema knowledge in order to build blocks of low noise and high discriminativeness, assuming implicitly that all input records abide by a schema with attributes of known quality.

Limitations of Existing Approaches. The existing progressive methods suffer from the following major drawbacks:

(1) In practice, their fundamental assumption that schema is a-priori known holds for a small portion of the data we would like to handle. For instance, Web data typically comprises large, semi-structured, heterogeneous entities that manifest two main challenges of Big Data [1], [2]: *(i) Volume*, as they involve millions of entities that are described by billions of statements, and *(ii) Variety*, since their descriptions entail thousands of different attribute names. More generally, in a Big Data integration scenario, schema-alignment is too expensive and time consuming when multiple heterogeneous data sources are involved [2], thus yielding a prohibitively high cost for pay-as-you-go applications.

(2) Even when the schema assumption holds, there is plenty of room for improving the performance of existing schema-based progressive methods. We demonstrate this in Figure 1 over four established, real-world and diverse datasets: the state-

[1]https://www.nchannel.com/blog/challenges-ecommerce-catalog-management/

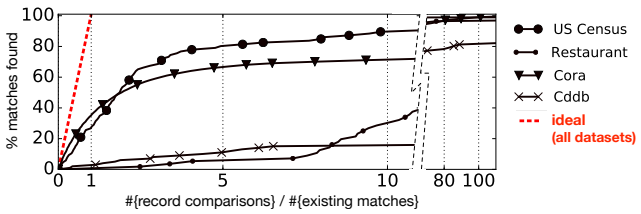

Fig. 1. The performance of Progressive Sorted Neighborhood on 4 real-world structured datasets.

of-the-art schema-based method, Progressive Sorted Neighborhood (PSN) [4], [5], finds only ∼60% and ∼85% of all matches for `Cora` and `US Census`, respectively, after executing 10 times the number of comparisons required by the optimal algorithm to identify 100% of the matches (i.e., 1 comparison per pair of duplicates). For the rest of the datasets, the performance is also far from optimal: for `Restaurant`, PSN identifies almost all matches only after performing two orders of magnitude more comparisons than the optimal algorithm, while for `Cddb`, it detects less than 80% of the existing matches with the same (excessive) number of comparisons.

Our Contributions. We propose novel and unsupervised methods for Progressive ER that inherently address the Variety of Big Data: they operate in a schema-agnostic fashion, which overrides the need to search for and identify highly discriminative attributes, rendering schema knowledge unnecessary. Our methods are also more effective in addressing the Volume of Big Data, since they identify matches earlier than the top-performing schema-based method. They actually go beyond the state-of-the-art in Progressive ER by introducing and exploiting *redundancy*, i.e., by associating every profile with multiple blocking keys. Instead, existing schema-based progressive methods typically rely on highly discriminative attributes, which yield *redundancy-free keys* such that two profiles cannot appear together in more than one block.

More specifically, our redundancy-based methods rely on two principles. The first one is called *similarity principle*, as it assumes that any two matching profiles have blocking keys that are closer in alphabetical order than those of non-matching ones. The second one is called *equality principle*, since it assumes that the matching likelihood of any two profiles is proportional to the number of blocks they share. Both principles have been successfully applied in Batch ER [6], but their application to the progressive context is non-trivial, as we show empirically. For this reason, we introduce more advanced methods for every principle.

Through an exhaustive experimental evaluation over 7 well-known datasets, we verify that similarity-based methods excel in structured datasets, outperforming even the state-of-the-art schema-based progressive method. These datasets typically involve a large portion of textual information, which provides reliable matching evidence when sorted alphabetically. In contrast, our equality-based method is the top-performer over semi-structured datasets (e.g., RDF data); it can exploit the semantics of the URIs that abound in this type of datasets, disregarding the useless information of URI prefixes, which introduce noise when sorted alphabetically.

On the whole, we make the following contributions:

- We introduce a *schema-agnostic* approach to Progressive ER, which inherently addresses the Variety issue of Big Data.
- We show that adapting existing schema-based methods to schema-agnostic Progressive ER is a non-trivial task: we introduce a naïve, schema-agnostic method, showing experimentally that it fails to address the Volume issue of Big Data.
- We present 3 novel advanced, schema-agnostic progressive methods, which successfully address both the Volume and the Variety challenges of Big Data. They are classified in two categories: those based on a sorted list of profiles, leveraging the *similarity principle*, and those based on a graph of profiles, leveraging the *equality principle*.
- We perform a series of experiments over 7 established, real-world datasets, demonstrating experimentally the superiority of our methods in comparison to the existing schema-based state-of-the-art method, both in terms of effectiveness and time efficiency. We also investigate the relative performance of our methods, highlighting the top-performing ones, and providing guidelines for method selection.

The rest of the paper is structured as follows: Section II discusses the main works in the literature, while Section III describes the background of our methods. We present a naïve schema-agnostic solution to Progressive ER in Section IV, and three advanced ones in Section V. We elaborate on our extensive experimental evaluation in Section VI and conclude the paper in Section VII, along with directions for future work.

II. Related Work

Schema-based Progressive Methods. The state-of-the-art progressive method is *Progressive Sorted Neighborhood* (**PSN**) [4], [5]. Based on Batch Sorted Neighborhood [7], it associates every profile with a schema-based blocking key. Then, it produces a *sorted list of profiles* by ordering all blocking keys alphabetically. Comparisons are progressively defined through a sliding window, w, whose size is *iteratively incremented*: initially, all profiles in consecutive positions (w=1) are compared, starting from the top of the list; then, all profiles at distance w=2 are compared and so on and so forth, until the processing is terminated.

However, the performance of PSN depends heavily on the attribute(s) providing the schema-based blocking keys that form the sorted list(s) of profiles. In case of low recall, the entire process is repeated, using multiple blocking keys per profile. As a result, PSN requires domain experts, or supervised learning on top of labeled data in order to achieve high performance. In contrast, our methods are completely unsupervised and schema-agnostic.

Two more schema-based methods were proposed in [5]: *Hierarchy of Record Partitions* (**HRP**) and *Ordered List of Records* (**OLR**). The main idea of HRP is to build a hierarchy

of blocks, such that the matching likelihood of two profiles is proportional to the level in which they appear together for the first time: the blocks at the bottom of the hierarchy contain the profiles with the highest matching likelihood, and vice versa for the top hierarchy levels. Thus, the hierarchy of blocks can be progressively resolved, level by level, from the leaves to the root. This approach has been improved in the literature in two ways: *(i)* OLR exploits this hierarchy in order to produce a list of records sorted by their likelihood to produce matches, involving a lower memory consumption than HRP at the cost of a slightly worse performance. *(ii)* A schema-based variation of HRP is adapted to the MapReduce parallelization framework for even higher efficiency in [8].

However, both HRP and OLR are difficult to apply in practice. The hierarchies that lie at their core can be generated only when the distance of two records can be naturally estimated through a certain attribute (e.g., product price) [5]. The number of the hierarchy layers, L, has to be determined a-priori, along with L similarity thresholds and the similarity measure that compares attribute values. Moreover, they both exhibit a performance inferior to PSN [5]. For these reasons, we do not consider these two methods any further.

Finally, Altowim et al. [9] propose a progressive *joint* solution in the context of multiple, relational datasets of different entity types. In joint ER [10], the result on one dataset can be exploited to resolve the others. As an example, let us consider a joint ER on a *movie dataset* and on an *actor dataset*: discovering matches among actors can help to determine whether two movies associated to those actors are matching too (and vice versa). Note that this approach is applicable only to relational data.

Crowdsourced (or Oracle) Methods. In *Crowdsourced* ER [11], humans are asked to label candidate profile pairs as either matching or non-matching, i.e., they are asked to behave like a binary *match function*. Such a function is typically assumed to be *perfect* (i.e., being equivalent to an *oracle* [12]) and *transitive* [13]. For example, given three profiles (p_1, p_2, p_3), if the crowd finds that p_1 matches with p_2, and p_2 with p_3, then the comparison between p_1 and p_3 is not crowdsourced, but is automatically deduced as a match. Progressive crowdsourced methods [13], [12], [14] exploit this transitivity to maximize the progressive recall of ER. In this work, though, we propose general methods for Progressive ER that are independent of the employed match function, i.e., we do not assume the match function to be transitive, nor to be perfect — a setting that is common for (non-crowdsourced) match functions [15].

Yet, our methods could be combined with the current state-of-the-art crowdsourced one, which is presented in [14]. Before submitting the first record pair to the crowd, this approach builds a list of record pairs sorted in decreasing *matching likelihood*. To assess this matching likelihood, it computes the string similarity (e.g., Jaro-Winkler or Jaccard) of all possible record pairs[2]. This is a prohibitively expensive pre-processing step when a low latency response is required, even if a blocking method is used to avoid the all-pairs comparisons. As an alternative, our methods could be employed to generate the sorted list of record pairs efficiently (instead of using them directly with match functions).

[2]The same pre-processing step is required by [11], [12] and [13].

III. Preliminaries

At the core of ER lies the notion of *entity profile* (or simply *profile*), which constitutes a uniquely identified set of attribute name-value pairs. An individual profile is denoted by p_i, with i standing for its id in a *profile collection* P. Two profiles $p_i, p_j \in P$ are called *duplicates* or *matches* ($\mathbf{p_i} \equiv \mathbf{p_j}$) if they represent the same real-world entity.

Depending on the input data, ER takes two forms [1], [2]: *(1)* Clean-clean ER receives as input two duplicate-free, but overlapping profile collections, P_1 and P_2, and returns as output all pairs of duplicate profiles they contain, $P_1 \cap P_2$. *(2)* Dirty ER takes as input a single profile collection that contains duplicates in itself and produces a set of equivalence clusters, with each one corresponding to a distinct profile.

To scale ER to large data collections, *blocking* is employed to cluster similar profiles into *blocks* so that it suffices to consider comparisons among the profiles of every block [16]. Each profile is *indexed* into blocks according to one or more criteria called *blocking keys*. If a blocking key depends on the schema(ta) of the data source(s), we call it *schema-based*, otherwise *schema-agnostic*.

An individual block is symbolized by b_i, with i corresponding to its id. The size of b_i (i.e., the number of profiles it contains) is denoted by $|b_i|$ and its cardinality (i.e., the number of comparisons it involves) by $\|b_i\|$. A set of blocks B is called *block collection*, with $|B|$ standing for its size (i.e., total number of blocks) and $\|B\|$ for its aggregate cardinality (i.e., the total number of comparisons entailed by B): $\|B\|=\sum_{b_i \in B} \|b_i\|$. The set of blocks associated with a specific profile p_i is denoted by B_i, and the *average number of profiles per block* by $|\bar{\mathbf{b}}|=\sum_{b \in B} |b|/|B|$. Finally, the comparison between profiles p_i and p_j is symbolized by c_{ij}, while $|\bar{\mathbf{p}}|$ stands for the *average number of blocking keys per profile*.

A. Progressive ER

In Batch ER, the profile comparisons entailed in a block collection B are executed without a specific order. Let T_o be the overall time required for performing Batch ER on B. Based on T_o, Progressive ER is formally defined by two requirements [4], [5]:

- *Improved Early Quality.* If both Progressive and Batch ER are applied to B and terminated at the same time $t \ll T_o$, then the former should detect significantly more matching profiles than the latter.
- *Same Eventual Quality.* The result produced at time T_o by Progressive and Batch ER should be identical. Even though progressive methods rarely run for so a long time as T_o, this requirement ensures their correctness, verifying that they yield the exact same outcome as batch methods.

In the following, we break the functionality of progressive methods into two phases:

i) The **initialization phase** takes as input the profiles to be resolved, builds the data structures needed for their processing, and processes them until producing the first (i.e., overall best) comparison.
ii) The **emission phase** returns the next best comparison from a list of candidate comparisons, ranked in non-increasing order of matching likelihood. In other words, it identifies the remaining pair of profiles that has the highest matching likelihood.

By definition, the initialization phase is activated just once, while the emission phase is repeated whenever a new comparison is requested for processing.

B. Core Data Structures

We now describe two fundamental data structures for our progressive methods: the *Blocking Graph* and the *Neighbor List*. Every method discussed in the following has at its core either the former or the latter. Note that both data structures are known from the literature, sometimes with different names (e.g., the Neighbor List is called *sorted list of records* in [5]).

Blocking Graph — This data structure lies at the core of Batch Meta-blocking [1], [2], [17], [18], [19], which aims at restructuring an existing block collection B into a new one B' that has similar recall, but significantly higher precision than B. Meta-blocking relies on the assumption that the matching likelihood of any two profiles is analogous to their degree of co-occurrence in a block collection. This means that B has to be generated by a blocking method that yields **redundancy-positive blocks**, where the similarity of two profiles is proportional to the number of blocks they share.

Based on redundancy, which is common for schema-agnostic blocking methods [18], Meta-blocking represents the block collection as a *blocking graph*. This is an undirected weighted graph $\mathcal{G}_B(V_B, E_B)$, where V_B is the set of nodes, and E_B is the set of weighted edges. Every node $n_i \in V_B$ represents a profile $p_i \in P$, while every edge $e_{i,j}$ represents a comparison $c_{i,j} \in B \subseteq P \times P$. A schema-agnostic *weighting function* is employed to weight the edges, leveraging the co-occurrence patterns of profiles in B: each edge is assigned a weight that is derived exclusively from the (characteristics of the) blocks its adjacent profiles have in common. For example, the **ARCS** function sums the inverse cardinality of common blocks, assigning higher scores to pairs of profiles sharing smaller (i.e., more distinctive) blocks: $ARCS(p_i, p_j, B) = \sum_{b_k \in B_i \cap B_j} 1/\|b_k\|$. Similarly, all other weighting functions [18], [19] assign high weights to edges connecting profiles with strong co-occurrence patterns and low weights to casual co-occurrences.

Example 1. *Figure 2a shows a set of entity profiles, P. Figure 2b illustrates the block collection that is generated by applying Token Blocking [20] to P, i.e., by creating a separate block for every token that appears in any attribute value of the input profiles (these tokens are called **attribute value tokens** in the following). Finally, Figure 2c depicts the Blocking Graph that is derived from the block collection of Figure 2b, when using the ARCS function for edge weighting.*

Note that materializing and sorting all edges of a blocking graph is impractical for large datasets, due to the resulting huge graph size (i.e., the number of edges it contains) [18]. For this reason, all existing Meta-blocking methods [18], [19] discard low-weighted edges through a *pruning algorithm*, while building the Blocking Graph. As a result, they retain only the most promising comparisons, which are collected and employed for Batch ER. Based on such a Blocking Graph, we present in Section V-B a novel algorithm that generates comparisons in a progressive way.

Neighbor List — The *Neighbor List* is the core data structure of Sorted Neighborhood [7] and its derived methods (i.e., PSN [4], [5]). It is a *list of profiles* that is generated by sorting all profiles alphabetically, according to the blocking keys that represent them. This data structure is exploited to generate comparisons under the assumption that the matching likelihood of any two profiles is analogous to their *proximity* after sorting.

The Neighbor List can be built from *schema-based* or from *schema-agnostic* blocking keys and is typically employed to generate blocks: a window slides over the Neighbor List, and blocks correspond to groups of profiles that fall into the same window. The size of the window is iteratively incremented. The resulting blocks are called **redundancy-neutral blocks**, because the similarity of two profiles is not related to the number of blocks they share; the corresponding blocking keys might be close when sorted alphabetically, but rather dissimilar.

Example 2. *To understand the notion of redundancy-neutral blocks, consider the sorted schema-agnostic blocking keys (i.e., the attribute value tokens) of the profiles in Figure 2a, which are depicted in Figure 2d. The keys `'carl'` and `'ellen'` are placed in consecutive positions, but the corresponding profiles have nothing in common. Figure 2e shows the Neighbor List that corresponds to this sorted list of schema-agnostic blocking keys.*

Note that in the schema-agnostic Neighbor List, every profile typically has **multiple placements** (e.g., once for each attribute value token). Hence, multiple distances can be measured for any pair of matching profiles. In Section V-A, we present two approaches that leverage this phenomenon to improve the early quality of Progressive ER.

IV. Naïve Method

Schema-based progressive methods (see Section II) are hard to apply in a domain like Web data, where Variety renders the selection of schema-based blocking keys into a non-trivial task. Yet, we can convert the state-of-the-art schema-based progressive method (PSN) into a schema-agnostic one with minor modifications, as explained below. However, our experimental analysis (Section VI) shows that this method

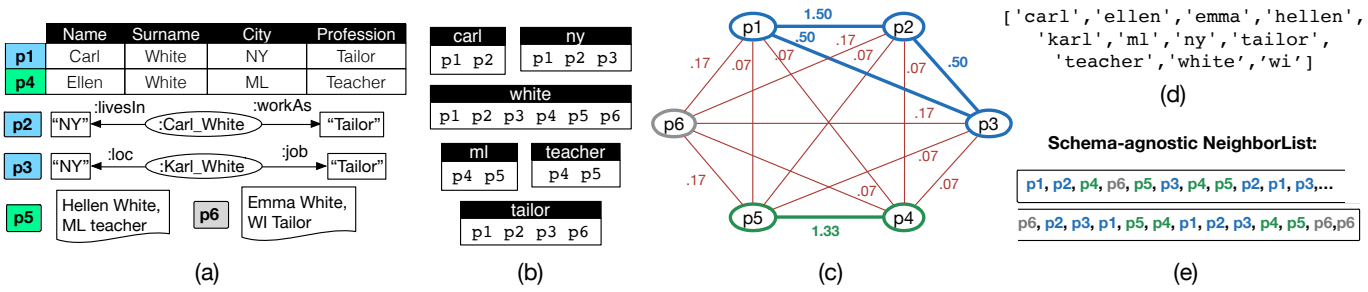

Fig. 2. (a) A set of profiles P that is extracted from a *data lake* with a variety of data formats: structured/relational data (p_1, p_4), semi-structured/RDF data (p_2, p_3) and unstructured/free-text data (p_5, p_6). Note that $p1 \equiv p2 \equiv p3$ and $p4 \equiv p5$. (b) The block collection B derived from P by applying Token Blocking to its profiles. (c) The Blocking Graph derived from B, with every edge representing a profile comparison that is weighted by the ARCS function. (d) The sorted list of attribute value tokens that appear in the profiles of P. (e) The corresponding schema-agnostic Neighbor List.

has inherent limitations that lead to poor performance, thus calling for the development of more advanced schema-agnostic progressive methods.

*1) Schema-Agnostic PSN (**SA-PSN**):* The main idea of this approach is to combine the sliding window with incremental size of PSN [5] with the Neighbor List of the schema-agnostic Sorted Neighborhood [6]. The resulting method is called *Schema-Agnostic Progressive Sorted Neighborhood* (SA-PSN).

Notice that consecutive places may involve the same profile (i.e., a profile which contains two alphabetically consecutive tokens), or two profiles from the same source. For this reason, SA-PSN allows comparisons only if they involve different profiles (Dirty ER), or profiles stemming from different sources (Clean-clean ER).

Overall, SA-PSN involves a parameter-free functionality that requires no schema-based blocking key definition and has low space and time complexities. Its space complexity is actually linear with respect to the size of the input dataset, $|P|$, i.e., $O(|\bar{\mathbf{p}}|\cdot|P|)$, because it merely keeps in memory the Neighbor List. Its time complexity is dominated by the sorting of blocking keys in alphabetical order, $O(|\bar{\mathbf{p}}|\cdot|P|\cdot\log(|\bar{\mathbf{p}}|\cdot|P|))$, thus ensuring high scalability.

On the flip side, SA-PSN may perform ***repeated comparisons***: the same pair of profiles might co-occur multiple times in the various windows. Moreover, the proximity of two profiles in the list may be partially random; if more than two profiles share the same blocking key, they are inserted with a relatively random order in the Neighbor List. We call this phenomenon ***coincidental proximity***. Note that PSN also suffers from coincidental proximity, which is a critical point to consider when devising the schema-based blocking keys.

Example 3. *Figure 3 applies PSN and SA-PSN to the profiles of Figure 2a. For PSN, we assume that the schema of p_1 and p_4 describes all other profiles, even $p5$ and $p6$, which represent unstructured data and would require an information extraction preprocessing step. This assumption allows for defining a schema-based blocking key that concatenates the surname and the first two letters of the name. In this context, PSN in Figure 3a starts by emitting all comparisons produced by the initial window size, $w = 1$; then, it continues with those comparisons entailed by window $w = 2$ etc. The final pair of matches is emitted during the 15^{th} comparison, i.e., after raising the window size to $w = 5$. In Figure 3b, SA-PSN applies the same procedure to the schema-agnostic Neighbor List, finding all matching profiles within the initial window frame $w = 1$, after the 14^{th} comparison. Figure 3b also shows examples of: (i) repeated comparisons, e.g., c_{12} is emitted as the 1^{st} and the 9^{th} comparison within the same window frame, $w = 1$; and (ii) coincidental proximity, since all 6 profiles are associated with the token `white` and are placed in random order at the end of the Neighbor List.*

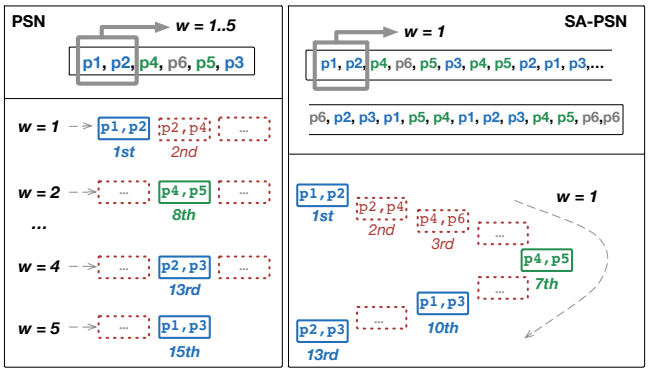

Fig. 3. Progressive emission of comparisons for (a) PSN, and (b) SA-PSN; dashed boxes indicate non-matching comparisons.

V. Advanced Methods

We now introduce more elaborate methods for schema-agnostic Progressive ER, using a broad spectrum of techniques. We distinguish them into two categories: the *similarity-based* ones, which employ a *weighted* Neighbor List, and the *equality-based* one, which employs a Blocking Graph. They are presented in Sections V-A and V-B, respectively. The former achieve the highest performance over structured (relational) data, while the latter excels over the semi-structured or unstructured Web data (Section VI).

Note that all our methods employ a data structure called **Comparison List**, which essentially constitutes a list of comparisons sorted in non-increasing order of matching likelihood. Its purpose is to store the best comparisons that were detected

during the initialization phase so that they are efficiently emitted during the emission phase. Whenever the Comparison List gets empty, it is refilled with the next batch of the best remaining comparisons, during the next emission phase.

A. Similarity-based Methods

In the previous section, we explained that SA-PSN suffers from two drawbacks: it contains numerous repeated comparisons and it defines a processing order of comparisons that is partially random, due to *coincidental proximity*. To address both disadvantages, we propose the use of a *weighted* Neighbor List, which employs a *weighting scheme* in order to associate every comparison with a numerical estimation of the likelihood that it involves a pair of matching profiles. This weighting scheme leverages the Neighbor List, with a functionality that is both schema- and domain-agnostic. Consequently, our approach addresses inherently the Variety of Web data.

We note that Meta-blocking [18] cannot be used in this case: it does not apply to the Neighbor List of SA-PSN, since it requires redundancy-positive blocks as input, whereas SA-PSN produces redundancy-neutral blocks. Therefore, SA-PSN calls for a novel weight-based functionality.

To this end, we propose the *Relative Co-occurrence Frequency* (**RCF**) weighting scheme. RCF counts how many times a pair of profiles lies at a distance of w positions in the Neighbor List and then normalizes it by the number of positions corresponding to each profile. To efficiently implement RCF and *weighted* Neighbor List, we go beyond Neighbor List by introducing a new data structure called **Position Index**. In essence, this is an inverted index that associates every profile (id) with its positions in the Neighbor List. Thus, it is generic enough to accommodate any weighting scheme that similarly to RCF relies on the co-occurrence frequency of profile pairs.

Below, we present two algorithms that exploit the RCF weighting scheme. Both of them are compatible with any other schema-agnostic scheme that infers the similarity of profiles exclusively from their co-occurrences in the incremental sliding window. The core idea of these algorithms is to trade a higher computational cost of the initialization phase, and probably the emission phase, for a significantly better comparison order.

*1) Local Schema-Agnostic PSN (**LS-PSN**):* This approach applies the selected weighting scheme only to the comparisons of a specific window size, thus defining a local execution order. At its core lie two data structures:

i) NL, which is an array that encapsulates the Neighbor List such that $NL[i]$ denotes the profile id that is placed in the i^{th} position of the Neighbor List. An exemplary NL array is shown in Step 1.i of Figure 4.

ii) PI, which stands for *Position Index*, is an inverted index that points from profile ids to positions in NL. It is implemented with an array that uses profile ids as indexes, such that $PI[i]$ returns the list of the positions associated with profile p_i in NL. This array accelerates the estimation of comparison weights, since it minimizes the computational cost of retrieving the neighbors of any profile in the current window, as described below.[3]

Algorithm 1: Initialization phase for LS-PSN.

```
   Input: (i) Profile collection P, (ii) Weighting scheme, wScheme
   Output: The overall best comparison
1  windowSize = 1;
2  ComparisonList ← ∅;
3  NL[] ← buildNeighborList(P);
4  PI[] ← buildPositionIndex(NL[]);
5  foreach p_i ∈ P do
6      distinctNeighbors ← ∅; // a set containing distinct neighbors
       frequency[] ← ∅;
7      foreach position ∈ PI[i] do
8          p_j ← NL[position+windowSize];
9          if isValidNeighbor(p_j) then
10             frequency[j]++;
11             distinctNeighbors.add(j);
12         p_k ← NL[position-windowSize];
13         if isValidNeighbor(p_k) then
14             frequency[k]++;
15             distinctNeighbors.add(k);
16     foreach j ∈ distinctNeighbors do
17         weight_{i,j} ← wScheme(frequency[j], j, i);
18         ComparisonList.add(getComparison(i, j, weight_{i,j});
19 sortInDescreasingWeight(ComparisonList);
20 return ComparisonList.removeFirst();
```

Algorithm 2: Emission phase for LS-PSN.

```
   Output: The next best comparison
1  if ComparisonList.isEmpty() then
2      windowSize++;
       /* repeat lines 5 - 20 in Algorithm 1          */
3  return ComparisonList.removeFirst();
```

Based on these data structures, the initialization phase of LS-PSN is outlined in Algorithm 1. Initially, it sets the window size to 1 (Line 1), considering only consecutive profiles. Then, it creates its data structures (Lines 2-4) and for every profile p_i (Line 5), it iterates over all its positions in the Position Index (Line 8). In every position, LS-PSN checks the neighbors in both directions, i.e., the profiles located $windowSize$ places before and after p_i (Lines 13 and 9, respectively) - provided that the corresponding positions are within the limits of the Neighbor List. For every neighbor p_j, LS-PSN checks if $j<i$ (Lines 10) and $k<i$ (Line 14) to avoid repeated comparisons. For every valid neighbor, LS-PSN increases its frequency (Lines 11 and 15) and adds it into the set of neighbors (Lines 12 and 16). Then, the overall weight for every comparison is computed according to the selected weighting scheme[4] (Line 18). Finally, all comparisons are aggregated and sorted from the highest weight to the lowest (Line 20) and the top one is returned (Line 21).

Note that Algorithm 1 pertains to Dirty ER. Yet, it can be adapted to Clean-clean ER with two minor modifications: *(i)*

[3]Instead of a Position Index, LS-PSN could use a hash index that has comparisons as keys and weights as values. This approach, however, would increase both the space and the time complexity of comparison weighting.

[4]Assuming a comparison between p_i and p_j, i.e., $c_{i,j}$, the corresponding RCF weight is equal to $\frac{frequency[j]}{PI[i].length()+PI[j].length()-frequency[j]}$.

Line 5 iterates over the profiles of P_1, and *(ii)* in Lines 10 and 14, a neighbor p_j is considered valid only if $p_j \in P_2$.

The emission phase of LS-PSN is illustrated in Algorithm 2 and is common for both Clean-clean and Dirty ER: if the Comparison List corresponding to the current window is not empty, the top weighted one is removed and returned as output. If the list is empty, the window size is incremented (Line 2) and the process for extracting all comparisons of the new window (Lines 5 - 20 in Algorithm 1) is repeated.

Example 4. *We demonstrate the functionality of LS-PSN by applying it to the profiles of Figure 2a. The result appears in Figure 4. Step 0 extracts all blocking keys and sorts them alphabetically, while Step 1.i forms NL and slides a window of size 1 over it. In Step 1.ii, we see the result of the nested loops in Lines 5 - 16 for the RCF weighting scheme for $windowSize$=1. In Step 1.iii, all comparisons are weighted and sorted from the highest to the lowest weight. Finally, the sorted comparisons are emitted one by one in Step 1.iv. Note that the first three comparisons correspond to the three pairs of duplicate profiles.*

Complexity Analysis. LS-PSN mainly keeps in memory the Neighbor List along with the Position Index. For both data structures, the space complexity is $O(|\bar{\mathbf{p}}|\cdot|P|)$, depending linearly on the number of input profiles. Similar to SA-PSN, the time complexity of the initialization phase is dominated by the sorting of blocking keys in alphabetical order, i.e., $O(|\bar{\mathbf{p}}|\cdot|P|\cdot\log(|\bar{\mathbf{p}}|\cdot|P|))$. Finally, the time complexity of the emission phase is usually constant, $O(1)$, simply emitting the next comparison from the Comparison List. Whenever this list gets empty, LS-PSN renews its contents by repeating their initialization phase, raising the complexity to $O(|\bar{\mathbf{p}}|\cdot|P|\cdot\log(|\bar{\mathbf{p}}|\cdot|P|))$; the only difference with the initialization phase is the incremented window size.

*2) Global Schema-Agnostic PSN (**GS-PSN**):* The main drawback of LS-PSN is the *local* execution order it defines for a specific window size. This means that LS-PSN is likely to emit the same comparison(s) multiple times, for two or more different window sizes, since it does not remember past emissions. GS-PSN aims to overcome this drawback by defining a *global* execution order for all the comparisons in a range of window sizes $[1, w_{max}]$. To this end, its initialization phase differs from Algorithm 1 in that Line 1 is converted into an iteration over all window sizes in $[1, w_{max}]$; this loop starts before Line 8 and ends before Line 20. This allows for a simpler emission phase, which just returns the next best comparison, until Comparison List gets empty.

Compared to LS-PSN, GS-PSN introduces one more configuration parameter, namely w_{max}, in order to eliminate all repeated comparisons in a particular range of windows. Thus, GS-PSN occupies more space, $O(w_{max}\cdot|\bar{\mathbf{p}}|\cdot|P|)$, due to its Comparison List, which contains 1 comparison per position in the Neighbor List for every window size, in the worst case. The time complexity of the initialization phase is the same as for LS-PSN, being dominated by the sorting of blocking keys in alphabetical order, while the emission phase exhibits a consistently constant time complexity, $O(1)$. Note also that GS-PSN takes into account more co-occurrence patterns than LS-PSN, when determining comparison weights. Consequently, its matching likelihood estimations are expected to be more accurate than those of LS-PSN.

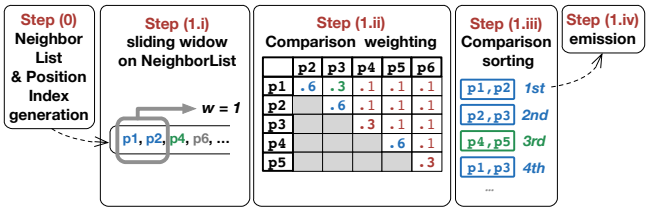

Fig. 4. Applying LS-PSN to the profiles of Figure 2a.

B. Equality-based Methods

The methods in this category rely on a redundancy-positive block collection that is derived from any suitable schema-agnostic blocking method or workflow [18]. From these blocks, we extract the Blocking Graph of Meta-blocking, using the weights of its edges as approximations for the matching likelihood of the corresponding comparison. In particular, we order the graph edges in decreasing weight in order to produce a sorted list of comparisons at the level of individual blocks or profiles. Below, we propose a novel algorithm of this type.

*1) Progressive Block Scheduling (**PBS**):* This algorithm is specifically designed for Progressive ER, but relies on a Batch ER technique. Indeed, *Block Scheduling* has been proposed in order to optimize the processing order of blocks in the context of Batch ER, based on the probability that they contain duplicates [1]. It assigns to every block a weight that is proportional to the likelihood that it contains duplicates and then, it sorts all blocks in descending weight order. Even though we would like to use such a functionality for Progressive ER, it is not applicable, because: *(i)* it does not specify the execution order of comparisons inside blocks with more than two profiles, and *(ii)* its weighting cannot generalize to Dirty ER, applying exclusively to Clean-clean ER.

We now describe our algorithm, PBS, in detail:

(1) PBS introduces a weighting mechanism that applies uniformly to Clean-clean and Dirty ER. In fact, it relies on the reasonable hypothesis that the smaller a block is, the more distinctive information it encapsulates and the more likely it is to contain duplicate profiles, and vice versa. Thus, it sets weights inversely proportional to block cardinalities (i.e., $1/\|b_i\|$) and sorts blocks in decreasing weights; the fewer comparisons a block entails, the higher it is ranked.

(2) PBS defines the processing order of comparisons inside every block using the Blocking Graph. This means that for each block b_i with $\|b_i\|>1$, PBS associates all comparisons with a weight derived from any schema-agnostic weighting scheme of Meta-blocking. Then, it sorts them from the highest weight to the lowest one.

It is worth noting that all repeated comparisons are discarded before computing their weight. In fact, the efficient detection of repeated comparisons is crucial for PBS. This

Algorithm 3: Initialization phase for PBS.

Input: (i) Profile collection P, (ii) Weighting scheme, $wScheme$
Output: The overall best comparison

1 $B \leftarrow$ buildRedundancyPositiveBlocks(P);
2 $B' \leftarrow$ blockScheduling(B);
3 $ProfileIndex \leftarrow$ buildProfileIndex(B');
4 $b_k \leftarrow B'$.removeFirst();
5 $ComparisonList \leftarrow \emptyset$;
6 **foreach** $c_{ij} \in b_k$ **do**
7 $\quad B_i \leftarrow ProfileIndex$.getBlocks($e_i$);
8 $\quad B_j \leftarrow ProfileIndex$.getBlocks($e_j$);
9 $\quad$ **if** *nonRepeated*(k, B_i, B_j) **then**
10 $\quad\quad w_{i,j} \leftarrow wScheme(k, B_i, B_j)$;
11 $\quad\quad ComparisonList$.add(getComparison($i$, j, $w_{i,j}$));
12 sortInDescreasingWeight($ComparisonList$);
13 **return** $ComparisonList.removeFirst()$;

Algorithm 4: Emission phase for PBS.

Output: The next best comparison

1 **if** $ComparisonList.isEmpty()$ **then**
$\quad$ `/* repeat lines 4 - 12 in Algorithm 3 */`
2 **return** $ComparisonList.removeFirst()$;

functionality is based on a data structure called **Profile Index**, which constitutes an inverted index that associates every profile with the ids of the blocks that contain it. In this way, it facilitates the efficient computation of comparison weights, similar to the Position Index of LS/GS-PSN. Note that the Profile Index is generic enough to accommodate any weighting scheme that is based on the block co-occurrence frequency of profile pairs.

In practice, the Profile Index is implemented as a two-dimensional array. The first dimension is of size $|P|$ such that $ProfileIndex[i]$ points to an array that contains all ids of the blocks involving profile p_i. As a result, the second dimension contains arrays of variable length. The block ids in every such array are sorted from the lowest to the highest one in order to ensure high efficiency for the two operations that are built on top of the Profile Index.

The first operation is the *Least Common Block Index* (**LeCoBI**) condition, which checks whether a comparison is repeated in the following way: given a comparison c_{ij} in block b_Y, the LeCoBI condition identifies the least common block id, X, between the profiles p_i and p_j and compares it with the id of b_Y, Y. If the two ids match ($X = Y$), c_{ij} corresponds to a new comparison. Otherwise $X < Y$, which means that c_{ij} has already been compared in block b_X, but is repeated in block b_Y. Note that $X > Y$ is impossible, because the id of every block indicates its position in the processing list after sorting all blocks in increasing cardinalities (i.e., b_k denotes the block placed in the k^{th} position after sorting). Note also that by ordering the block ids of the second dimension in increasing order, the Profile Index minimizes the checks required for detecting the least common block id, thus accelerating the LeCoBI condition.

The second operation is *Edge Weighting*, which infers the matching likelihood of every comparison from the weight of the corresponding edge in the blocking graph. Given a non-

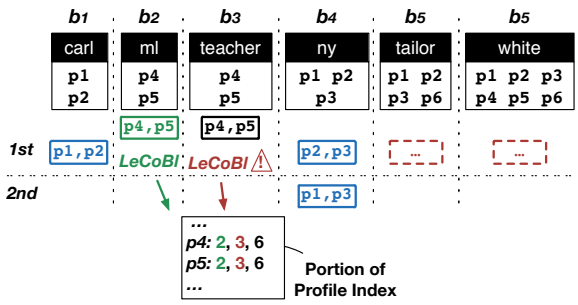

Fig. 5. Applying PBS to the blocks of Figure 2b.

repeated comparison c_{ij}, it compares the block lists associated with profiles p_i and p_j in order to estimate the number of blocks they share. This number, which lies at the core of practically all Meta-blocking weighting schemes [18], can be derived from the evidence provided by the Profile Index. Note that by ordering the block ids of its second dimension in increasing order, the Profile Index allows for accelerating Edge Weighting by traversing the two block lists in parallel.

On the whole, the initialization phase of PBS appears in Algorithm 3. Initially, it creates a redundancy-positive block collection and sorts its elements in non-decreasing order of comparisons (Lines 1-2). Then, it builds the corresponding Profile Index (Line 3) and goes on to remove the first (i.e., smallest) block, iterating over its comparisons (Lines 4-6). For every comparison c_{ij}, PBS gets the block lists that are associated with profiles p_i and p_j from the Profile Index (Lines 7-8). Based on these lists, it evaluates the LeCoBI condition, checking whether c_{ij} is repeated or not (Line 9). If c_{ij} is a new comparison, it is placed in the Comparison List along with the weight of the corresponding Blocking Graph edge (Lines 10-11). After processing all comparisons in the current block, the elements of the Comparison List are sorted in decreasing weight and the first one is emitted (Lines 12-13).

The emission phase of PBS appears in Algorithm 4. If the Comparison List is empty, it processes the next block $b' \in B'$, applying the Lines 4-12 of Algorithm 3 to it. Otherwise, the next best comparison is emitted from the Comparison List.

Example 5. *Figure 5 illustrates the functionality of PBS by applying it to the blocks of Figure 2b. First, it sorts them in non-decreasing cardinality and assigns to each one an incremental block id that indicates its processing order (note that we chose a random permutation of the blocks that have the same number of comparisons, without affecting the end result). Then, PBS processes the sorted list of blocks one block at a time, emitting iteratively the comparisons entailed in every block. Inside every block, all comparisons that satisfy the LeCoBI condition (i.e., non-repeated comparisons) are sorted according to the corresponding edge weight in the Blocking Graph of Figure 2c. For instance, when PBS processes* b_2*, the comparison* c_{45} *satisfies the LeCoBI condition, since the least common block id shared by* $p4$ *and* $p5$ *is 2. This means that PBS encounters* c_{45} *for the first time in* b_2*, assigning the edge weight 1.33 to it. In contrast, when PBS processes* b_3*, the comparison* c_{45} *does not satisfy the LeCoBI condition*

TABLE I
DATASET CHARACTERISTICS: ER TYPE, NUMBER OF ENTITY PROFILES, NUMBER OF *attribute names*, AND NUMBER OF EXISTING MATCHES.

	ER type	$\|P\|$	#attr.	$\|\mathcal{D}_P\|$
Structured Datasets				
census	Dirty ER	841	5	344
restaurant	Dirty ER	864	5	112
cora	Dirty ER	1.3k	12	17k
cddb	Dirty ER	9.8k	106	300
Large, Heterogeneous Datasets				
movies	Clean-clean ER	28k—23k	4—7	23k
dbpedia	Clean-clean ER	1.2M—2.2M	30k—50k	893k
freebase	Clean-clean ER	4.2M—3.7M	37k—11k	1.5M

anymore and is thus discarded.

Complexity Analysis. The space complexity of PBS is dominated by the space requirements of the Profile Index, i.e., $O(|\bar{\mathbf{p}}|\cdot|P|)$. The time complexity of its initialization time is dominated by the sorting of blocks in non-decreasing comparisons, i.e., $O(|B|\cdot\log|B|)$. In contrast, the cost of building the block collection B is insignificant, as it typically requires a single iteration over the input profiles, $O(|P|)$. Finally, the time complexity of its emission phase is usually constant, unless its Comparison List gets empty. In these cases, PBS refills its Comparison List with the sorted comparisons of the next block to processed. The time complexity of this procedure is very low, since it is dominated by the sorting of all comparisons in an individual block, i.e., $O(\|\bar{b}\|\cdot\log\|\bar{b}\|)$ on average, rather than the sorting of the entire block collection, B.

VI. EXPERIMENTS

System setup. All experiments have been performed on a server running Ubuntu 14.04, with 80GB RAM, and an Intel Xeon E5-2670 v2 @ 2.50GHz CPU. All methods are implemented in Java 8 and the code is publicly available[5].

Datasets. For the experimental evaluation, we employ 7 diverse real-world datasets that are widely adopted in the literature as benchmark data for ER [16], [18], [19], [21], [22]. Their characteristics are reported in Table I. The census, restaurant, cora, and cddb datasets are extracted from a single data source containing duplicated profiles, hence they are meant to test Dirty ER tasks. The remaining datasets (movies, dbpedia, and freebase) are suitable for testing scalability, as well as Clean-clean ER, since they are extracted from two different data sources, where matching profiles exist only between a source and another: movies from imdb.com and dbpedia.org; dbpedia from two different snapshots of DBpedia (dbpedia.org 2007-2009)[6]; freebase from developers.google.com/freebase/ and dbpedia.org (extracted from [22]). For all the datasets, the ground truth is known and provided with the data.

For the *structured* datasets, the *best schema-based blocking keys* for PSN are known from the literature [6], [16][7]. Note that the schema-based methods are inapplicable to the *large, heterogeneous* datasets. This is due to the size of the attribute set and the lack of a schema-alignment for Clean-clean datasets — movies has a total of 11 distinct attributes, but to the best of our knowledge no schema-based blocking key is known from the literature to perform well, while the schema-alignment for determining a schema-based blocking key is non-trivial. Finally, in dbpedia and freebase, there is a very small overlap in the attributes describing their profile collections.

Parameter configuration. We apply the following settings to all datasets. For LS-PSN and GS-PSN, we set w_{max}=20 for structured datasets and w_{max}=200 for large, heterogeneous datasets. For PBS, we use the *Token Blocking Workflow* to derive the redundancy-positive block collection. This workflow has been experimentally verified to address effectively and efficiently the Volume and Variety of Web data [18]. It consists of the following steps: *(1)* Schema-agnostic Standard Blocking [6], a.k.a. Token Blocking [20], creates a separate block for every attribute value token that stems from at least two profiles. *(2) Block Purging* [18] discards large blocks that correspond to stop words, involving more than 10% of the input profiles. *(3) Block Filtering* [18] retains every profile in 80% of its most important (i.e., smallest) blocks. *(4)* ARCS performs edge weighting on the Blocking Graph.

Metrics. Recall is typically employed to evaluate the effectiveness of a Batch ER method m over a profile collection P. It measures the portion of detected matches: $recall$=$|\mathcal{D}_m|/|\mathcal{D}_P|$, where $\mathcal{D}_m$ is the set of matches detected (emitted) by m, while $\mathcal{D}_P$ is the set of all matches in P.

In Progressive ER, we are interested in how fast matches are emitted. To illustrate this, we consider **recall progressiveness** by plotting the evolution of recall (vertical axis) with respect to the *normalized number of emitted comparisons* (horizontal axis): ec^*=$ec/|\mathcal{D}_P|$, where ec is the number of emitted comparisons at a certain time during the processing. The purpose of this normalization is twofold: *(i)* it allows for using the same scale among different datasets, and *(ii)* it facilitates the comparison of all progressive methods with the *ideal* one, which achieves recall=1 after emitting just the first $|\mathcal{D}_P|$ comparisons, i.e., at ec^*=1.

To facilitate the comparisons between progressive methods, we quantify their progressive recall using the **area under the curve** (AUC) of the above plot[8]. For a method m, we indicate with $AUC_m@ec^*$ the value of AUC for a given ec^*; for instance, $AUC_{PSN}@5$ is the area under the recall curve of the method PSN after the emission of ec=$5\cdot|\mathcal{D}_P|$ comparisons. To restrict $AUC_m@ec^*$ to the interval $[0,1]$, we normalize it with the performance of the ideal method: $AUC^*_m@ec^* = \frac{AUC_m@ec^*}{AUC_{ideal}@ec^*}$. $AUC^*_m@ec^*$ is called **normalized area under the curve**: higher values correspond to a better *progressiveness*, with the ideal method having AUC^*_{ideal}=1 for any value of ec^*.

[5]https://stravanni.github.io/progressiveER/

[6]Due to the constant changes in DBpedia, the two versions share only 25% of the *name-value* pairs, forming an non-trivial ER task [6], [18].

[7]See also the code at: https://sourceforge.net/projects/febrl and https://sourceforge.net/projects/erframework.

[8]The AUC expressed in function of ec (not the normalized ec^*) is known in the literature as *progressive recall* [14], and is employed for the same purpose.

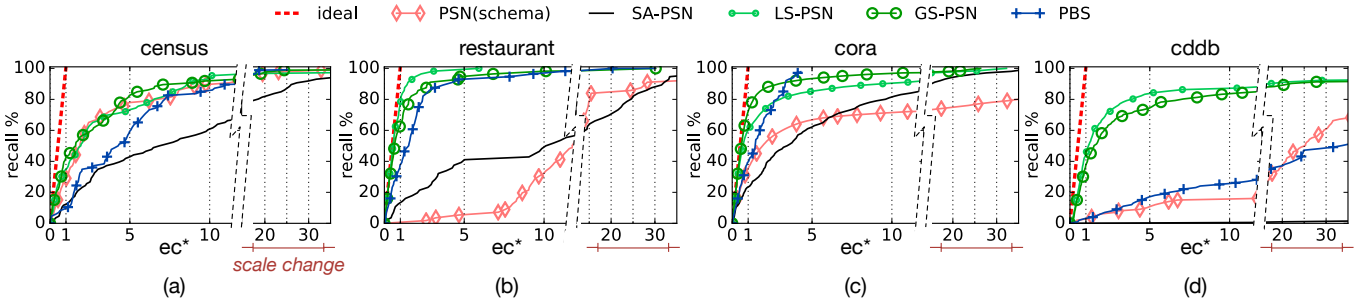

Fig. 6. Recall progressiveness over the structured datasets.

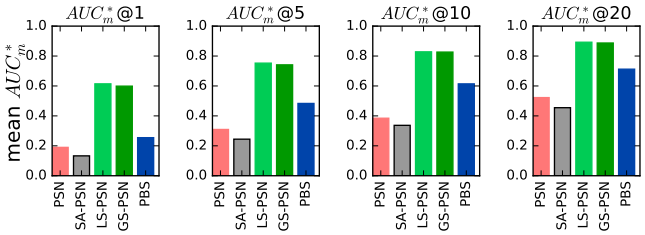

Fig. 7. Mean AUC^*_m over the structured datasets.

For the time performance evaluation of a method m, we consider the **initialization time** and the **comparison time**: the former is the time required to emit the first comparison, considering all the pre-processing steps (e.g., Token Blocking, Block Purging, Block Filtering for PBS); the comparison time is the average time between two consecutive comparison emissions. It includes both the emission time (i.e., the time required for generating the next best comparison) and the time required for applying the selected match function to that comparison.

Baselines. In the following, we use PSN and SA-PSN as baseline methods. As explained above, the best schema-based blocking keys, which are necessary for PSN, are only known for the Dirty ER datasets. For the Clean-clean ER ones, no such blocking keys have been reported in the literature. As a result, we consider only SA-PSN as baseline method for Clean-clean ER datasets.

A. Structured Datasets

We now compare our schema-agnostic methods against the state-of-the-art schema-based method, i.e. PSN [4], [5], on the structured datasets. We assess the relative effectiveness of all methods with respect to recall progressiveness. The corresponding plots appear in Figure 6. They depict the performance of all methods for up to ec^*=30, i.e., we measure the recall for a number of comparisons thirty times the comparisons required by the ideal method to complete each ER task. We focus, though, on the interval [0,10] in order to highlight the behavior of the methods in the early stage of ER, which is the most critical for pay-as-you-go applications.

We observe that the advanced schema-agnostic methods outperform PSN and SA-PSN across all datasets[9]. Only for `census` does PSN perform better than PBS (but not better that LS/GS-PSN) – see Figure 6a. This is because `census` contains very discriminative attributes, whose values are employed as blocking keys for PSN[10], identifying its duplicates with very high precision. Moreover, the profiles of `census` have short strings as attribute values: on average, every profile contains just 4-5 distinct tokens in its values. Inevitably, this sparse information has significant impact on the performance of similarity- and equality-based methods, restricting the co-occurrence patterns that lie at their core, i.e., the co-occurrences in windows for the former, and in blocks for the latter. The impact is larger in the latter case, due to the stricter definition of co-occurrence, which requires the equality of tokens, not just their similarity.

On the other hand, for datasets with a high token overlap of matching profiles, and low discriminative attributes, the recall progressiveness of PBS is significantly higher than (Figure 6b-c) or similar to (Figure 6d) that of schema-based PSN.

Among the advanced methods, we now list the best performer for each dataset. On `census` (Figure 6a), GS-PSN is the best performer, but LS-PSN is only slightly worse. On `restaurant` (Figure 6b) and `cddb` (Figure 6d), LS-PSN has the best recall progressiveness. On `cora` (Figure 6c), GS-PSN has the best initial progressiveness, but PBS reaches the highest recall from ec^*=4 on — note that the final recall of PBS is lower than 100%, because the underlying Token Blocking cannot identify all duplicates in `cora`.

We now compare all the methods with respect to their mean value of normalized area under the curve. Figure 7 shows the mean AUC^* of all methods across all structured datasets for four different values of ec^*: 1, 5, 10 and 20. We observe that, on average, for any level of AUC^*, LS-PSN and GS-PSN are the top performers, in particular for the earliest phase of Progressive ER: their AUC^*@1 is three times the AUC^*@1 of PSN and PBS.

Overall, we conclude that the best performing methods for structured datasets are LS-PSN and GS-PSN (the difference in their performance is almost negligible). Thus, the selection of one method over the other should be driven by the differences in their space and time complexities for the initialization and emission phases, depending on w_{max}. The higher w_{max} is, the higher gets the space complexity of GS-PSN in comparison to LS-PSN; thus, LS-PSN should be

[9]In Figure 6d, the curve of SA-PSN is too low to be visible, almost coinciding with the horizontal axis.

[10]Soundex encoded surnames concatenated to initials and zipcodes.

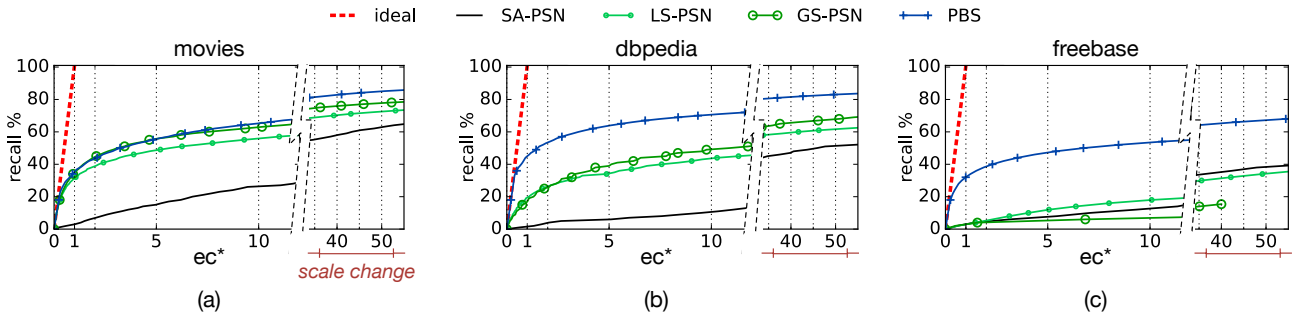

Fig. 8. Recall progressiveness over the large, heterogeneous datasets.

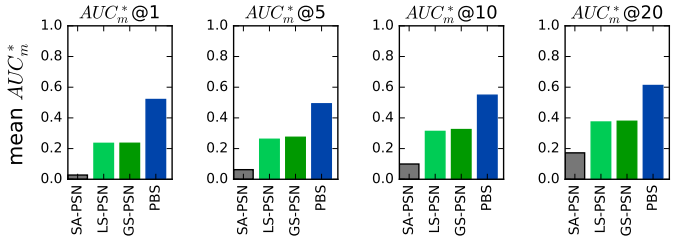

Fig. 9. Mean AUC^*_m over the large, heterogeneous datasets.

preferred when the availability of memory may be a issue. On the other hand, if memory is not an issue, GS-PSN should be preferred, since it avoids multiple emissions of the same comparisons.

B. Large, Heterogeneous Datasets

We now assess the relative performance of all methods with respect to recall progressiveness over the large, heterogeneous datasets `movies`, `dbpedia`, `freebase`. The corresponding plots appear in Figure 8.

The results confirm our intuition about the ineffectiveness of the naïve SA-PSN, since all advanced methods outperform it to a significant extent across all datasets.

The only exceptions are LS-PSN and GS-PSN[11] on `freebase`, which perform poorly: the performance of LS-PSN is similar to that of SA-PSN, while GS-PSN has lower recall progressiveness than SA-PSN, terminating before achieving a recall greater that 20%. The performance of these two advanced methods can be explained by the characteristics of the dataset. `Freebase` is composed of RDF triples. The extracted tokens consist of RDF keywords, URI, and other RDF properties, which generate a noisy Neighbor List, since their alphabetical ordering is often meaningless. Thus, the RCF weighting scheme cannot approximate correctly the similarity of the profiles. On the other hand, PBS is able to get the most of the semantics in URI tokens, due to the equality requirement, thus being more robust on `freebase` than the similarity-based methods.

Overall, PBS is the best performer across all the large, heterogeneous datasets (Figure 8). To quantify the difference in performance of all the methods, we compare them with respect to their mean value of normalized area under the curve. Figure 9 shows the mean AUC^* of all methods across all datasets for four different values of ec^*: 1, 5, 10 and 20. We observe that PBS is consistently the best performer, regardless of the level of AUC^*. Compared to LS-PSN and GS-PSN, its AUC^* is higher by more than 20%.

C. Time Efficiency Evaluation.

We note that our methods are general and decoupled from the match function employed to determine whether two profiles are matching or not. Yet, to assess their efficiency in terms of execution time, we evaluate them in combination with two match functions (ED and JS) based on the Damerau-Levenshtein edit distance [23] and the Jaccard similarity [24], respectively[12]. The former is meant to test the performance of our methods with an *expensive* match function, while the latter with a *cheap* one. The time complexities of edit-distance and jaccard-sim are $O(s \cdot t)$ and $O(s+t)$, respectively, where s and t are the lengths of the two strings to be compared (i.e., the two profiles compared with the match function).

The schema-based methods are not considered in this evaluation, since they inherently require an additional overhead time to select the blocking keys (and to perform the schema-alignment in the case of Clean-clean ER). There is a plethora of techniques to perform these two tasks [26], [27], [28], [29], but it is out of the scope of this work to determine which one is the best, since our proposed methods do not rely on them.

In Figure 10, we report the result of the time experiments on the datasets `movies` and `dbpedia`. (We do not consider `freebase` for this test, because comparing data composed of RDF triples would require more advanced match functions.) In particular, Figures 10a-d plot the performance of all methods, considering both the initialization time and the comparison time. The initialization times are listed in Figure 10e and are independent of the match function. We note that for all methods and datasets the emission time is negligible (w.r.t. the time required by the match functions), and we do not report it here: it is at least two orders of magnitude smaller than that required by the match functions to compare two profiles.

[11] On `freebase`, we limited the number of maximum comparisons of GS-PSN according to the available memory, i.e., 80GB.

[12] In a real-world scenario, each match function would require a threshold parameter to discriminate between matching and non-matching pairs, on the basis of their edit distance (or Jaccard similarity). Here, we are only interested in measuring the time performance, not the effectiveness of the match function; hence, we do not employ any threshold, and the outcome of the match function is assumed to be identical to the known ground truth. Furthermore, more complex similarity functions (such as [25]) could be employed to achieve high quality results.

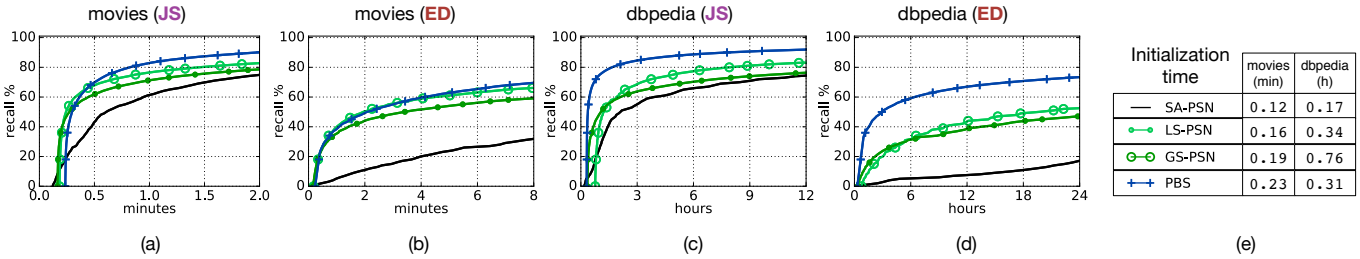

Fig. 10. Time experiments with jaccard-sim (a,c) and edit-dist (b,d); initialization times (e).

The results in Figure 10 clearly show that our advanced methods produce most of the matches much earlier than the baseline, with both the expensive and cheap match functions. LS-PSN is able to outperform SA-PSN since the early stages of the process, thanks to its fast initialization phase (see left part of Figures 10a-d). GS-PSN and PBS exhibit a similar behavior with minor exceptions: SA-PSN combined with a cheap matching function reaches the 20% recall mark faster than PBS on `movies` (Figure 10a), and faster than GS-PSN on `dbpedia` (Figure 10c).

Overall, PBS achieves higher level of recall much earlier than similarity-based methods on large and heterogeneous datasets. The only exceptions are LS-PSN and GS-PSN on `movies` (Figure 10a) with the cheap match function, where both similarity-based methods reach recall $= 40\%$ earlier than PBS, thanks to their lower initialization time. This is because the overhead of the *Token Blocking Workflow*, which lies at the core of PBS, becomes negligible when using an expensive match function (Figure 10b) or when applying it to large datasets (Figure 10c-d).

VII. Conclusions and Future Work

We have introduced *schema-agnostic* methods to maximize the recall progressiveness of Entity Resolution for *pay-as-you-go* applications, while addressing the Volume and Variety dimensions of Big Data. They can be distinguished into equality-based (PBS) and similarity-based methods (LS-PSN and GS-PSN). PBS was shown to be the best choice on large and heterogeneous datasets, while LS/GS-PSN on structured datasets described with a limited number of attributes. Moreover, our experimental evaluation with several real, structured datasets demonstrated that the proposed methods significantly outperform the schema-based state-of-the-art method in the field, PSN, identifying most of the matches much earlier.

An interesting direction for extending our work is to investigate how to combine the proposed methods with crowdsourcing strategies [13], [12], [14].

References

[1] V. Christophides, V. Efthymiou, and K. Stefanidis, *Entity Resolution in the Web of Data*. Morgan & Claypool, 2015.

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

[3] L. Getoor and A. Machanavajjhala, "Entity resolution: Theory, practice & open challenges," *PVLDB*, vol. 5, no. 12, pp. 2018–2019, 2012.

[4] T. Papenbrock, A. Heise, and F. Naumann, "Progressive duplicate detection," *IEEE TKDE*, vol. 27, no. 5, pp. 1316–1329, 2015.

[5] S. E. Whang, D. Marmaros, and H. Garcia-Molina, "Pay-as-you-go entity resolution," *IEEE TKDE*, vol. 25, no. 5, pp. 1111–1124, 2013.

[6] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika, "Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data," *PVLDB*, vol. 9, no. 4, pp. 312–323, 2015.

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

[8] Y. Altowim and S. Mehrotra, "Parallel progressive approach to entity resolution using mapreduce," in *ICDE*, 2017, pp. 909–920.

[9] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra, "Progressive approach to relational entity resolution," *PVLDB*, vol. 7, no. 11, pp. 999–1010, 2014.

[10] S. E. Whang and H. Garcia-Molina, "Joint entity resolution," in *ICDE*, 2012, pp. 294–305.

[11] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "Crowder: Crowdsourcing entity resolution," *PVLDB*, vol. 5, no. 11, pp. 1483–1494, 2012.

[12] N. Vesdapunt, K. Bellare, and N. N. Dalvi, "Crowdsourcing algorithms for entity resolution," *PVLDB*, vol. 7, no. 12, pp. 1071–1082, 2014.

[13] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng, "Leveraging transitive relations for crowdsourced joins," in *SIGMOD*, 2013, pp. 229–240.

[14] D. Firmani, B. Saha, and D. Srivastava, "Online entity resolution using an oracle," *PVLDB*, vol. 9, no. 5, pp. 384–395, 2016.

[15] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, "Swoosh: a generic approach to entity resolution," *VLDB J.*, vol. 18, no. 1, pp. 255–276, 2009.

[16] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *IEEE TKDE*, vol. 24, no. 9, pp. 1537–1555, 2012.

[17] G. Papadakis, G. Papastefanatos, and G. Koutrika, "Supervised meta-blocking," *PVLDB*, vol. 7, no. 14, pp. 1929–1940, 2014.

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

[19] G. Simonini, S. Bergamaschi, and H. V. Jagadish, "BLAST: a loosely schema-aware meta-blocking approach for entity resolution," *PVLDB*, vol. 9, no. 12, pp. 1173–1184, 2016.

[20] G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser, "Efficient entity resolution for large heterogeneous information spaces," in *WSDM*, 2011, pp. 535–544.

[21] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *PVLDB*, vol. 3, no. 1, pp. 484–493, 2010.

[22] A. Harth, "Billion Triples Challenge data set," Downloaded from http://km.aifb.kit.edu/projects/btc-2009/, 2009.

[23] G. Bard, "Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric," in *ACSW Frontiers 2007*, 2007, pp. 117–124.

[24] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[25] F. Benedetti, D. Beneventano, S. Bergamaschi, and G. Simonini, "Computing inter-document similarity with context semantic analysis," *Inf. Syst.*, 2018. [Online]. Available: https://doi.org/10.1016/j.is.2018.02.009

[26] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *ICDE*, 2002, pp. 117–128.

[27] P. Shvaiko and J. Euzenat, "Ontology matching: State of the art and future challenges," *IEEE TKDE*, vol. 25, no. 1, pp. 158–176, 2013.

[28] S. Bergamaschi, D. Ferrari, F. Guerra, G. Simonini, and Y. Velegrakis, "Providing insight into data source topics," *J. Data Semantics*, vol. 5, no. 4, pp. 211–228, 2016.

[29] M. Michelson and C. A. Knoblock, "Learning blocking schemes for record linkage," in *AAAI*, pp. 440–445.