ProgressER: Adaptive Progressive Approach to Relational Entity Resolution

YASSER ALTOWIM, King Abdulaziz City for Science and Technology, University of California, Irvine
DMITRI V. KALASHNIKOV, AT&T Labs Research, University of California, Irvine
SHARAD MEHROTRA, University of California, Irvine

Entity resolution (ER) is the process of identifying which entities in a dataset refer to the same real-world object. In relational ER, the dataset consists of multiple entity-sets and relationships among them. Such relationships cause the resolution of some entities to influence the resolution of other entities. For instance, consider a relational dataset that consists of a set of research paper entities and a set of venue entities. In such a dataset, deciding that two research papers are the same may trigger the fact that their venues are also the same. This article proposes a progressive approach to relational ER, named ProgressER, that aims to produce the highest quality result given a constraint on the resolution budget, specified by the user. Such a progressive approach is useful for many emerging analytical applications that require low latency response (and thus cannot tolerate delays caused by cleaning the entire dataset) and/or in situations where the underlying resources are constrained or costly to use. To maximize the quality of the result, ProgressER follows an adaptive strategy that periodically monitors and reassesses the resolution progress to determine which parts of the dataset should be resolved next and how they should be resolved. More specifically, ProgressER divides the input budget into several resolution windows and analyzes the resolution progress at the beginning of each window to generate a resolution plan for the current window. A resolution plan specifies which blocks of entities and which entity pairs within blocks need to be resolved during the plan execution phase of that window. In addition, ProgressER specifies, for each identified pair of entities, the order in which the similarity functions should be applied on the pair. Such an order plays a significant role in reducing the overall cost because applying the first few functions in this order might be sufficient to resolve the pair. The empirical evaluation of ProgressER demonstrates its significant advantage in terms of progressiveness over the traditional ER techniques for the given problem settings.

CCS Concepts: • **Information systems** → **Deduplication**; **Data cleaning**; **Entity resolution**; • **Theory of computation** → Data integration;

Additional Key Words and Phrases: Data cleaning, progressive computation, entity resolution, relational entity resolution, collective entity resolution, resolution plan, resolution workflow

ACM Reference format:
Yasser Altowim, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2018. Adaptive Progressive Approach to Relational Entity Resolution. *ACM Trans. Knowl. Discov. Data.* 12, 3, Article 33 (March 2018), 45 pages.
https://doi.org/10.1145/3154410

This work was supported in part by NSF Grants III-1429922, III-1118114, CNS-1059436, and by DHS Grant 206-000070. Y. Altowim was supported by KACST's Graduate Studies Scholarship.
Authors' addresses: Y. Altowim, King Abdulaziz City for Science and Technology, P.O Box 6086, Riyadh 11442, Riyadh, Saudi Arabia; email: yaltowim@kacst.edu.sa; D. V. Kalashnikov, AT&T Labs Research, 1 AT&T Way, Bedminster, NJ 07921; email: dvk@research.att.com; S. Mehrotra, Dept. of Computer Science, University of California, Irvine, CA 92697; email: sharad@ics.uci.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1556-4681/2018/03-ART33 $15.00
https://doi.org/10.1145/3154410

1 INTRODUCTION

The significance of data quality research is motivated by the observation that the effectiveness of data analysis technologies is closely tied to the quality of data on which the analysis is performed. That is why today's organizations spend a substantial percentage of their budgets on cleaning tasks such as removing duplicates, correcting errors, and filling missing values, to improve data quality prior to pushing the data through the analysis pipeline. In this article, we primarily focus on one data cleaning problem known as *Entity Resolution* (ER) (Benjelloun et al. 2009; Whang and Garcia-Molina 2012; Chen et al. 2009; Dong et al. 2005). ER is often used as a general term for several data disambiguation tasks such as *Deduplication*, *Record Linkage*, and *Entity Matching*. In this article, we specifically focus on the deduplication task, wherein the goal is to identify which entities in a dataset refer to the same real-world object.

A typical ER process often consists of three standard phases: (i) blocking, (ii) similarity computation, and (iii) clustering. Blocking divides the dataset into a set of (possibly overlapping) blocks such that entities in different blocks are unlikely to co-refer. The goal of blocking is to reduce the (often quadratic) process of applying ER on the entire dataset to that of applying ER on these small blocks. The similarity computation phase then computes the similarity between entities that reside in the same block. This phase is often computationally expensive as each comparison of a pair of entities may require applying several similarity functions on the pair to determine whether they co-refer or not. Finally, the clustering phase groups duplicate entities into clusters such that each cluster uniquely represents one real-world object.

Such an ER process is traditionally performed as an offline pre-processing step when creating a data warehouse prior to making the data available to analysis. This offline strategy, however, is not suitable for many emerging applications that require (near) real-time analysis or deal with continually arriving data or massive amounts of data that cannot be cleaned in its entirety. To address the needs of such applications, recent research has begun to consider how progressiveness can be incorporated in the context of ER (Whang et al. 2013b; Papenbrock et al. 2015).

Figure 1 illustrates the concept of progressive ER (Whang et al. 2013b). It depicts the quality of the cleaned data as a function of the resolution cost for three different types of ER approaches. Traditional ER algorithms produce results only after resolving the entire dataset, reaching the maximum quality at the end. The incremental curve shows the behavior of such algorithms when they are configured to constantly produce results while resolving the dataset. Progressive ER, on the other hand, aims to resolve the dataset such that the *rate* at which the data quality improves is maximized.

Such a progressive approach is motivated by several key perspectives. It is well suited for applications that require low latency response (and thus cannot tolerate delays caused by cleaning the entire dataset), and/or interactive applications that need to first compute initial analysis results and then progressively refine those results while the data is being cleaned. A progressive approach is also useful for small or medium sized enterprises that continually collect, clean, and analyze very large datasets. In this case, it would be counterproductive for such enterprises to spend their limited computational resources on cleaning each dataset in its entirety. However, if each dataset is resolved progressively using a limited cost budget, we would obtain the highest possible resolution quality for the dataset and achieve a better resource utilization. Such a cost-effective resolution method is even more financially desirable if these enterprises utilize public cloud infrastructures for their cleaning tasks.

The main idea of progressive ER is to prioritize/rank which pairs of entities to resolve first in order to identify as many duplicate entities in the dataset as early as possible. (Note that changing the order of resolving pairs may affect the final result (Benjelloun et al. 2009), which may hence

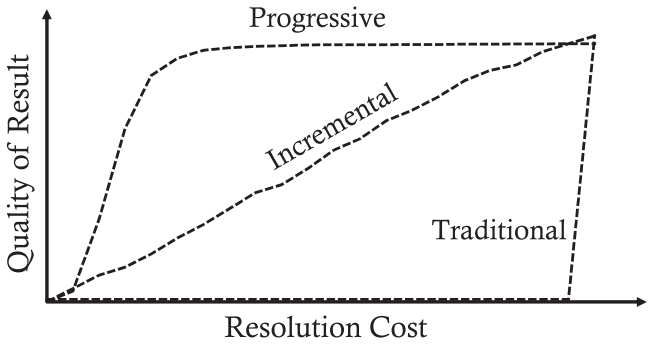

Fig. 1. The quality of the cleaned data as a function of the resolution cost for three types of ER approaches: Traditional, Incremental, and Progressive.

cause the final quality of the progressive approach to be different than those of traditional ER approaches, as illustrated in Figure 1.) In particular, the authors in Whang et al. (2013b) propose the idea of using "*hints*" for determining which entity pairs tend to be duplicates. The authors focus on how to resolve a given block progressively, and for that, they propose several concrete ways of constructing hints that can be then utilized with a variety of ER algorithms to prioritize the resolution within blocks. Subsequently, Papenbrock et al. (2015) proposes two hint-based progressive ER approaches that are built on top of the Sorted Neighbor (*SN*) algorithm (Hernández and Stolfo 1995).

While Whang et al. (2013b) and Papenbrock et al. (2015) address the problem of progressive ER, their solutions consider resolving only a single entity-set containing duplicates. In this article, we study the problem for *relational* ER (Culotta and McCallum 2005; Dong et al. 2005; Whang and Garcia-Molina 2012; Bhattacharya and Getoor 2007a). In relational ER, the dataset consists of multiple entity-sets and relationships among them. In such a dataset, the resolution of some entities might influence the resolution of other entities. For instance, consider a relational dataset that consists of a set of research paper entities and a set of venue entities. In this dataset, deciding that two papers are the same may trigger the fact that their venues are also the same. Prior research (Bhattacharya and Getoor 2007a; Dong et al. 2005; Culotta and McCallum 2005) has shown that such decision propagation via relationships can significantly improve data quality in datasets where such relationships can be specified.

Such dependencies among the resolution decisions influence our design in two ways. First, as more parts of the dataset are resolved, new information about which entities tend to co-refer becomes available. Thus, an *adaptive strategy* that dynamically revises its ranking is more suited for progressive relational ER. Second, unlike a single entity-set situation where there may not be a strong reason to prefer one block over another to resolve first, such a block-level prioritization is significantly more important when resolving relational datasets. For instance, resolving a block that influences many other parts of the dataset might significantly improve the performance of a progressive approach.

Motivated by the two factors, we develop ProgressER, which is an adaptive progressive approach to relational ER that aims to generate the highest quality result given a constraint on the resolution budget, specified by the user. To achieve adaptivity, ProgressER continually reassesses how to solve two key challenges: "which parts of the dataset to resolve next?" and "how to resolve them?" For that, it divides the resolution process into several *resolution windows* and analyzes the resolution progress at the beginning of each window to generate a *resolution plan* for the current window. A

resolution plan specifies which blocks and which entity pairs within blocks need to be resolved during the *plan execution* phase of that window. In addition, ProgressER associates, with each identified pair of entities, a *workflow*—the order in which to apply the similarity functions on the pair. Such an order plays a significant role in reducing the overall cost because applying the first few functions in this order might be sufficient to resolve the pair.

While an adaptive approach may help quickly identify duplicate pairs in relational datasets, such a technique must be implemented carefully so that the benefits obtained are not overshadowed by the associated overheads of adaptation. We address this challenge by designing efficient algorithms, appropriate data structures, and statistics gathering methods that do not impose high overheads. Our results in Section 8 show that ProgressER generates high-quality results using limited resolution budgets.

In addition, we propose an iterative algorithm for training *resolve* functions. A resolve function (explained in details in Section 2) is a function that is applied on a pair of entities to determine whether the two entities co-refer or not. Training a resolve function of an entity-set in relational ER is challenging as it depends on the effectiveness of the resolve functions of other entity-sets. For instance, when applied on two paper entities, the resolve function of the paper entity-set analyzes the similarity between their attribute values, and the decision of whether their venues co-refer or not. Such a decision is made by the resolve function of the venue entity-set. Hence, the resolve function of the paper entity-set should be trained in such a way that makes it resistant to any wrong decision made by the resolve function of the venue entity-set.

In summary, the *main contributions* of this article are as follows:

- —We propose an adaptive progressive approach to relational ER (Section 3).
- —We present benefit and cost models for generating a resolution plan (Section 4).
- —We show that the problem of generating a resolution plan is NP-hard, and thus propose an efficient approximate solution that performs well in practice (Section 5).
- —We define the concept of the contribution of similarity functions, and then show how the cost and contribution of multiple (possibly *correlated*) similarity functions can be exploited to generate workflows. We also propose an algorithm that learns the contribution values of functions from a training dataset (Section 6).
- —We propose an iterative algorithm for training resolve functions whose performance depends on that of other resolve functions (Section 7).
- —We experimentally evaluate our approach using publication and synthetic datasets and demonstrate the efficiency of the proposed solution (Section 8).

Our main contributions above optimize two dimensions of relational ER: the efficiency of the ER process and the quality of the results. Despite the fact that ProgressER exploits the relationships among entities to improve the quality of resolving entity pairs, all of our contributions (except for the iterative training algorithm presented in Section 7) are algorithms that focus on optimizing the efficiency of the ER process. More specifically, those algorithms aim at best utilizing the user-specified input resolution budget by iteratively (i) identifying a small set of useful entity pairs to resolve next, and (ii) resolving those identified pairs with the least amount of cost using effective workflows. The second and third points in the list of contributions above achieve (i), whereas the fourth point in the list achieves (ii).

2 NOTATION AND PROBLEM DEFINITION

In this section, we first develop our notation, and then define the problem of progressive ER. We summarize the notation used throughout this article in Table 1.

Table 1. Frequently Used Notation

Notation	Description
$\mathcal{D} = \{R, S, T, \dots\}$	A relational dataset $\mathcal{D}$ containing a number of entity-sets R, S, T, ...
r_i	An entity of entity-set R; $r_i \in R$.
$R.A = \{R.a_1, R.a_2, \dots\}$	$R.A$ is the set of attributes of entity-set R.
$\mathcal{B}_R = \{R_1, R_2, \dots\}$	$\mathcal{B}_R$ is the set of blocks of entity-set R.
$F_R = \{f_1^R, f_2^R, \dots\}$	F_R is the set of similarity functions of entity-set R.
$\mathcal{A}(f_i^R)$	The set of attributes on which the similarity function f_i^R is defined.
c_i^R	The average cost of applying the similarity function f_i^R on a pair of entities.
$L_{R \to S}$	An influence from entity-set R to S where R is called the *influencing* entity-set and S is called the *dependent* entity-set.
$L(\mathcal{D})$	The set of influences of $\mathcal{D}$.
$Inf(R)$ (resp. $Dep(S)$)	The set of all $L_{R \to X}$ (resp. $L_{X \to S}$) influences for any entity-set X.
$G = (V, E)$	G is a directed graph, where V is a set of nodes and E is a set of edges.
$BK(v_i)$	The block that contains both of the two entities that the node v_i represents.
$V(R_i)$	The set of all nodes of block R_i.
$V^+(R_i)$	The set of duplicate nodes of block R_i.
$V^*(R_i)$	The set of nodes of R_i that have been resolved in the previous windows.
$\mathfrak{R}_R$	The resolve function of entity-set R.
$\mathbf{f}_i$	The feature vector corresponding to node v_i that is given as input to a resolve function.
$\langle sim_i, dis_i \rangle$	The output of the resolve function when applied on a node v_i, where $sim_i \in [0, 1]$ and $dis_i \in [0, 1]$ represent the collected evidence that the two entities in v_i co-refer and are different respectively.
BG	The user-specified resolution budget.
$G^p = (V^p, E^p)$	G^p is a partially constructed graph, where $V^p \subseteq V$ is the set of instantiated nodes and $E^p \subseteq E$ is the set of instantiated edges.
RV, **UV**, and **CV**	Different classes of nodes. See Figure 5.
W	The duration of the plan execution phase of each window.
UB	The set of uninstantiated blocks.
$\mathbf{P} := \langle \mathbf{P}_B, \mathbf{P}_V \rangle$	A resolution plan $\mathbf{P}$ that specifies the set of blocks ($\mathbf{P}_B$) to be instantiated in the window and the set of nodes ($\mathbf{P}_V$) to be resolved in the window.
$\mathcal{S}$	The state of the graph G, which is the set of duplicate nodes in the graph.
$Benefit(v_i)$	The benefit of resolving the node v_i.
$\mathcal{P}(v_i)$	The probability that the node v_i is duplicate.
$Imp(v_i)$	The impact of resolving the node v_i.
$C^{ins}(R_i)$	The cost of instantiating the block R_i.
$C^{res}(v_j)$	The cost of resolving the node v_j.
t_j^{R+} (resp. t_j^{R-})	The positive (resp. negative) contribution value of the similarity function f_j^R.
$t_{j\|i}^{R+}$ (resp. $t_{j\|i}^{R-}$)	The conditional positive (resp. negative) contribution value that function f_j^R is expected to provide when applied on a pair of duplicate (resp. distinct) entities, on which f_i^R has already been applied.

2.1 Relational Dataset

Let $\mathcal{D}$ be a relational dataset that contains a number of entity-sets $\mathcal{D} = \{R, S, T, \dots\}$. Each entity-set R has a set of attributes $R.A = \{R.a_1, R.a_2, \dots, R.a_{|R.A|}\}$ and contains a set of entities of the same type $R = \{r_1, r_2, \dots, r_{|R|}\}$ such that each entity $r_i \in R$ has a value for each attribute $R.a_j \in R \cdot A$. Table 2 shows an example of a relational publication dataset that contains three entity-sets: Papers (P), Authors (A), and Venues (U). Entity-set P, for example, has six attributes: P_Id, Title, Abstract, Keywords, Authors, and Venue, and contains four entities $P = \{p_1, p_2, p_3, p_4\}$.

Table 2. Toy Publication Dataset

Entity-set Papers

Block	P_Id	Title	Abstract	Keywords	Authors	Venue
P_1	p_1	Transaction Support in Read Optimized and ...	...	{File System, Transactions}	$\{a_1, a_2\}$	u_1
	p_3	Transaction Support in Read Optimized and ...	...	{File System, Transactions}	$\{a_3, a_4\}$	u_3
P_2	p_2	Read Optimized File System Designs: ...	...	{File System, Database}	$\{a_1\}$	u_2
	p_4	Berkeley DB: A Retrospective	...	{File System, Database}	$\{a_3\}$	u_4

Entity-set Authors

Block	A_Id	Name	Email	Papers
A_1	a_1	Margo Seltzer	margo@harvard.edu	$\{p_1, p_2\}$
	a_3	Margo I. Seltzer	seltzer@gmail.com	$\{p_3, p_4\}$
A_2	a_2	Michael Stonebraker	stonebraker@mit.edu	$\{p_1\}$
	a_4	M. Stonebraker	stonebraker@ucb.edu	$\{p_3\}$

Entity-set Venues

Block	V_Id	Name	Papers
U_1	u_1	Very Large Data Bases	$\{p_1\}$
	u_3	VLDB	$\{p_3\}$
U_2	u_2	ICDE Conference	$\{p_2\}$
	u_4	IEEE Data Eng. Bull.	$\{p_4\}$

An attribute $R.a_i \in R.A$ is called a *reference* attribute if its values are references to other entities in other entity-sets. For example, the Authors and Venue attributes of entity-set P in Table 2 are reference attributes because their values are references to author and venue entities respectively.

Entity-set R is considered dirty if at least two of its entities r_i and r_j represent the same real-world object, and hence r_i and r_j are duplicate. The three entity-sets in Table 2 are dirty as they contain duplicate entities. Entity-set P contains one duplicate pair $\langle p_1, p_3 \rangle$, entity-set A contains two duplicate pairs $\langle a_1, a_3 \rangle$ and $\langle a_2, a_4 \rangle$, and entity-set U contains one duplicate pair $\langle u_1, u_3 \rangle$.

2.2 Standard Phases of ER

A typical ER process consists of several standard phases of data transformations. We list these phases below and explain how we instantiate some of them.

— *Blocking* (McCallum et al. 2000; Baxter et al. 2003) partitions each entity-set into (possibly overlapping) smaller blocks such that (i) if two entities might co-refer, they will be placed together into at least one block and (ii) if two entities are not placed together into at least one block, they are highly unlikely to co-refer. Blocking is the main divide-and-conquer style efficiency technique in ER. Its goal is to reduce the (often quadratic) process of applying ER on the entire dataset to that of applying ER on these small blocks.

In ProgressER, we assume that each entity-set $R \in \mathcal{D}$ is partitioned into a set of blocks $\mathcal{B}_R = \{R_1, R_2, \ldots, R_{|\mathcal{B}_R|}\}$. In our example in Table 2, the first column of each table represents the block in which the entities reside. For instance, entity-set P has two blocks P_1 and P_2; each contains two entities. For clarity, we will present this article as if the blocks of each entity-set do not overlap. Extending ProgressER to overlapping blocks is explained in details

Table 3. Description of the Similarity Functions

Function	Attributes	Similarity Algorithm
f_1^P	P.Title	Edit Distance
f_2^P	P.Abstract	TF.IDF
f_3^P	P.Keywords	Edit Distance
f_1^A	A.Name	Edit Distance
f_2^A	A.Email	Jaro–Winkler Distance
f_1^U	U.Name	Edit Distance

in Appendix A.1. We note that our experiments on the publication datasets use overlapping blocks.

— *Problem representation* maps the dataset into an internal data representation of the ER algorithm. We represent our problem as a graph, as detailed in Section 2.3.

— *Similarity computation* computes the similarity between entities in the same block. This phase is often computationally expensive as it may require comparing/resolving $O(n^2)$ pairs of entities per cleaning block of size n. Each resolution might apply several similarity functions on the pair of entities. A high-quality similarity function can be expensive in general as it may require invoking compute-intensive algorithms, consulting external data sources, and/or seeking human input through crowdsourcing (Nuray-Turan et al. 2012; Whang et al. 2013a; Vesdapunt et al. 2014).

Correspondingly, in ProgressER, each entity-set $R \in \mathcal{D}$ is associated with multiple similarity functions $F_R = \{f_1^R, f_2^R, \ldots, f_{|F_R|}^R\}$. Each function f_i^R takes as input a pair of entities $\langle r_j, r_k \rangle$ and returns a normalized similarity score for r_j and r_k—a value in the $[0, 1]$ interval. This score is computed by f_i^R by analyzing the values of r_j and r_k in some non-reference attributes $\mathcal{A}(f_i^R)$ of R. For instance, function f_1^P in Table 3 is defined on the Title attribute of entity-set P, i.e., $\mathcal{A}(f_1^P) = \{\text{Title}\}$. Given two titles of two papers, f_1^P returns their title similarity using the standard Edit-Distance algorithm (Smith and Waterman 1981). We note that ProgressER treats similarity functions as "black-boxes" in the sense that a function f_i^R can internally use any algorithm to compute the similarity between the values of the attributes $\mathcal{A}(f_i^R)$.

Each similarity function f_i^R in turn is associated with a cost value c_i^R, which represents an average cost of applying f_i^R on a pair of entities. ProgressER is agnostic to the way this cost is set. For instance, it can be set as the average execution time of f_i^R learned from the dataset or could be a unit-based cost set by the domain analysts, and so on.

After applying the similarity functions on a pair of entities, a *resolve* function is then used to determine, based on the outcomes of these similarity functions, whether this pair is duplicate or not. We explain our resolve functions in Section 2.3.

— *Clustering* groups duplicate entities into clusters based on the values computed by the resolve functions such that each cluster represents one real-world object, and each real-world object is represented by one cluster (Benjelloun et al. 2009; Dong et al. 2005; Nuray-Turan et al. 2012).

2.3 Relational Entity Resolution

The task of relational ER is to resolve all entity-sets of a given relational dataset. In addition to exploiting the similarity between the entity attribute values as in traditional ER, relational ER

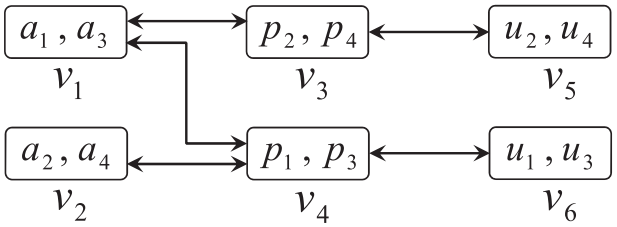

Fig. 2. The graph representation that corresponds to the publication dataset in Table 2.

also utilizes the relationships among the entity-sets to further increase the quality of the result (Bhattacharya and Getoor 2007a; Dong et al. 2005).

To illustrate, consider the paper pair $\langle p_1, p_3 \rangle$ in Table 2. By applying the similarity functions on it, we can decide that p_1 and p_3 co-refer. This decision can be propagated to the pair of their venues $\langle u_1, u_3 \rangle$. Based on that, we might then decide that $\langle u_1, u_3 \rangle$ co-refer, which might not have been achievable had we relied only on the similarity function of U since their venue names are spelled very differently.

Influence. In relational ER, there is an *influence* $L_{R \to S}$ from entity-set R to S if resolving some pairs from R can influence (provide evidence) for resolving some pairs from S. For example, the dataset in Table 2 has an influence $L_{P \to U}$ since if two papers are the same, their venues are likely to be the same as well. We denote the set of influences of $\mathcal{D}$ as $L(\mathcal{D})$. For instance, if $\mathcal{D}$ corresponds to the dataset in Table 2, then $L(\mathcal{D}) = \{L_{P \to A}, L_{P \to U}, L_{A \to P}, L_{U \to P}\}$.

In an influence $L_{R \to S}$, R is called the *influencing* entity-set and S is the *dependent* entity-set. We use $Inf(R)$ to denote the set of all $L_{R \to X}$ influences for any X, and $Dep(S)$ to denote the set of all $L_{X \to S}$ influences for any X. In the example in Table 2, $Inf(P) = \{L_{P \to A}, L_{P \to U}\}$ and $Dep(P) = \{L_{A \to P}, L_{U \to P}\}$.

ER graph. To capture decision propagation, it is often convenient to model the problem as a graph (Dong et al. 2005).

Graph $G = (V, E)$ is a directed graph, where V is a set of nodes and E is a set of edges. A node v_i is created for a pair of entities $\langle r_j, r_k \rangle$ *iff* there exists at least one block R_l that contains both r_j and r_k. The node v_i represent the fact that $\langle r_j, r_k \rangle$ could be the same, which needs to be further checked. An edge is created from node $v_i = \langle r_j, r_k \rangle$ to node $v_l = \langle s_m, s_n \rangle$ *iff* there exists an influence $L_{R \to S}$ and the resolution of v_i influences the resolution of v_l.

Figure 2 shows the graph corresponding to the publication dataset in Table 2. Node v_3 represents the possibility that entities p_2 and p_4 could be the same. An edge from v_1 to v_3 indicates that the resolution of v_1 influences the resolution decision of v_3 via $L_{A \to P}$. Thus, node v_3 has one *influencing* node v_1 via $L_{A \to P}$, whereas node v_1 has two *dependent* nodes v_3 and v_4 via $L_{A \to P}$.

The number of nodes in G depends upon the utilized blocking functions. Using aggressive functions often leads to small blocks, which causes the number of nodes to be small and hence improves the efficiency of the ER process. However, aggressive functions may often increase the chance of placing the two entities of a duplicate pair in two different blocks, sacrificing the quality of the result.

The number of edges, on the other hand, depends upon the number of influences in $\mathcal{D}$ and the fan-out degree of each influence $L_{R \to S}$, i.e., the average number of entity pairs of S that will be influenced by the resolution of an entity pair of R. For instance, the fan-out degree of $L_{P \to U}$ in Table 2 is at most 1 since a pair of paper entities may influence only one pair of venue entities. Note that an edge is created between these two pairs of entities only if the two entities of each pair

share at least one block, indicating that the blocking functions also affect the number of edges in the graph.

Having defined the graph, we now can develop some useful auxiliary notation that relates this graph to blocks. Let $BK(v_i)$ be the function that, given a node v_i, returns the block that contains both of the two entities that v_i represents. Let $V(R_i)$ be the set of all nodes of block R_i. Note that $V(R) = \bigcup_{i=1}^{|\mathcal{B}_R|} V(R_i)$.

From the graph representation point of view, the task of relational ER can be viewed as that of determining whether each node $v_i \in V$ represents a duplicate pair of entities or not. That resolution decision is based on the outputs of applying the similarity functions of R on v_i and the resolution decisions of v_i's influencing nodes. Such a decision can be made after the application of a *resolve* function on the node v_i.

Resolve function. Each entity-set R is associated with a resolve function $\mathfrak{R}_R$. When $\mathfrak{R}_R$ is applied on a node $v_i \in V(R)$, it outputs a pair of confidence values: a *similarity* confidence value $sim_i \in [0, 1]$ and a *dissimilarity* confidence value $dis_i \in [0, 1]$ that represent the collected evidence that the two entities in v_i co-refer and are different, respectively. Initially, both values are set to 0. After calling $\mathfrak{R}_R$ on v_i, the node v_i is marked as `duplicate` if $sim_i = 1$, as `distinct` if $dis_i = 1$, or as `uncertain` otherwise. A node is said to be `certain` if it is marked as either `duplicate` or `distinct`. We will denote the set of duplicate nodes of block R_i as $V^+(R_i)$ and of entity-set R as $V^+(R)$.

As is common, our resolve function $\mathfrak{R}_R$ takes as input a feature vector $\mathbf{f}_i$ that corresponds to node $v_i = \langle r_j, r_k \rangle$. The vector $\mathbf{f}_i$ is of size $|\mathbf{f}_i| = |F_R| + |Dep(R)|$ and encodes two types of features: (1) the outputs of applying the similarity functions of R on the pair $\langle r_j, r_k \rangle$ and (2) the sets of confidence value pairs of all nodes that influence v_i, where a set is included per each influence $L_{S \to R} \in Dep(R)$.

Example 2.1. Consider node v_3 in Figure 2. Suppose that the outputs of applying the similarity functions f_1^P, f_2^P, and f_3^P on v_3 are 0.2, 0.1, and 0.9, respectively, and that $\langle sim_1, dis_1 \rangle = \langle 1.0, 0.0 \rangle$ and $\langle sim_5, dis_5 \rangle = \langle 0.0, 1.0 \rangle$. The input to $\mathfrak{R}_P$ is therefore the feature vector $\mathbf{f}_3 = (0.2, 0.1, 0.9, \{\langle 1.0, 0.0 \rangle\}, \{\langle 0.0, 1.0 \rangle\})$. The first three values are the outputs of the three functions, while the fourth and fifth values are the sets of the confidence value pairs of all nodes influencing v_3 via $L_{A \to P}$ and $L_{U \to P}$, respectively.

We note that ProgressER treats resolve functions as "black-boxes" in the sense that the internal implementation of a resolve function can use any method to determine if the input two entities co-refer or not. For instance, it could use linear regression with appropriate weights assigned to different evidences (i.e., attributes and influences), a rule-based algorithm, or a machine-learning classifier.

This model of resolve functions is generic enough to capture a wide range of such possible implementations, including the ones proposed in Dong et al. (2005) and Whang and Garcia-Molina (2012). In our experiments on the publication datasets, we implement the resolve function of each entity-set as a binary Naive Bayes classifier that classifies each input feature vector into either of the two classes "duplicate" or "distinct" and returns the probability that the input vector belongs to each class.

2.4 Progressive Entity Resolution

Given a relational dataset $\mathcal{D}$ along with a set of similarity functions F_R and a resolve function $\mathfrak{R}_R$ for each entity-set $R \in \mathcal{D}$ and a resolution budget BG, our goal is to develop a progressive ER approach that produces a high-quality result by using no more than BG units of cost.

It can be easily shown that developing an *optimal* progressive approach is infeasible in practice as it would require an "oracle" that knows in advance the set of pairs whose resolution will have

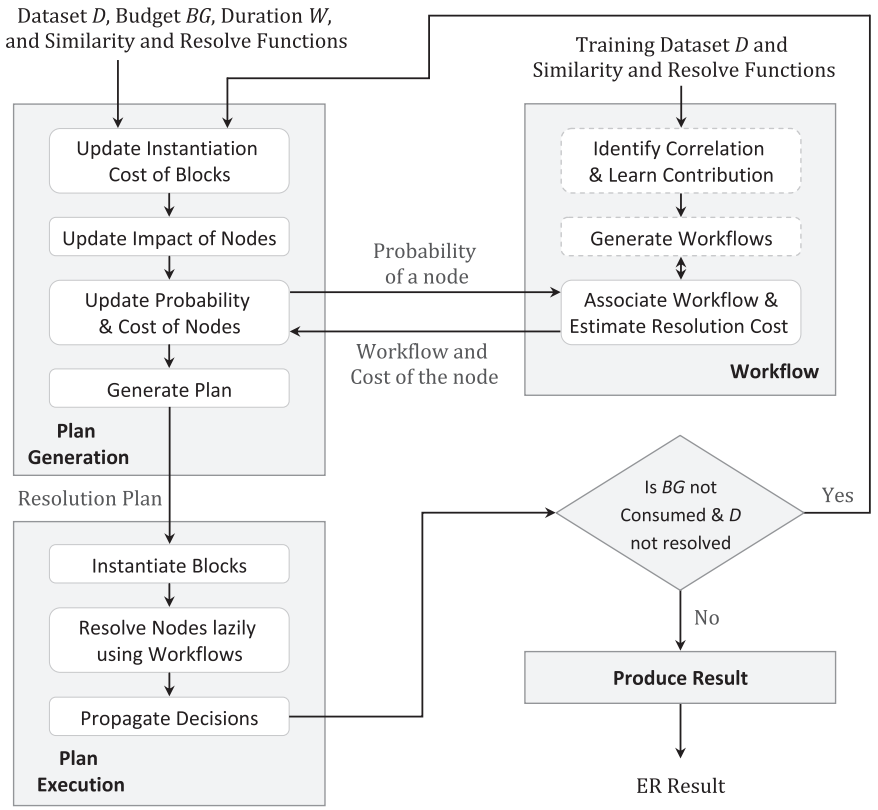

Fig. 3. The overall architecture of ProgressER. The tasks in the two top most components of the workflow module are performed offline before the invocation of the approach.

the highest influence on the quality of the result. Thus, our goal translates into finding a good strategy that best utilizes the given budget by saving cost at two different levels. First, our strategy should resolve only the parts of the dataset that have higher influence on the quality of the result. That is, it should aim to find and resolve as many duplicate pairs as possible, because the more duplicate pairs it correctly identifies, the higher the quality of the result is expected to be.[1] Second, it should resolve those identified pairs with the least amount of cost.

3 OUR APPROACH

In this section, we explain our approach ProgressER. The high-level overview and overall architecture of the approach are presented in Algorithm 1 and Figure 3, respectively. (ProgressER relies on a number of parameters. Appendix A.2 discusses how to set these parameters.)

Partially constructed graph. ProgressER resolves the dataset $\mathcal{D}$ by incrementally constructing the ER graph and resolving its nodes. At any instance of time, ProgressER maintains a partially constructed ER graph $G^p = (V^p, E^p)$, which is a subgraph of the ER graph $G(V, E)$ that corresponds

[1]Similar to Whang et al. (2013b) and Papenbrock et al. (2015), we assume that most of the entity pairs in the dataset are distinct, which is often the case in real-world datasets. Hence, a good strategy for generating a high-quality result given a constraint on the resolution budget should aim to identify the duplicate pairs as quickly as possible, and then declare the remaining pairs to be distinct once the budget is consumed.

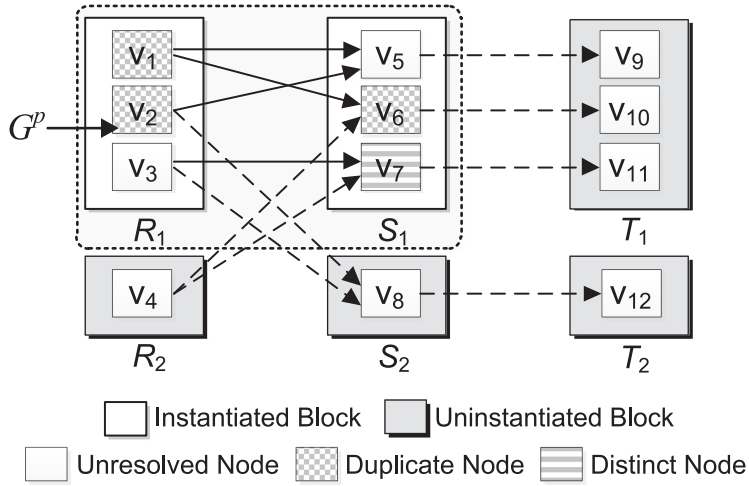

Fig. 4. An example of a graph G and a partially constructed graph G^p.

ALGORITHM 1: Overview of ProgressER.

```
ProgressER(𝒟, BG, W, {F_R, F_S, ...}, {𝔎_R, 𝔎_S, ...})
1   G^p ← {}   // Partial graph (empty initially)
2   c_0 ← 0    // Cost incurred so far
3   while c_0 < BG and isNotResolved(𝒟) do
4     ⟨P̄, c_1⟩ ← Plan-Generation(G^p, 𝒟, W)
5     c_2 ← Plan-Execution(P̄, G^p, {F_R, F_S, ...}, {𝔎_R, 𝔎_S, ...})
6     c_0 ← c_0 + c_1 + c_2
7   return Produce-Result(G^p)
```

to the entire dataset $\mathcal{D}$; i.e., $V^p \subseteq V$ and $E^p \subseteq E$. We refer to the nodes and edges in G^p as *instantiated* nodes and edges of G. The instantiated nodes in G^p are further separated into *resolved* nodes, denoted as **RV**, and *unresolved* nodes, denoted as **UV**, based on whether the approach has already resolved the node (i.e., applied the similarity functions on it and then marked it as duplicate, distinct, or uncertain) or not. Figure 4 depicts an example of a graph G and a partially constructed graph G^p, and Figure 5 shows the different classes of nodes in ProgressER and the relationships among them.

Resolution windows. To incrementally build and resolve the graph G^P, ProgressER divides the total budget BG into several resolution windows. Each *while* iteration can be viewed as a window (Lines 4–6 in Algorithm 1). Each window consists of two phases: plan generation followed by plan execution. In the plan generation phase, ProgressER determines the activities that should be performed during the next W units of cost. The parameter W is the duration of the plan execution phase and it is of the same cost unit as BG. For example, if the unit of cost is the execution time, then BG could be, say, 1 hour and W could be 3 minutes. Generating a plan for a window involves identifying a set of nodes to be resolved during the plan execution phase of that window. This set is chosen from the set of *candidate* nodes $\mathbf{CV} = V - \mathbf{RV}$ (which are the nodes that have not been resolved yet) based on a trade-off between the benefit and cost of resolving these nodes.

Before we explain the plan generation and execution phases, let us first describe how ProgressER incrementally builds the graph G^p. If a node v_i is chosen to be resolved and it is not currently

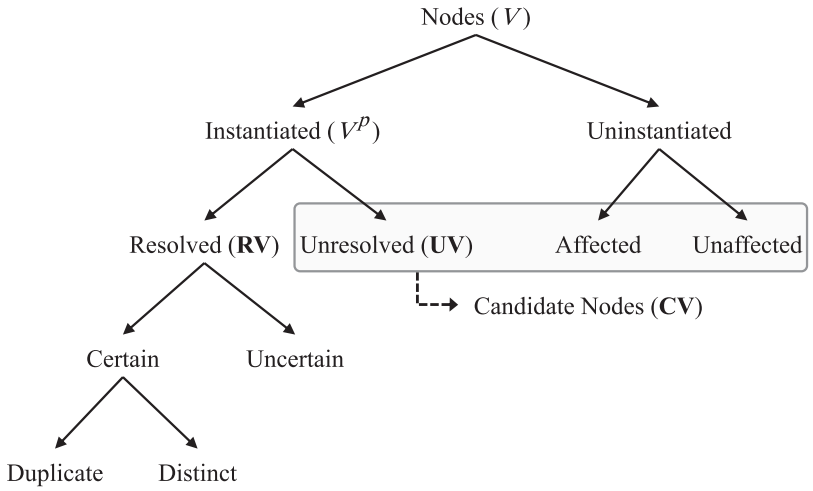

Fig. 5. The relationships among the different classes of nodes.

instantiated in G^p, then it is first instantiated. Instantiation of nodes is performed in unit of *blocks*; that is, if a node v_i needs to be instantiated in a window, then ProgressER instantiates the entire block $BK(v_i)$.

Block instantiation. The process of instantiating a block R_i involves reading all entities of R_i, creating a node for each pair of entities of R_i, and adding edges based on dependency. In addition, it also determines, for each entity in R_i, the set of its dependent entities via each influence $L_{R\to S} \in Inf(R)$ and the blocks to which those dependent entities belong. For example, when instantiating the block A_2 in Table 2, ProgressER determines that entity p_1 depends upon entity a_2 via $L_{A\to P}$, entity p_3 depends upon entity a_4 via $L_{A\to P}$, and entities p_1 and p_3 belong to the block P_1. Such information is essential to create the outgoing edges from the nodes in $V(R_i)$ to their dependent instantiated nodes, and to infer their uninstantiated dependent nodes, which can be useful in determining which nodes to resolve in the next windows. Instantiating at the granularity of a block instead of a node at a time helps bring significant savings since the cost of reading the nodes (possibly from disk/storage) is only incurred once per block. As a result, prior to the beginning of a window, the blocks of $\mathcal{D}$ can be classified as either *instantiated* blocks, which are the blocks that have been instantiated in previous windows, or *uninstantiated* blocks, denoted as **UB**.

Plan generation. The Plan-Generation(.) function (Line 4 in Algorithm 1) generates a resolution plan **P**, and then returns it along with the cost of generating that plan (the variable c_1). A resolution plan $\mathbf{P} := \{\boldsymbol{P}_\mathrm{B}, \boldsymbol{P}_\mathrm{V}\}$ specifies the set of blocks to be instantiated in the window, denoted as $\boldsymbol{P}_\mathrm{B}$, and the set of nodes to be resolved in the window, denoted as $\boldsymbol{P}_\mathrm{V}$. A plan **P** is said to be *valid* only if, for every uninstantiated node $v_i \in \mathbf{P}_\mathrm{V}$, the block $BK(v_i) \in \mathbf{P}_\mathrm{B}$. For example, suppose that the graph G^p at the beginning of the current window is as depicted in Figure 4. If $\mathbf{P}_\mathrm{V} = \{v_3, v_8\}$ and $\mathbf{P}_\mathrm{B}$ does not contain S_2, then the generated plan is not valid as the resolution of v_8 will not be applicable without instantiating S_2.

Each possible resolution plan is associated with a notion of cost and benefit. The cost of a plan **P** is the summation of the *instantiation* cost of every block in $\mathbf{P}_\mathrm{B}$ and the *resolution* cost of every node in $\mathbf{P}_\mathrm{V}$. To estimate the resolution cost of a node v_i, ProgressER first associates a *workflow* with that node, which specifies how v_i is to be resolved (explained next). The benefit of a plan **P** measures the value of resolving the nodes in $\mathbf{P}_\mathrm{V}$. Note that the action of instantiating a block

$R_i \in \mathbf{P}_\mathrm{B}$ by itself does not provide any benefit to the resolution process, but it may be required to ensure the validity of the plan. The benefit and cost models are presented in Section 4.

The process of generating a plan starts by ensuring that every block in **UB** is associated with an updated instantiation cost value and every node in **CV** is associated with updated resolution cost and benefit values. Then, it enumerates a small set of alternative valid plans whose estimated cost is less than or equal to W, and chooses the plan with the highest estimated benefit. Note that the process of generating a plan itself consumes some cost from the budget BG. Thus, a key challenge is to generate a beneficial plan at a small cost. The process of generating a valid resolution plan is explained in Section 5.

Plan execution. The PLAN-EXECUTION(.) function (Line 5 in Algorithm 1) executes the generated plan and then returns the actual execution cost. Note that the actual cost of executing a plan (the variable c_2) might be different from its estimated cost since ProgressER can never accurately determine the *exact* cost of instantiating a block or resolving a node.

The process of executing a plan $\mathbf{P}$ starts by instantiating the blocks $\mathbf{P}_B$. It then iterates over all the nodes in the set $\mathbf{P}_V$ to resolve them. One naive strategy for resolving a node $v_i \in V(R)$ would be to apply all the similarity functions of R on that node and then call the resolve function $\mathfrak{R}_R$ on it to determine its resolution decision. However, it is often sufficient, in practice, to apply a subset of these functions to resolve the node to a `certain` decision. Thus, ProgressER follows a *lazy* resolution strategy that delays the application of a similarity function on a node until it is needed, resulting in a significant cost saving.

To resolve a node $v_i \in V(R)$ using this strategy, ProgressER applies the similarity functions of R on v_i in a specific order. Such an order is referred to as the *workflow* of v_i. After the application of each similarity function, ProgressER calls the resolve function $\mathfrak{R}_R$ on v_i to determine if it is `certain` or not. If the node is `uncertain`, it applies the next similarity function in the workflow on v_i and then calls $\mathfrak{R}_R$ on the node again. This process continues until the node is `certain` or there are no more similarity functions to apply on the node.

Example 3.1. Suppose that the workflow associated with node v_3 in Figure 2 is $f_1^P \rightarrow f_3^P \rightarrow f_2^P$. To resolve v_3 using the lazy resolution strategy, ProgressER first applies f_1^P on it and then calls the resolve function $\mathfrak{R}_P$. If the returned value of either sim_3 or dis_3 is 1, then v_3 is marked as `certain` and the resolution of the node stops. Otherwise, ProgressER applies f_3^P on the node and then calls $\mathfrak{R}_P$ to determine if it is `certain` or not. If not, ProgressER finally applies f_2^P on v_3, calls $\mathfrak{R}_P$, and then marks v_3 with either `duplicate`, `distinct`, or `uncertain` based on the outcome of $\mathfrak{R}_P$.

Given this lazy resolution strategy, each node $v_i \in V(R)$ should be associated with the workflow that causes it to be resolved to a `certain` decision using with the least amount of cost. Generating such a workflow should be based on a tradeoff between the cost of the similarity functions of R and their contribution to the resolution decision of a node. The details of how workflows are generated and associated with nodes are presented in Section 6.

After resolving all nodes in $\mathbf{P}_V$, their resolution decisions are propagated to their instantiated dependent nodes that are still `uncertain`. To do so, ProgressER stores those dependent nodes into a set H, and then iterates over them. For each node $v_j \in H$, ProgressER calls the corresponding resolve function on v_j and then inserts its uncertain dependent nodes into H to further propagate the decision of that node. To ensure that the propagation process terminates, ProgressER inserts those dependent nodes into H only if v_j is resolved to a certain decision or if the increase in sim_j or dis_j due to the last application of the resolve function on v_j exceeds a small constant. This propagation methodology is borrowed from Dong et al. (2005).

Note that to improve the efficiency of the ER process, ProgressER does not propagate the resolution decision of a node v_i to v_j (where v_j is a dependent node of v_i) if v_j is marked as `certain`, i.e., it does not call the resolve function on v_j again to update its decision. This implies that a mistake in marking a node as `certain` will not be corrected as the execution proceeds forward, requiring each resolve function to be implemented carefully so that it does not return $sim_k = 1$ (resp. $dis_k = 1$), when called on a node v_k, unless it is highly confident that v_k is `duplicate` (resp. `distinct`).

Early termination. ProgressER employs resolution windows until the specified budget *BG* is consumed. Then, the PRODUCE-RESULT(.) function (Line 7 in Algorithm 1) outputs the ER result of resolving $\mathcal{D}$. To produce such a result, this function needs to quickly make a decision on the outcome of the uncertain and unresolved nodes in *G*. Although there exist several sophisticated approaches to inferring the resolution decisions of those nodes from the already certain resolved nodes, such approaches would add to the overall computational complexity. Therefore, this function simply declares those nodes to be `distinct`. Since ProgressER finds duplicate entities early, this strategy is expected to perform well.

4 BENEFIT AND COST MODELS

In this section, we define and discuss how we estimate the benefit and cost of a resolution plan.

4.1 Benefit Model

We will first describe how ProgressER computes the benefit of a node v_i and then show how it estimates the benefit of a plan.

In relational ER, the resolution decision of a node v_i depends upon the resolution decisions of other nodes in the graph *G*. Thus, the probability of v_i to be duplicate at any instance of time can be inferred from the *state* of the graph *G* at that time, which we denote as $\boldsymbol{\mathcal{S}}$ and define as the set of nodes in the graph that are marked as duplicate. Given that the goal of ProgressER is to resolve as many duplicate nodes in *G* as possible, the benefit of a node v_i can be determined based on whether v_i is duplicate or not (*direct benefit*) and on how useful declaring v_i to be duplicate is in identifying more duplicate nodes in *G* (*indirect benefit* or the *impact* of v_i). More formally, the benefit of a node v_i, which is a unitless value, is defined as follows:

$$Benefit(v_i) = \mathcal{P}(v_i|\boldsymbol{\mathcal{S}}) + \mathcal{P}(v_i|\boldsymbol{\mathcal{S}}) * Imp(v_i), \tag{1}$$

where $\mathcal{P}(v_i|\boldsymbol{\mathcal{S}})$ is the probability that the node v_i is duplicate given the current state of the graph *G*, and $Imp(v_i)$ is the impact of v_i, which is defined as follows:

$$Imp(v_i) = \sum_{v_j \in Top_{v_i}} \mathcal{P}(v_j|\boldsymbol{\mathcal{S}}_{v_i}) - \sum_{v_k \in Top} \mathcal{P}(v_k|\boldsymbol{\mathcal{S}}), \tag{2}$$

where $\boldsymbol{\mathcal{S}}_{v_i}$ is the state of the current graph after declaring v_i to be duplicate, *Top* is a set of unresolved nodes that can be resolved within the remaining cost of our budget *BG* (i.e., the total of their resolution costs and the instantiation costs of their blocks, if needed, is less than or equal to the remaining budget) such that the summation of their probability values given $\boldsymbol{\mathcal{S}}$ is maximized, and Top_{v_i} is a set of unresolved nodes that can be resolved within the remaining cost of our budget minus the resolution cost of v_i such that the summation of their probability values given $\boldsymbol{\mathcal{S}}_{v_i}$ is maximized. For simplicity, we will denote the value $\mathcal{P}(v_i|\boldsymbol{\mathcal{S}})$ as $\mathcal{P}(v_i)$ henceforth.

Computing the exact benefit of a node v_i is infeasible in practice for two reasons. First, computing the probability value of a node given a state requires the graph *G* to be fully instantiated to infer the dependency among the nodes (in contrast, we construct *G* progressively), and it is also equivalent to the *belief update* problem in Bayesian Networks, which is known to be NP-hard in

general (Cooper 1990). Second, identifying each of the *Top* and Top_{v_i} sets can be easily proven to be NP-hard as it contains the traditional *knapsack* problem as a special case.[2] Therefore, we use heuristics to estimate the values of $\mathcal{P}(v_i)$ and $Imp(v_i)$.

4.1.1 Probability Estimation. We estimate the value of $\mathcal{P}(v_i)$ using the Noisy-Or model (Pearl 1988), which models the interaction among several causes on a common effect. We model the node v_i being duplicate as an *effect* y_i that can be produced by any members of a set of binary heterogeneous *causes* $\mathbf{X}^i = \{x_1^i, x_2^i, \ldots, x_n^i\}$. In our case, these causes correspond to the direct influencing nodes of v_i and the block $BK(v_i)$. That is, the value $\mathcal{P}(v_i)$ is estimated based on which of v_i's influencing nodes are duplicate and/or on the percentage of duplicate nodes in the block $BK(v_i)$.

Each cause x_j^i can be either *present* or *absent* and has a probability $\mathcal{P}(y_i|x_j^i)$ of being capable of producing the effect when it is present and all other causes in $\mathbf{X}^i$ are absent. The Noisy-Or model assumes that the set of causes $\mathbf{X}^i$ are independent of each other[3] and thus computes the conditional probability $\mathcal{P}(v_i) = \mathcal{P}(y_i|\mathbf{X}^i)$ as follows:

$$\mathcal{P}(v_i) = 1 - (1 - \delta) \prod_{x_j^i \in \mathbf{X}_p^i} \frac{1 - \mathcal{P}(y_i|x_j^i)}{1 - \delta}, \tag{3}$$

where the parameter δ is a leak value (explained later) and $\mathbf{X}_p^i$ is the set of present causes in $\mathbf{X}^i$. If a cause x_j^i corresponds to a node v_k that influences v_i via $L_{S \to R}$, then x_j^i is present only if v_k was resolved to duplicate in a previous window, and thus ProgressER sets $\mathcal{P}(y_i|x_j^i)$ to the value $\mathcal{P}_{S \to R}$. This value refers to the probability that a node $v_l \in V(R)$ is duplicate given that there exists a duplicate node $v_m \in V(S)$ that influences v_l via $L_{S \to R}$ and all other causes of y_l are absent. Such a probability value can be specified by a domain expert or learned from a training dataset. On the other hand, if x_j^i corresponds to the block $BK(v_i) = R_k$, then ProgressER sets $\mathcal{P}(y_i|x_j^i)$ to the value $\frac{|V^+(R_k)|}{|V^*(R_k)|}$, where $V^*(R_k)$ is the set of nodes of R_k that have been resolved in the previous windows, but we require that x_j^i be present only if the fraction of resolved nodes in R_k is at least α, i.e., $|V^*(R_k)| \geq \alpha * |V(R_k)|$, where α is a predefined threshold. Such a requirement is not necessary to ProgressER, but rather helps improve the accuracy of estimating the probability value of nodes.

Example 4.1. Consider the graph in Figure 4. Suppose that $\mathcal{P}_{R \to S} = 0.4$ and $\alpha = 0.3$. Then, $\mathbf{X}_p^5$ consists of three causes that correspond to the nodes v_1 and v_2 and the block S_1, whereas $\mathbf{X}_p^8$ consists of one cause that corresponds to the node v_2. Therefore, $\mathcal{P}(v_5) = 1 - (1 - 0.4) * (1 - 0.4) * (1 - \frac{1}{2}) = 0.82$ and $\mathcal{P}(v_8) = 1 - (1 - 0.4) = 0.4$.

The above model states that if all influencing nodes of v_i are either distinct, uncertain, or unresolved, and the fraction of resolved nodes in $BK(v_i)$ is less than α, then the node v_i cannot be duplicate. To overcome this limitation, the *Leaky* Noisy-Or model (Henrion 1987) allows us to assign a non-zero *leak* probability value δ to $\mathcal{P}(v_i)$. This leak value represents the probability that v_i is duplicate in the absence of all explicitly modeled causes of y_i, and it can be different for different entity-sets. For instance, the leak value δ assigned to nodes of R can be different from that assigned to nodes of S.

[2]The problem of identifying the *Top* set is equivalent to the knapsack problem if all nodes are already instantiated. The remaining budget can be viewed as the knapsack capacity and each node can be treated as an item. A node probability and resolution cost can be respectively viewed as the value and weight of the corresponding item.

[3]One could also use a more advanced model such as the *Recursive* Noisy-Or model (Lemmer and Gossink 2004) to allow for dependent causes to be used in estimating the probability of an effect. However, such dependencies among causes are rarely observed in datasets and would require significant effort to be identified.

4.1.2 Impact Estimation. To estimate the value of $Imp(v_i)$, instead of considering all nodes in *Top* and Top_{v_i}, we restrict the impact computation to a small subset of nodes, denoted as $\mathcal{N}(v_i)$, and hence define the impact of v_i as follows:

$$Imp(v_i) = \sum_{v_j \in \mathcal{N}(v_i)} [\mathcal{P}(v_j|\mathcal{S}_{v_i}) - \mathcal{P}(v_j|\mathcal{S})]. \tag{4}$$

Clearly, the set $\mathcal{N}(v_i)$ should include the nodes that will be affected the most by the resolution of v_i. Thus, we include in $\mathcal{N}(v_i)$ the unresolved nodes that can be reached from v_i by following at most β edges through unresolved nodes only. (The parameter β is explained at the end of this subsection.) For example, suppose that the graph G at the beginning of the current window is as depicted in Figure 4. If $\beta = 2$, then $\mathcal{N}(v_3) = \{v_8, v_{12}\}$. Although node v_{11} is unresolved and is within two hops from v_3, it is not included in $\mathcal{N}(v_3)$ because it cannot be reached from v_3 through unresolved nodes.

Even after restricting the impact computation to the set $\mathcal{N}(v_i)$ only, computing an impact value for every candidate node v_i is still infeasible for two reasons. First, computing $\mathcal{P}(v_j|\mathcal{S}_{v_i})$ is in general an NP-hard problem (Cooper 1990). Although there are approximation algorithms for this problem (Jensen and Nielsen 2007), such algorithms are still expensive for our (near) real-time settings. Second, identifying $\mathcal{N}(v_i)$ might not be possible in certain cases unless the graph G is fully instantiated. For instance, knowing that v_{12} is part of $\mathcal{N}(v_3)$ in the example above requires ProgressER to instantiate v_8 first in order to determine its dependent nodes.

Thus, ProgressER instead computes a single estimated impact value, denoted as $Imp(R)$, for every entity-set R and uses that single value as the impact value of every candidate node $v_i \in V(R)$. This value is computed based on statistics that we collect prior to and throughout the execution of our approach.

To estimate the value of $Imp(R)$, we first define $U(k, L_{R\to S})$ as the function that computes, for k nodes of entity-set R, the estimated number of their unresolved direct dependent nodes via $L_{R\to S}$. That is, this function estimates the number of the direct dependent nodes via $L_{R\to S}$ that will be impacted by the resolution of the k nodes. The output of this function can be computed as follows:

$$U(k, L_{R\to S}) = \left\lceil k * degree(L_{R\to S}) * \left(1 - \frac{|V^*(S)|}{|V(S)|}\right)\right\rceil, \tag{5}$$

where $degree(L_{R\to S})$ is the average number of direct dependent nodes that a node $v_i \in V(R)$ can have via $L_{R\to S}$. Such a value can be initially set by a domain expert or learned from a training dataset, and can then be adjusted as the approach proceeds forward.

Now, the impact value $Imp(R)$ can be estimated by calling the Compute-Impact(.) function in Algorithm 2 with these values (R, β).

Example 4.2. Let $\mathcal{D}$ correspond to the dataset whose ER graph is depicted in Figure 4. Suppose that $L(\mathcal{D}) = \{L_{R\to S}, L_{S\to T}\}$. Let us further assume that $\mathcal{P}_{R\to S} = 0.4$ and $\mathcal{P}_{S\to T} = 0.5$, and that $degree(L_{R\to S}) = 2$ and $degree(L_{S\to T}) = 1$. Assume we want to compute the value of $Imp(R)$ and that the value of β is 2. In this case, the value of $U(1, L_{R\to S}) = 1 * 2 * (1 - \frac{2}{4}) = 1$, and the value of $U(1, L_{S\to T}) = 1 * 1 * (1 - \frac{0}{4}) = 1$. Thus, the value of $Imp(R) = \mathcal{P}_{R\to S} * U(1, L_{R\to S}) + \mathcal{P}_{R\to S} * \mathcal{P}_{S\to T} * U(1, L_{S\to T}) = 0.4 * 1 + 0.4 * 0.5 * 1 = 0.6$.

Using this model allows the impact computation to be performed very efficiently (as ProgressER needs to compute only a single impact value for each entity-set $R \in \mathcal{D}$) and works well as we show in Section 8.

We now explain how ProgressER sets the value of the parameter β. At any instance of time, ProgressER uses the same value for β for all entity-sets, but it updates that value throughout the

ALGORITHM 2: Impact Computation.

```
COMPUTE-IMPACT(R, β)
1   if β = 0 then
2     return 0
3   imp ← 0
4   Visited ← {R}
5   foreach L_{R→S} ∈ Inf(R) do
6     k ← U(1, L_{R→S}) // Equation (5)
7     ⟨m, Visited⟩ ← COMPUTE-PROB-INC(R, S, k, Visited, β)
8     imp ← imp + m
9   return imp

COMPUTE-PROB-INC(R, S, k, Visited, β)
1   Visited ← Visited ∪ {S}
2   n ← P_{R→S} * k
3   if β − 1 = 0 then
4     return ⟨n, Visited⟩
5   foreach L_{S→T} ∈ Inf(S) s.t. T ∉ Visited do
6     k ← U(k, L_{S→T}) // Equation (5)
7     ⟨m, Visited⟩ ← COMPUTE-PROB-INC(S, T, k, Visited, β − 1)
8     n ← n + P_{R→S} * m
9   return ⟨n, Visited⟩
```

execution. Intuitively, β should be continually reduced to limit the impact computation to only the dependent nodes that have a high chance of being resolved in the remaining budget RM. Thus, ProgressER updates the value of β at the beginning of each window to $\lfloor \gamma * \frac{RM}{BG} + 0.5 \rfloor$, where γ is the number of edges in the longest non-cyclic path between any two nodes in the *influence* graph whose nodes correspond to the entity-sets in $\mathcal{D}$ and whose direct edges correspond to the influences $L(\mathcal{D})$. For example, the influence graph of the dataset whose ER graph is depicted in Figure 4 consists of $|\mathcal{D}| = 3$ nodes (that correspond to the entity-sets R, S, and T) and two edges ($R \rightarrow S$ and $S \rightarrow T$), and thus $\gamma = 2$. Decreasing the value of β will cause the impact computation of a node $v_i \in V(R)$ to be restricted to only the dependent nodes of the entity-sets that are close to R in the influence graph, which are the nodes that will be affected the most by the resolution of v_i and thus have a high chance of being resolved later.

4.1.3 Plan Benefit. We measure the benefit of a plan $\mathbf{P}$ as the summation of the benefit of resolving each node in $\mathbf{P}_V$[4]:

$$Benefit(\mathbf{P}) = \sum_{v_i \in \mathbf{P}_V} Benefit(v_i). \tag{6}$$

4.2 Cost Model

The cost of a resolution plan $\mathbf{P}$ is computed as follows:

$$Cost(\mathbf{P}) = \sum_{R_i \in \mathbf{P}_B} C^{ins}(R_i) + \sum_{v_j \in \mathbf{P}_V} C^{res}(v_j), \tag{7}$$

[4]A more accurate model of estimating the benefit of a *group* of nodes can be developed, but that would significantly add to the complexity of the plan generation process.

ALGORITHM 3: Updating the Impact, Probability, and Cost Values of Nodes.

```
PERFORM-ANALYSIS(UV, UB, 𝒟, L(𝒟), β)
1   UPDATE-INST-COST(UB)
2   UPDATE-IMPACT(𝒟, L(𝒟), β)
3   UPDATE-PROB-AND-COST(UV)
4   AV ← GET-AFFECTED-NODES(UB)
5   UPDATE-PROB-AND-COST(AV)
6   for i ← 1 to |UB| do
7     COMPUTE-SINGLE-PROB-AND-COST(UB[i])
```

where $C^{ins}(R_i)$ is the cost of instantiating the block R_i, and $C^{res}(v_j)$ is the cost of resolving the node v_j. The cost $C^{ins}(R_i)$ typically consists of the cost of reading the block R_i plus the cost of creating nodes for the pairs of entities of R_i. The instantiation cost depends upon where the blocks are stored. In practice, blocks could be stored on any local or remote storage device. In Section 7, we provide a model for estimating the instantiation cost of blocks that are stored on a local machine's disk.

The resolution cost $C^{res}(v_j)$ depends upon the order in which the similarity functions of R are applied on the node v_j; i.e., the workflow of v_j. Associating a workflow with a node depends upon the probability of that node to be duplicate. Thus, to estimate the resolution cost of a node v_j, ProgressER first estimates the value of $\mathcal{P}(v_j)$, uses that value to associate a workflow with v_j, and then estimates the value of $C^{res}(v_j)$ based on the associated workflow. The details of how the resolution cost of a node is estimated given its associated workflow are presented in Section 6.

5 PLAN GENERATION

In order to generate a resolution plan in a window, ProgressER first needs to perform a benefit-vs-cost analysis on the uninstantiated blocks **UB** and candidate nodes **CV** to ensure that every block in **UB** is associated with an updated instantiation cost value and every node in **CV** is associated with updated resolution cost and benefit values. This analysis step is discussed in details in Section 5.1. Given the benefit and cost values of the candidate nodes, and the instantiation cost values of the uninstantiated blocks, ProgressER needs to generate a *valid* resolution plan **P** such that $Cost(\mathbf{P}) \leq W$ and $Benefit(\mathbf{P})$ is maximized. Generating such a plan can be proven to be NP-hard (Burg et al. 1999). Thus, we propose in Section 5.2 an efficient approximate solution that performs well in practice.

5.1 Benefit-vs-Cost Analysis

Algorithm 3 describes how the benefit-vs-cost analysis is performed.

Instantiation cost. The algorithm starts by passing the blocks **UB** to the function UPDATE-INST-COST(.) to update their instantiation cost values. This function needs to compute the instantiation cost of every block in **UB** at the beginning of the first window. These cost values do not need to be updated in any subsequent window unless the blocks can overlap. More details on when this function should update these values can be found in Appendix A.1.

Impact. The algorithm then calls the UPDATE-IMPACT(.) function to compute a single impact value for each $R \in \mathcal{D}$ using the COMPUTE-IMPACT(.) function in Algorithm 2.

Probability and cost. Next, the algorithm passes the set of instantiated unresolved nodes **UV** to the function UPDATE-PROB-AND-COST(.), which updates the probability and resolution cost values of only the nodes whose values may have changed due to the resolution in the previous window. That

is, the probability value of a node $v_i \in \mathbf{UV}$ is updated only if one of the following two conditions holds. First, at least one of its influencing nodes has been resolved to duplicate in the previous window. Second, some of the nodes resolved in the previous window belong to the block $BK(v_i)$ and $BK(v_i)$ is a present cause in estimating the probability of v_i. Once the probability value of a node $v_i \in \mathbf{UV}$ is updated, this function uses the new probability to reassociate a workflow with v_i and then recomputes the cost of v_i based on the new workflow.

Before we describe how the algorithm updates the probability and cost values of the uninstantiated nodes, we classify those nodes into *affected* and *unaffected* nodes. An uninstantiated node is said to be *affected* if it has at least one duplicate influencing instantiated node. For instance, the node v_8 in Figure 4 is affected because v_2 influences v_8 via $L_{R \to S}$ and v_2 was resolved to duplicate in a previous window. To update the probability and cost values of the affected uninstantiated nodes, the algorithm passes the set of blocks **UB** to the Get-Affected-Nodes(.) function to retrieve those nodes and then passes them to the same Update-Prob-And-Cost(.) function. To compute/update the probability of affected uninstantiated nodes, ProgressER maintains a set of *counter* values for each affected uninstantiated node $v_i \in V(R)$. Each counter value corresponds to an influence $L_{S \to R} \in Dep(R)$, and it denotes the number of duplicate instantiated nodes influencing v_i via $L_{S \to R}$. In Figure 4, the set of counter values for v_8 consists of a single counter that corresponds to the influence $L_{R \to S}$ and its value is 1.

Since the set of present causes of any unaffected uninstantiated node cannot contain any influencing node (according to the definition of the unaffected nodes), all unaffected uninstantiated nodes of a block in **UB** have the same probability value. Thus, the algorithm iterates over the blocks in **UB** and, for each block R_i, assigns a single probability value (the leak value) for all unaffected uninstantiated nodes of R_i. It then computes a single resolution cost for them. If the leak value is not to be updated throughout the execution of ProgressER, the algorithm will call the Compute-Single-Prob-And-Cost(.) function at the beginning of the first window only. It is important to note that, except for their count, a probability and a cost value, ProgressER does not store any information about the unaffected uninstantiated nodes of block $R_i \in \mathbf{UB}$.

5.2 Algorithm

Algorithm 4 describes our solution for the problem of generating a valid resolution plan in a window. The algorithm initially calls the Select-Nodes(.) function (Line 4) to compute the maximum benefit that can be obtained by considering only the instantiated unresolved nodes **UV**. The input to this function is a set of nodes, a cost budget, and the dataset $\mathcal{D}$ (the impact values are associated with the entity-sets of $\mathcal{D}$). This function first identifies a subset of the input nodes whose total resolution cost is less than or equal to the input budget and whose total benefit is as large as possible, and then returns this subset of nodes along with their total benefit. The implementation of this function is discussed later.

The algorithm then sorts the blocks in **UB** in a non-increasing order based on the benefit that they are expected to add to the current window once they are instantiated (Line 5). To sort these blocks, the Sort-Blocks(.) function first computes, for each block $R_i \in \mathbf{UB}$, a usefulness value as follows:

$$\mathcal{U}(R_i) = \frac{\sum_{v_j \in V(R_i)} Benefit(v_j)}{C^{ins}(R_i) + \sum_{v_j \in V(R_i)} C^{res}(v_j)}. \tag{8}$$

Then, the function sorts the blocks in a non-increasing order based on their usefulness values and then returns the sorted list of blocks.

Next, the algorithm iterates over the sorted list of blocks, and for each block, checks if the total benefit that can be obtained in the window will increase if that block is instantiated (Lines 8 and

ALGORITHM 4: Generating a Valid Resolution Plan.

```
GENERATE-PLAN(UV, UB, W, 𝒟)
1   BlockSet P_B ← ∅
2   NodeSet P_V ← ∅, M ← ∅
3   Double max, t
4   ⟨P_V, max⟩ ← SELECT-NODES(UV, W, 𝒟)
5   BlockList List ← SORT-BLOCKS(UB, 𝒟)
6   for i ← 1 to |List| do
7     Block B ← List[i]
8     ⟨M, t⟩ ← SELECT-NODES(P_V ∪ V(B), W − C^ins(B), 𝒟)
9     if t > max then
10      P_B ← P_B ∪ {B}
11      P_V ← M
12      max ← t
13      W ← W − C^ins(B)
14    else
15      break
16  return ⟨P_B, P_V⟩
```

9). If so, the algorithm adds the block to the set of blocks $\boldsymbol{P}_{\mathrm{B}}$, and updates the values of the set $\boldsymbol{P}_{\mathrm{V}}$ and the other variables (Lines 10–13). Otherwise, it exits the **while** loop. Finally, the algorithm returns the sets $\boldsymbol{P}_{\mathrm{B}}$ and $\boldsymbol{P}_{\mathrm{V}}$ (Line 16).[5]

Select nodes. The SELECT-NODES(.) function chooses a subset of nodes from the input set such that their total benefit is maximized and their total cost is less than or equal to the input budget. In general, the benefit of some nodes in the input set might be dependent upon whether some other nodes in the input set have been added to the output set of nodes or not. For example, consider the graph in Figure 4. For simplicity, suppose that the input set to this function consists of all candidate nodes. Let us further assume that the impact of each candidate node is computed using Equation (4) and that the node $v_8 \in \mathcal{N}(v_3)$. This means that the node v_8 was used in computing the current impact value of v_3, and thus the impact of v_3 needs to be updated once v_8 is added to the output set of nodes.

Accounting for such dependencies among the input nodes is infeasible in practice. To illustrate, suppose that the SELECT-NODES(.) function in the example above added v_{12} to the output set. Since v_{12} belongs to an uninstantiated block and none of its influencing nodes is instantiated, determining which nodes to update as a result of this addition might not be applicable unless the sets of influencing nodes of all nodes in G are known. Even if such information is available, accounting for the dependency among the nodes might be unnecessary since the output set of nodes usually constitutes a small percentage of the nodes in V, and hence the likelihood that they will be dependent upon each other is in general low. Such a likelihood will even decrease as the value of β decreases until it becomes zero at the later stages of the execution when the benefits of nodes are restricted to their direct benefit, i.e., their probability values.

[5]Our algorithm assumes that any block R_i can be instantiated completely within a window, i.e., $C^{ins}(R_i) \leq W$. This assumption, however, is not germane to our approach. That is, we could make a slight modification to our algorithm so that the process of instantiating a block can be split over two or more windows. For instance, if a block R_i is chosen to be instantiated in window, we instantiate in the window only the nodes of R_i that are chosen to be resolved in the window. The other nodes of R_i can be instantiated in other subsequent window(s).

Therefore, we do not account for such dependencies among the nodes when determining the output set of nodes of the Select-Nodes(.) function, viewing the problem as a traditional knapsack problem. Hence, we use the greedy algorithm that first sorts the nodes in the input set in a non-increasing order based on their benefit per cost unit, and then, starting from the head of the sorted list, it proceeds to insert the nodes into the output set until the budget is consumed. This greedy algorithm works very well in cases where the items' weights (i.e., nodes' resolution costs) are very small relative to the knapsack size.

Initialization step. In the initial few resolution windows, ProgressER might not have adequate knowledge about which nodes tend to be duplicate and which blocks contain a high number of duplicate nodes. To obtain such knowledge requires ProgressER to explore several blocks in the initial few windows. To address this requirement, ProgressER needs to employ a different strategy for generating a resolution plan. Using Algorithm 4 in such cases might result in instantiating a lesser number of blocks than desired, and thus unnecessarily resolving a large number of nodes of these blocks. To illustrate, suppose that the first two blocks in the sorted list returned from the Sort-Blocks(.) function at the beginning of the first window belong to the same entity-set, i.e., all nodes of these two blocks have the same benefit per cost value. In this case, Algorithm 4 will not consider instantiating the second block unless the budget W is sufficient for instantiating and resolving *all* nodes of the first block. However, ProgressER should instantiate several blocks in each of the initial few windows and explore those blocks by resolving a few nodes from each of them.

To this end, ProgressER uses a different plan generation algorithm in each of these initial windows. This algorithm is a modification of Algorithm 4. It first starts by sorting the blocks in **UB** using the same Sort-Blocks(.) function. Then, it iterates over the blocks in the sorted list, starting from the head of the list. For each block R_i, the algorithm checks if the instantiation cost of this block plus the cost of resolving k randomly chosen nodes of R_i is less than or equal to the remaining budget of the current window, where $k = \lceil \alpha * |V(R_i)| \rceil$ and the value α is the threshold described in Section 4.1. If so, the algorithm inserts R_i and the k nodes into the sets $\mathbf{P}_\mathrm{B}$ and $\mathbf{P}_\mathrm{V}$, respectively, and then updates the budget accordingly, i.e., $W = W - C^{ins}(R_i) - C^r(R_i, k)$, where $C^r(R_i, k)$ is the cost of resolving the k nodes. Otherwise, the algorithm considers the next block in the sorted list and performs the same steps on it. This process continues until the budget W is consumed or we have iterated over all blocks in the sorted list. Finally, if W is not fully consumed, the algorithm randomly chooses extra nodes from the blocks in $\mathbf{P}_\mathrm{B}$ and adds them to the set $\mathbf{P}_\mathrm{V}$ to fill the budget W. This algorithm is employed in the initial few windows and hence can be viewed as an initialization step of ProgressER.

6 WORKFLOWS

Given $|F_R|$ similarity functions associated with entity-set R, there are $|F_R|!$ different possible orders of function application that could be employed to resolve a node $v_i \in V(R)$. ProgressER, however, should apply these functions in the order that leads to a certain resolution decision with the least amount of cost. To generate such an order, we need to differentiate between these functions in terms of their costs and their contributions to the resolution decision of a node. In Section 6.1, we define our concept of the contribution of similarity functions, and then describe in Section 6.2 how ProgressER generates a workflow for a node. In Section 6.3, we discuss how ProgressER deals with correlation among similarity functions. Next, we present in Section 6.4 an algorithm that learns the contribution values from a training dataset. In Section 6.5, we describe how workflows are associated with nodes, and finally show in Section 6.6 how ProgressER estimates the resolution cost of a node given its associated workflow.

6.1 Contribution of Similarity Functions

In order to resolve a node v_i, ProgressER needs to obtain sufficient positive or negative evidence. Each similarity function f_j^R, when applied on a node v_i, provides positive and/or negative evidence to the resolution of v_i. Evidence is considered positive (resp. negative) if it increases the chance that the resolve function will return 1 as the similarity (resp. dissimilarity) confidence of v_i. The amounts of the positive and negative evidence of f_j^R are measured by the resolve function $\mathfrak{R}_R$ when it is applied later on v_i.

Similarity functions differ from each other in the amount of evidence that they provide to the resolution decision. Hence, in order to generate a workflow for a node $v_i \in V(R)$, we need to estimate the amount of positive and negative evidence that the similarity functions in F_R would provide *w.r.t.* the resolve function $\mathfrak{R}_R$. Thus, we define for each function f_j^R, a *positive* contribution $t_j^{R+} \in [0, 1]$, and a *negative* contribution $t_j^{R-} \in [0, 1]$. The positive (resp. negative) contribution of a function f_j^R is the amount of positive (resp. negative) evidence that f_j^R is expected to provide when it is applied on a *duplicate* (resp. *distinct*) node. More formally, t_j^{R+} (resp. t_j^{R-}) is defined as the average similarity (resp. dissimilarity) confidence value sim_k (resp. dis_k) that would be obtained from the resolve function $\mathfrak{R}_R$ after calling it on a *duplicate* (resp. *distinct*) node v_k on which only the function f_j^R has been applied.

Example 6.1. Consider functions f_2^P and f_3^P in Table 3 that are defined respectively on the Abstract and Keywords attributes of the Papers entity-set. If the resolve function $\mathfrak{R}_P$ accounts for the fact that two distinct papers might have similar sets of keywords (e.g., the pair $\langle p_2, p_4 \rangle$ in Table 2), then the value of t_2^{P+} should be greater than that of t_3^{P+}. To illustrate, suppose that node v_i represents a duplicate pair of paper entities, and that the outputs of applying f_2^P and f_3^P on v_i are both 1.0. In this case, the similarity confidence value sim_i that would be obtained from $\mathfrak{R}_P$ after calling it with these parameters $(*, 1.0, *, \{*, *\}, \{*\})$ would be higher than the value sim_i that would be obtained from $\mathfrak{R}_P$ after calling it with these parameters $(*, *, 1.0, \{*, *\}, \{*\})$, where the symbol $*$ represents a missing value. This is because $\mathfrak{R}_P$ knows that having similar sets of keywords does not necessarily imply that the node is duplicate.

Example 6.2. Consider functions f_1^A and f_2^A in Table 3 that are defined respectively on the Name and Email attributes of the Authors entity-set. If the resolve function $\mathfrak{R}_A$ accounts for the fact that one person might have different email addresses (e.g., the pair $\langle a_1, a_3 \rangle$ in Table 2), then the value of t_1^{A-} should be greater than that of t_2^{A-}. To illustrate, suppose that node v_i represents a distinct pair of author entities, and that the outputs of applying f_1^A and f_2^A on v_i are both 0. In this case, the dissimilarity confidence value dis_i that would be obtained from $\mathfrak{R}_A$ after calling it with these parameters $(0, *, \{*, *\})$ would be higher than the value dis_i that would be obtained from $\mathfrak{R}_A$ after calling it with these parameters $(*, 0, \{*, *\})$. This is because $\mathfrak{R}_A$ knows that having different email addresses does not necessarily imply that the node is distinct.

As illustrated in Figure 3, the contribution values of functions are predetermined prior to the invocation of ProgressER. We propose in Section 6.4 an algorithm that learns those values from a training dataset.

6.2 Workflow Generation

The process of generating a workflow for a node $v_i \in V(R)$ proceeds as follows. First, ProgressER computes for each similarity function $f_j^R \in F_R$ a utility value as follows:

$$[\theta * t_j^{R+} + (1 - \theta) * t_j^{R-}]/c_j^R, \tag{9}$$

Table 4. Contribution and Cost Values of Similarity Functions

Function	Positive contribution	Negative contribution	Cost
f_1^R	0.7	0.5	1
f_2^R	0.2	0.7	1
f_3^R	0.6	0.4	1
f_4^R	0.3	0.4	1

where $\theta \in [0, 1]$ is a greediness parameter that controls which type of function contribution, positive or negative, to use when generating the workflow. The higher the value of θ is, the more ProgressER relies on the positive contribution of the functions. Hence, ProgressER sets the value of θ to the probability of the node $\mathcal{P}(v_i)$.

Then, ProgressER sorts the functions in F_R in a non-increasing order based on their utility values. This order of functions is the workflow of the node v_i. Such a workflow is expected to maximize the chance of resolving v_i to a `certain` decision with the least amount of cost since the functions with highest contribution per unit cost of resolution will be applied first.

Example 6.3. Consider the four similarity functions f_1^R, f_2^R, f_3^R, and f_4^R shown in Table 4. When generating a workflow for a node v_i whose probability value $\mathcal{P}(v_i)$ is 0.7, the utility values of those functions will be $\frac{0.7*0.7+0.3*0.5}{1} = 0.64$, $\frac{0.7*0.2+0.3*0.7}{1} = 0.35$, $\frac{0.7*0.6+0.3*0.4}{1} = 0.54$, and $\frac{0.7*0.3+0.3*0.4}{1} = 0.33$, respectively. Thus, the workflow will be $f_1^R \rightarrow f_3^R \rightarrow f_2^R \rightarrow f_4^R$.

6.3 Correlated Similarity Functions

We define the concept of correlated similarity functions as follows. Two functions f_i^R and f_j^R are *correlated iff* there exists an attribute $R.a_k \in \mathcal{A}(f_i^R)$ and an attribute $R.a_l \in \mathcal{A}(f_j^R)$ such that $R.a_k$ and $R.a_l$ are correlated.[6]

For two correlated functions f_i^R and f_j^R, applying f_i^R on a node, on which f_j^R has been already applied, may not provide the expected positive/negative evidence of f_i^R. Hence, such correlation needs to be considered when generating workflows.

To this end, for each two correlated functions f_i^R and f_j^R, in addition to computing their contribution values, we compute these conditional positive and negative contributions: $t_{i|j}^{R+}$, $t_{j|i}^{R+}$, $t_{i|j}^{R-}$, $t_{j|i}^{R-}$. The value of $t_{j|i}^{R+}$ (resp. $t_{j|i}^{R-}$) is the amount of positive (resp. negative) evidence that f_j^R is expected to provide when applied on a pair of duplicate (resp. distinct) entities, on which f_i^R has already been applied. Note we assume that $t_{i|j}^{R+} = t_i^{R+}$, and that $t_{i|j}^{R-} = t_i^{R-}$ if f_i^R and f_j^R are not correlated. This assumption, however, is not germane to ProgressER, but rather helps reduce the number of contribution values that need to be computed for the similarity functions of R.

In Example 6.3, suppose that functions f_1^R and f_3^R are correlated with each other. Now, when generating a workflow for v_i whose $\mathcal{P}(v_i)$ is 0.7, ProgressER first chooses the function that has the highest utility value, which is f_1^R. For the second function in the workflow, it chooses the function whose has the highest utility value given that f_1^R has already been added to the workflow. Such a conditional utility value of a function f_j^R is computed as follows:

$$[\theta * t_{j|1}^{R+} + (1 - \theta) * t_{j|1}^{R-}]/c_j^R. \tag{10}$$

[6]Correlation among attributes is often identified using approaches such as Pearson's Correlation Coefficient (Reynolds and Reynolds 1977) and KL divergence (Kullback 1997).

ALGORITHM 5: Generating a Resolution Workflow.

```
GENERATE-WORKFLOW(F_R, θ)
1    Function f, h
2    Double pos, neg, value, max
3    for i ← 1 to |F_R| do
4      max ← −1
5      for j ← 1 to F_R.size() do
6        f ← F_R[j]
7        pos ← GET-POS-CONTRIBUTION(f, W)
8        neg ← GET-NEG-CONTRIBUTION(f, W)
9        value ← (θ * pos + (1 − θ) * neg))/ GET-COST(f)
10       if value > max then
11         max = value
12         h = f
13     W.add(f)
14     F_R.remove(h)
15   return W
```

Assuming that the values of $t^{R+}_{3|1}$ and $t^{R-}_{3|1}$ are 0.1 and 0.25, respectively, then the conditional utility value of functions f^R_2, f^R_3, and f^R_4 are 0.35, $\frac{0.7*0.1+0.3*0.25}{1} = 0.145$, and 0.33, respectively. Hence, the second function in the workflow will be f^R_2. Similarly, the third function will be the one that has the highest utility value given that both f^R_1 and f^R_2 have been already applied on v_i. Thus, the generated workflow for the node will be $f^R_1 \rightarrow f^R_2 \rightarrow f^R_4 \rightarrow f^R_3$.

Algorithm 5 describes the process of generating a workflow for a node $v_i \in V(R)$ in the case of correlated functions. (Note that this algorithm is equivalent to Equation (9) if no correlation exists among the functions.) The input to this algorithm is the similarity functions of R and the greediness parameter $\theta = \mathcal{P}(v_i)$. At the beginning of the algorithm, the workflow $\mathcal{W}$ contains no functions. At the first iteration of the outer loop, it identifies the first function to add to the workflow. This is done in the inner loop by iterating over all the functions, and picking the one with the highest contribution per cost unit value. Then, the algorithm adds the chosen function to $\mathcal{W}$, and removes it from the functions F_R so that it is not considered when choosing the next function to add to the workflow. Note that determining the next function to add to the workflow is affected by the functions that have been already added to the workflow (Lines 7–8). For instance, the function GET-POS-CONTRIBUTION(.) returns the conditional positive contribution of function f given the functions that already exist in $\mathcal{W}$. In Example 6.3, after adding f^R_1 and f^R_3 to the workflow, calling the GET-POS-CONTRIBUTION(.) function with these values (f^R_2, $\mathcal{W}$) will return $t^{R+}_{2|\{1,3\}}$.

6.4 Learning Contribution Values

In order to compute the contribution values of similarity functions, we assume that a training dataset $\mathcal{D}'$ is available. This dataset is fully labeled with either "`duplicate`" or "`distinct`" for each pair of entities. Each entity-set $R' \in \mathcal{D}'$ consists of $|R'|$ entities. Let $V(R')$ be the set of nodes for entity-set R'. We divide these $V(R')$ nodes into two subsets $V^+(R')$ and $V^-(R')$ such that $V^+(R')$ contains only the duplicate nodes and $V^-(R')$ contains only the distinct nodes. These two subsets are used to compute the positive and negative contribution values of the similarity functions in F_R, respectively.

ALGORITHM 6: Computing a Contribution Value.

```
Comp-Contribution(R', Fun, positive)
1   NodeSet M;
2   Double total;
3   if positive = true then
4     M ← V+(R')
5   else
6     M ← V−(R')
7   foreach v_i ∈ M do
8     f_i ←New-Vector()
9     foreach f_j^R ∈ F_R do
10      if f_j^R ∈ Fun then
11        Apply f_j^R on v_i and then update f_i
12    Fill-Missing-Values(f_i)
13    ⟨sim_i, dis_i⟩ ← ℜ_R(f_i)
14    if positive = true then
15      total ← total + sim_i
16    else
17      total ← total + dis_i
18  return total/|M|
```

We will first present the algorithm that computes the contribution values for a group of functions. Then, we will explain how we use that algorithm to compute all required contribution values for the functions in F_R. We will describe only how we compute the positive contribution values using $V^+(R')$. The negative contribution values are computed in the same way but with $V^-(R')$ instead.

Learning algorithm. Algorithm 6 describes how the contribution value for a group of functions is computed. The algorithm takes as input an entity-set R', a group of functions *Fun* for which we want to compute the contribution value, and whether we want to compute the positive or negative contribution. The outer *foreach* loop iterates over all nodes in M, and, for each node, initializes a feature vector $\mathbf{f}_i$ and then applies only the functions in *Fun* on that node. The functions in *Fun* constitutes only a subset of the features in $\mathbf{f}_i$. Thus, the algorithm assigns a missing value $*$ for each other feature (Line 12). Next, the algorithm passes the feature vector to the resolve function $\mathfrak{R}_R$ and then adds up the value of sim_i or dis_i to the variable *total* depending upon the value of *positive*. This process continues until we iterate over all nodes in M. Finally, the algorithm returns the average value of the collected evidence.

Contribution Computation. We will illustrate the process of computing the contribution values using the following example.

Example 6.4. Consider the four similarity functions in Example 6.3. Let us assume that that functions f_1^R and f_3^R are correlated with each other, and functions f_3^R and f_4^R are correlated with each other.

Computing the positive contribution values for the functions in F_R consists of the following steps:

(1) We identify the correlation among functions and then create a set CF_i for each group of correlated functions. In Example 6.4, $CF_1 = \{f_1^R\}$, $CF_2 = \{f_2^R\}$, $CF_3 = \{f_3^R\}$, $CF_4 = \{f_4^R\}$,

$CF_5 = \{f_1^R, f_3^R\}$, and $CF_6 = \{f_3^R, f_4^R\}$. The sets of correlated functions can be overlapping. For instance, function f_3^R belongs to three different sets of correlated functions.

(2) We apply transitive closure on the sets of correlated functions to merge overlapping sets. For any two sets CF_i and CF_j, where $CF_i \cap CF_j \neq \emptyset$, we merge CF_j with CF_i and eliminate CF_j. This process is iteratively performed until there is no overlapping among the sets of functions. In Example 6.4, we merge CF_5 with CF_1 and then eliminate CF_5, merge CF_3 with CF_1 and then eliminate CF_3, merge CF_6 with CF_1 and then eliminate CF_6, and finally merge CF_4 with CF_1 and then eliminate CF_4.

(3) For each remaining set CF_i, we call Algorithm 6 with each non-empty subset in $P(CF_i)$, where $P(CF_i)$ is the power set of CF_i. In Example 6.4, we call the algorithm for $CR_1 = \{f_1^R, f_3^R, f_4^R\}$ seven times with these parameters: $(R', \{f_1^R\}, \text{true})$, $(R', \{f_3^R\}, \text{true})$, $(R', \{f_4^R\}, \text{true})$, $(R', \{f_1^R, f_3^R\}, \text{true})$, $(R', \{f_1^R, f_4^R\}, \text{true})$, $(R', \{f_3^R, f_4^R\}, \text{true})$, and $(R', \{f_1^R, f_3^R, f_4^R\}, \text{true})$. We denote the output of Algorithm 6 when called with these parameters (R', X, true) as $Cont(X)$.

(4) For each function $f_j^R \in CF_i$, we first set its positive contribution t_j^{R+} to $Cont(\{f_j^R\})$, and then set the conditional positive contribution value $t_{j|X}^{R+}$, for each set $X \in (P(F_R - \{f_j^R\}) - \{\})$, as follows:

— *Case* 1: If $X \subset CF_i$, then $t_{j|X}^{R+} = Cont(X \cup \{f_j^R\}) - Cont(X)$.

— *Case* 2: If $X \not\subset CF_i$ but $X \cap CR_i \neq \emptyset$, then $t_{j|X}^{R+} = t_{j|Y}^{R+}$ where $Y = X \cap CF_i$.

— *Case* 3: If $X \cap CF_i = \emptyset$, then $t_{j|X}^{R+} = t_j^{R+}$.

In Example 6.4, $F_R = \{f_1^R, f_2^R, f_3^R, f_4^R\}$ and hence, for $f_1^R \in CR_1$, we set its positive contribution t_1^{R+} to $Cont(\{f_1^R\})$, and then set these conditional contribution values as follows:

— $t_{1|\{2\}}^{R+} = t_1^{R+}$ (Case 3).

— $t_{1|\{3\}}^{R+} = Cont(\{f_1^R, f_3^R\}) - Cont(\{f_3^R\})$ (Case 1).

— $t_{1|\{4\}}^{R+} = Cont(\{f_1^R, f_4^R\}) - Cont(\{f_4^R\})$ (Case 1).

— $t_{1|\{2,3\}}^{R+} = t_{1|\{3\}}^{R+}$ (Case 2).

— $t_{1|\{2,4\}}^{R+} = t_{1|\{4\}}^{R+}$ (Case 2).

— $t_{1|\{3,4\}}^{R+} = Cont(\{f_1^R, f_3^R, f_4^R\}) - Cont(\{f_3^R, f_4^R\})$ (Case 1).

— $t_{1|\{2,3,4\}}^{R+} = t_{1|\{3,4\}}^{R+}$ (Case 2).

6.5 Associating Workflows with Nodes

As explained in Section 5, estimating the resolution cost of a node requires knowing the workflow that will be used to resolve that node. One naive strategy of associating a workflow with a node v_i would be to instantly generate a workflow for v_i using Algorithm 5. Such a strategy, however, can be inefficient as ProgressER will have to sort the similarity functions every time it needs to estimate the resolution cost of a node. Thus, ProgressER follows a more efficient strategy for associating workflows with nodes. This strategy requires ProgressER to maintain a list of ω pre-generated workflows $W_1^R, W_2^R, \ldots, W_\omega^R$ for each entity-set $R \in \mathcal{D}$. Each workflow W_k^R is generated as described above with the value of θ set to $\frac{k-1}{\omega-1}$. For example, if $\omega = 5$, then the θ values that ProgressER uses to generate these ω workflows will be 0, 0.25, 0.5, 0.75, and 1.0, respectively.

Given this strategy, to associate a workflow with a node $v_i \in V(R)$, ProgressER simply maps the node v_i to the workflow W_k^R whose θ value, i.e., $\frac{k-1}{\omega-1}$, is the closest (among all the workflows' of R) to the value of $\mathcal{P}(v_i)$, where the value k is computed as follows:

$$k = \lceil [\mathcal{P}(v_i) - (1/((\omega - 1) * 2))]/(1/(\omega - 1)) \rceil + 1. \tag{11}$$

For example, if $\omega = 5$ and $\mathcal{P}(v_i) = 0.65$, then $k = 4$, which means that the fourth workflow W_4^R whose θ value is 0.75 will be associated with v_i.

6.6 Estimating Resolution Cost

Given a node $v_i \in V(R)$ and its workflow W_k^R, ProgressER estimates the value of $C^{res}(v_i)$ as follows:

$$C^{res}(v_i) = \mathcal{P}(v_i) * C^{res+}\left(W_k^R\right) + (1 - \mathcal{P}(v_i)) * C^{res-}\left(W_k^R\right), \tag{12}$$

where $C^{res+}(W_k^R)$ (resp. $C^{res-}(W_k^R)$) is the expected cost that should be incurred to resolve a duplicate (resp. distinct) node using the workflow W_k^R. Such values can be easily learned from a training dataset.

7 TRAINING-BASED RESOLVE FUNCTIONS

As explained in Section 2, a resolve function $\mathfrak{R}_R$ is responsible for determining whether two entities r_i and r_j refer to the same real-world object or not. In practice, such a resolve function is often implemented as a machine-learning classifier such as a Naive Bayes, a Support Vector Machine (*SVM*), or a Decision Tree (*DTC*).

Training a classifier for an entity-set R depends upon whether the set $Dep(R)$ is empty or not. If $Dep(R) = \{\}$, which means that the resolution of R is not influenced by the resolution of any other entity-sets, then training a classifier for R is a well-known problem and has been studied extensively in the literature (Bilenko and Mooney 2003; Sarawagi and Bhamidipaty 2002). However, if $Dep(R) \neq \{\}$, which means that R is the dependent entity-set of some influences of $\mathcal{D}$, then training a classifier for R is challenging as it depends on the effectiveness of the resolve functions of the influencing entity-sets.

To illustrate, consider a publication dataset that has the same characteristics as the one shown in Table 2. To train a classifier for P, we need to create a feature vector of size five (three similarity functions and two influences $L_{A\rightarrow P}$ and $L_{U\rightarrow P}$) for each pair of paper entities in the training dataset, and annotate that vector with the correct resolution decision, i.e., either `duplicate` or `distinct`. The first three values in the feature vector of a pair $\langle p_i, p_j\rangle$ are simply the outputs of applying the similarity functions f_1^P, f_2^P, and f_3^P on the pair. However, it is not exactly obvious how to specify the fourth and fifth values in the vector (i.e., the values that correspond to the influences in $Dep(P)$).

One approach to this problem would be to assign values to those features based on the correct resolution decisions of the influencing pairs. For instance, we assign $\{\langle 1.0, 0.0\rangle\}$ to the fifth feature, which corresponds to the influence $L_{U\rightarrow P}$, if the venues of the two papers $\langle p_i, p_j\rangle$ are duplicate or $\{\langle 0.0, 1.0\rangle\}$ otherwise. We refer to this approach as *BasicTraining*. Such an approach might, however, lead to wrong resolution decisions during the actual resolution process. The classifier of P might wrongly classify a pair of paper entities if the resolve function $\mathfrak{R}_U$ cannot determine the correct decision of the influencing venue pair.

As an example, consider the publication dataset shown in Table 2. Suppose ProgressER first resolved the venue pair $\langle u_1, u_3\rangle$. Since the venue names of u_1 and u_3 are spelled very differently, the output similarity confidence for the venue pair will be low. Now, let us consider resolving the pair $\langle p_1, p_3\rangle$ next. Suppose that the similarity between the non-reference attribute values of p_1 and p_3 is not sufficient to declare them to be duplicate. Since the classifier of P has been trained (using the *BasicTraining* approach) with only two similarity confidence values of the venue pairs (i.e., either 0 or 1), it will be more likely that the classifier will wrongly assume that the venue pair is distinct, and hence might return 1 as the dissimilarity confidence value of $\langle p_1, p_3\rangle$, declaring it to be distinct. This wrong decision might then be propagated to the direct and indirect dependent

ALGORITHM 7: Training Classifiers.

```
ITERATIVE-TRAINING(n, 𝒟, BCL)
1   CL ← Set[n + 1]
2   CL[0] ← BCL
3   for i ← 0 to n − 1 do
4     G ←CREATE-GRAPH(𝒟)
5     foreach R ∈ 𝒟 do
6       foreach v_j ∈ V(R) do
7         Apply Similarity Functions F_R on v_j
8         Apply 𝔎_R on v_j // 𝔎_R uses CL[i].R
9         H ← H ∪ GET-DEPENDENT-NODES(v_j)
10      PROPAGATE-DECISIONS(G, H, CL[i])
11      foreach R ∈ 𝒟 do
12        foreach v_j ∈ V(R) do
13          x ←CREATE-FEATURE-VECTOR(v_j, G)
14          FV.R ← FV.R ∪ x
15        CL[i + 1].R ←BUILD-CLASSIFIER(FV.R)
16    return CL[n]
```

pairs of $\langle p_1, p_3 \rangle$, and negatively affect their resolution decisions. For instance, the resolve function $\mathfrak{R}_A$ might be misled with the wrong decision of $\langle p_1, p_3 \rangle$ when it is applied on the dependent pair $\langle a_1, a_3 \rangle$. The propagation of many wrong decisions through the graph will deteriorate the quality of the result.

To alleviate this problem, the resolve function $\mathfrak{R}_R$ should be trained in such a way that makes it resistant to such misleading behavior. That is, the value of the features that correspond to the influences in $Dep(P)$ should be specified based on the expected output of the associated resolve functions rather than the correct resolution decisions of the influencing pairs. In our example above, the fifth value of the feature corresponding to the pair $\langle p_1, p_3 \rangle$ should be set based on the expected output of $\mathfrak{R}_U$, rather than the correct decision of $\langle u_1, u_3 \rangle$.

To address this requirement, we propose an iterative algorithm, referred to as *IterativeTraining*, that trains classifiers for the entity-sets of a relational dataset. Algorithm 7 presents a high-level overview of *IterativeTraining*. The input to this algorithm is the number of iterations n, a training dataset $\mathcal{D}$, and a set of classifiers that have been trained using the *BasicTraining* approach. The set BCL contains a single classifier for each $R \in \mathcal{D}$. The variable CL is an array of $n + 1$ such sets of classifiers, where the first set $CL[0]$ contains the input classifiers and each of the remaining sets corresponds to a particular iteration.

In each iteration, the algorithm uses the classifiers of the previous iteration to resolve the dataset, and prepares a new set of training feature vectors. Each iteration can be conceptually viewed as consisting of two phases: resolution phase (Lines 5–10), and training phase (Lines 11–15). In the resolution phase, the algorithm first resolves each node in G and then adds its dependent nodes to the set H. Next, it calls the PROPAGATE-DECISIONS(.) function, which iterates over the nodes in H, and for each node $v_k \in V(R)$ that is not `certain`, applies the resolve function $\mathfrak{R}_R$ on v_k and then adds its dependents nodes in H to further propagate the decision of that node. To ensure that the propagation process terminates, that function inserts those dependent nodes into H only if v_k was resolved to a certain decision, or if the increase in sim_k or dis_k due to the last application of the resolve function on v_k exceeds a small constant. This phase is similar to the algorithm proposed in Dong et al. (2005). Note that the input set of basic classifiers are only needed to determine the resolution decisions of the nodes in the first iteration.

In the training phase, the algorithm creates a feature vector for each node $v_j \in G$. The value of each feature depends upon whether the feature corresponds to a function f_k^R or an influence $L_{S \to R}$. If the feature corresponds to a function f_k^R, then its value is the output of applying f_k^R on v_j. Otherwise, its value is the set of confidence value pairs of all nodes influencing v_j via $L_{S \to R}$. (Note that such confidence pairs are the outputs of the classifier of S that was trained in the previous iteration). Then, the algorithm adds the newly created vector to the set $FV \cdot R$. After iterating over all nodes of R, the algorithm builds a new classifier $CL[i+1] \cdot R$ for R using the set $FV \cdot R$.

Finally, the algorithm returns $CL[n]$ which is the set of classifiers that were trained in the last iteration of the *for* loop.

We note that this training algorithm is needed only if the resolve functions are implemented as machine-learning classifiers. Other possible implementations of resolve functions may not require a training dataset or a training algorithm. We also note that the classifiers generated using this algorithm can be used as resolve functions not only in our approach but also in several other relational ER approaches such as those proposed in Dong et al. (2005) and Whang and Garcia-Molina (2012).

8 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of ProgressER using publication and synthetic datasets.

8.1 Experimental Setup

8.1.1 Experimental Environment. Our algorithms were implemented in Java and the experiments were conducted on an Intel Core 2 Quad Processor running at 3.00GHz with 8GB of RAM.

8.1.2 Datasets. We utilized in our experiments two publication datasets called CiteSeerX[7] and MedLine[8] and synthetic datasets. The experimental results using these datasets are presented in Sections 8.2, 8.3, and 8.4, respectively.

8.1.3 Block Instantiation Cost. In our experiments, the publication and synthetic datasets are initially stored on disk. Each block R_i is stored in a single file that contains the entities of R_i along with their dependency information, i.e., which entities are dependent upon the entities of R_i via influence $L_{R \to S} \in Inf(R)$.[9] However, the information of which blocks those dependent entities belong to is stored in different files. Thus, the instantiation/loading cost of block R_i can be estimated as follows:

$$C^{ins}(R_i) = C^f(R_i) + |V(R_i)| * cc + \sum_{L_{R \to S} \in Inf(R)} C^b(S), \tag{13}$$

where $C^f(R_i)$ is the cost of reading the file that contains the block R_i from disk and it is a function of the number and the size of entities in R_i, cc is the cost of constructing a node for a pair of entities, and $C^b(S)$ is the cost of reading the blocking information of the referenced entities of S.

8.1.4 Quality Metric. Since the goal of our approach is to resolve as many duplicate pairs as possible using the budget BG, we use the pairwise *duplicate recall* as our quality metric. Duplicate

[7] http://csxstatic.ist.psu.edu/about/data.

[8] http://www.ncbi.nlm.nih.gov/pubmed.

[9] Each entity-set R has as many reference attributes as the number of influences in $Inf(R)$. The value of a reference attribute (that corresponds to an influence $L_{R \to S}$) for entity r_i contains the IDs of the entities of S that are dependent upon r_i via $L_{R \to S}$ (as presented in Table 2).

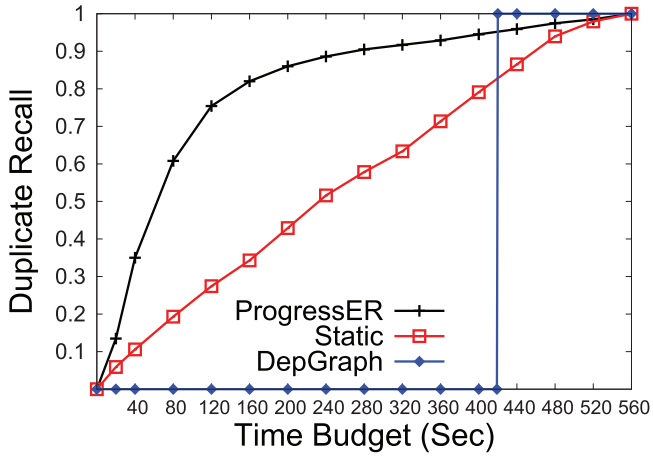

Fig. 6. Duplicate recall for different budget values.

recall is the ratio of the correctly resolved duplicate pairs to the total number of duplicate pairs in the ground truth. We do not use the duplicate precision (the ratio of the correctly resolved duplicate pairs to the total number of resolved duplicate pairs) because ProgressER always achieves higher than 0.99 precision.

8.2 CiteSeerX Dataset Experiments

In this section, we evaluate the efficacy of ProgressER using a real-world publication dataset called CiteSeerX. We obtained a subset of 30,000 publications from the entire collection, and then extracted from those publications information regarding Papers (P), Authors (A), and Venues (U) according to the following schema: Papers (Title, Abstract, Keywords, Authors, Venue), Authors (Name, Email, Affiliation, Address, Papers), and Venues (Name, Year, Pages, Papers). The cardinalities of the resultant entity-sets are $|P| = 30{,}000$, $|A| = 83{,}152$ and $|U| = 30{,}000$. (From each CiteSeerX publication, we extracted one paper entity, one venue entity, and as many author entities as the number of authors of the publication.) The CiteSeerX dataset does not come with its own ground truth; thus, we computed it by simply running the *DepGraph* algorithm (Dong et al. 2005) on the obtained dataset.

Each entity-set is divided into a set of blocks. We used two blocking functions to partition entity-set P into a set of *overlapping* blocks. The first function partitions the entities based on the first three characters of their titles, whereas the second function partitions them based on the last three characters of their titles. Also, we used a blocking function that partitions entity-set A into blocks based on the first character of the author's first name appended with the first two characters of his/her last name. Similarly, we used a blocking function that partitions entity-set U into blocks based on the first two characters of the venue name appended with the first two digits of the venue year. The total time needed to divide these three entity-sets into blocks is 76 seconds.

For this dataset, we use the same set of similarity functions given in Table 3 with the addition of four functions f_3^A, f_4^A, f_2^U, and f_3^U. These four functions are defined on the A.Affiliation, A.Address, U.Year, and U.Pages, respectively, and use the Edit-Distance algorithm to compute the similarity between their input values. The resolve function of each entity-set is implemented as a Naive Bayes classifier.

8.2.1 Budget vs. Recall. Figure 6 plots the duplicate recall as a function of the resolution cost budget (measured as the end-to-end execution time) for three ER algorithms. The *DepGraph* approach is the reference reconciliation algorithm proposed in Dong et al. (2005). Although this approach is not progressive, it is one of the few algorithms that resolve entities of multiple different types simultaneously without having to divide the resolution based on the entity type. On the other hand, the *Static* algorithm is a variant of our approach that differs only in how the SortBlocks(.) function in Algorithm 4 sorts the blocks. In this variant, the order of the blocks is *statically* determined at the beginning of the execution as follows. First, we sort the entity-sets based on their influences according to Whang and Garcia-Molina (2012). For example, if $Inf(R) = \{L_{R \to T}\}$, $Inf(S) = \emptyset$, and $Inf(T) = \{L_{T \to S}\}$, then R's blocks will appear first in the sorted list of blocks, then T's blocks, and finally S's blocks. Then, blocks of the same entity-set are sorted in a random fashion. In our implementation of *Static*, we chose P's blocks to appear first in the sorted list of blocks, then A's blocks, and finally U's blocks.

We note that this experiment does not study the benefit of using the lazy resolution strategy in resolving nodes. Hence, when resolving a pair of entities, all approaches apply all the corresponding similarity functions on that pair even if some of them are sufficient to resolve the pair.

To plot the curve of ProgressER, we ran it with different resolution budget values (correspond to the points on the curve). For each budget value, we ran ProgressER 10 times, recorded the achieved recall of each run, and then took the average recall of the 10 recall values. The curve of *Static* was plotted in the same way as we plotted the curve of ProgressER. For the *DepGraph* approach, we ran it to completion 10 times, recorded the completion time of each run, and then took the average completion time of the 10 values. Then, the recall corresponding to any budget value is set to zero if that budget value is less than the average completion time, or to one otherwise.

The results in Figure 6 show that ProgressER achieves high duplicate recalls using limited budget values. The performance gap between ProgressER and the *Static* approach demonstrates the importance of employing a good strategy for selecting which blocks to load into memory next. The *DepGraph* approach is not progressive, and thus it reaches the maximum recall only after resolving the entire dataset. Note that the two progressive approaches need more time, compared to the *DepGraph* algorithm, to reach the maximum recall. This amount of extra time represents the overhead that these approaches need to incur to perform progressive ER.

8.2.2 Lazy Resolution and Execution Time Phases. This experiment studies the benefit of resolving the nodes using the lazy resolution strategy with workflows. We compare ProgressER with two variants of it. The first one, referred to as *Full*, does not use the lazy resolution strategy. That is, when resolving a node $v_i \in V(R)$, it applies all the similarity functions of R on that node and then calls the resolve function to determine its resolution decision. The second variant, referred to as *Random*, uses the lazy resolution strategy but applies the similarity functions on a node in a random order.

To conduct this experiment, we ran each of the three approaches 10 times with the minimum budget value (the end-to-end execution time) that is sufficient for the approach to resolve all pairs in the dataset (i.e., apply the similarity functions on every pair). For each run, we recorded the exact total execution time along with the breakdown of that time (the times spent on generating plans, on reading blocks, on creating the graph, and on resolving nodes). Next, we took the average of those values and reported them in Table 5. For example, we ran ProgressER with $BG = 310$ seconds 10 times, and found that on average it finishes in 300.33 seconds and that the plan generation process (Section 5) takes on average 4.76% of the total execution time, whereas the plan execution process takes on average 95.11% (4.7% for reading blocks from disk, 8.4% for incrementally creating the

Table 5. Phases of the Execution Time for Three Different Approaches

	ProgressER	Random	Full
Execution time (seconds)	300.33	396.55	542.43
Plan generation	4.76%	3.81%	2.58%
Graph creation	8.40%	6.25%	4.72%
Reading blocks	4.70%	3.75%	2.90%
Node resolution	82.01%	86.17%	89.78%

Table 6. Similarity Functions of Each Curve

	f_1^P	f_2^P	f_3^P	f_1^A	f_2^A	f_3^A	f_4^A	f_1^U	f_2^U	f_3^U
Set_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Set_2	✓		✓	✓		✓		✓	✓	
Set_3	✓			✓				✓		

graph, and 82.01% for resolving the nodes). The achieved final recall of each approach using the specified time budget is 1.0.

This experiment demonstrates the following. First, using the lazy resolution strategy with workflows can significantly reduce the cost of applying the similarity functions on the nodes. This, in turn, offers the flexibility for developers to plug in multiple similarity functions of various cost and contribution values without having to worry about the cost of applying those functions as ProgressER can systematically resolve the nodes with the least amount of cost. Second, the percentage of the plan generation process decreases as the cost of resolving the nodes increases. In general, modern ER solutions employ more computationally expensive similarity functions, e.g., Nuray-Turan et al. (2012), Wang et al. (2012), and hence the overhead of generating resolution plans in such cases would be less noticeable.

8.2.3 Number of Similarity Functions. This experiment studies the effect of the number of similarity functions on the quality and efficiency of ProgressER. Each curve in Figure 7 corresponds to the performance of ProgressER when provided with a different set of similarity functions. The entire set of similarity functions and the subsets of those functions that are used with the three curves are shown in Table 6.

The result of this experiment demonstrates the benefit of resolving nodes using the lazy resolution strategy with workflows. While *Set_1* uses four more functions than *Set_2*, they both behave similarly. Those four functions do not cause any significant increase in the execution time because ProgressER in *Set_1* can resolve the nodes with least amount of cost by applying only the necessary functions on them. In fact, *Set_2* performs slightly worse than *Set_1* because some nodes might require applying more than the two defined functions to be resolved to a certain decision. Such uncertain nodes will, however, be eventually resolved to certain decisions when their influencing nodes are resolved and their decisions are propagated to those uncertain nodes.

This experiment also shows that defining insufficient number of similarity functions might lead to a drop on the maximum achieved recall; *Set_3* could not achieve more than 0.87 recall. ProgressER, however, aims to achieve that maximum recall as quickly as possible. For instance, it achieves 50% of that maximum recall in almost one fourth of the time needed to reach the maximum recall.

8.2.4 Correlation among Similarity Functions. This experiment compares ProgressER with its variant, referred to as *BasicGenerator*, that differs only in how workflows are generated. This

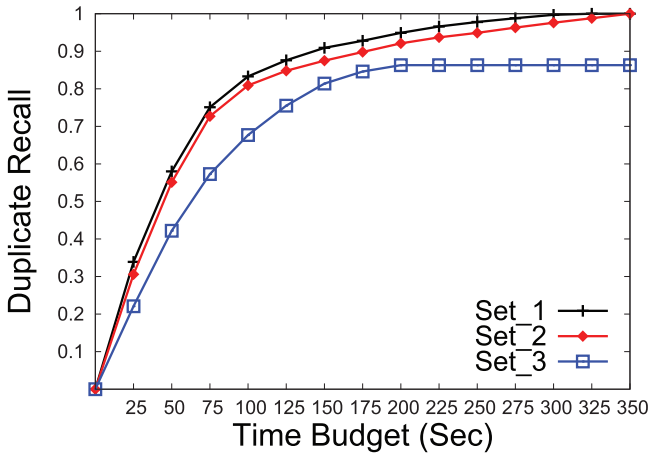

Fig. 7. Effect of similarity functions.

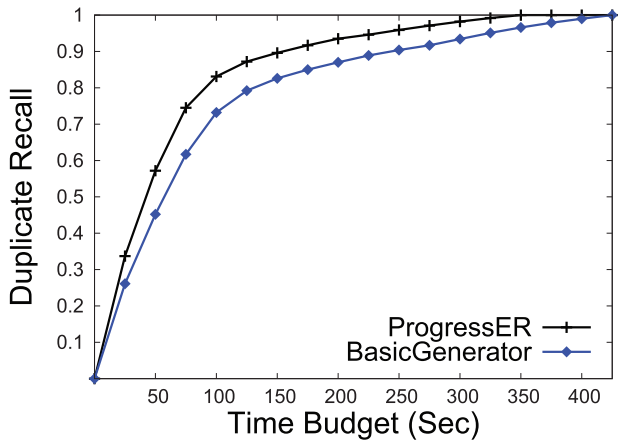

Fig. 8. Correlation among similarity functions.

variant assumes that the similarity functions are *uncorrelated*, and thus does not account for the *conditional* contribution values when generating workflows.

In addition to the set of similarity functions explained earlier, we added three more similarity functions to make a subset of each entity-set's functions correlated. The new functions f_4^P, f_5^A, and f_4^U are defined on the P.Title, the A.Name, and the U.Name attributes, respectively. Function f_4^P uses the soft TF.IDF measure to compute the similarity between two title values, whereas functions f_5^A and f_4^U use the JaroWinkler distance algorithm to compute the similarity between their input values. According to the definition of correlated functions, f_4^P is correlated with f_1^P, f_5^A is correlated with f_1^A, and f_4^U is correlated with f_1^U.

Figure 8 plots the result of this experiment. When resolving nodes, the *BasicGenerator* approach prefers to apply similarity functions that are correlated with the functions that have been already applied on the nodes. The application of such functions does not provide the expected positive and/or negative contribution, and thus unnecessarily increase the node resolution cost. In contrast, by considering the correlation among the similarity functions, ProgressER can delay or even

avoid the application of such unnecessary functions, which in turn reduces the resolution cost, and hence, improves the quality of ProgressER.

8.3 MedLine Dataset Experiments

In this section, we study the performance of our *IterativeTraining* algorithm using the MedLine dataset, which consists of a collection of publications in the area of bio-medicine and health. This dataset does not contain duplicate publications and hence we constructed the testing dataset as follows:

(1) We obtained a set of 10,000 publications from the entire dataset. Each publications consists of several attributes that contain information about its title, abstract, keywords, authors, venue, and so on. In this set, the same venue or author entity can be shared among multiple publications.
(2) We generated d duplicate entities for each publication such that the average number of duplicate entities of a publication is three and the value d forms a zipfian distribution with an exponent of 0.2.
(3) We used a data generator similar to the one used in Hassanzadeh et al. (2009) to inject various types and percentages of errors in the publication attribute values.
(4) We extracted from the publications information regarding Papers (P), Authors (A), and Venues (U) according to the following schema: Papers (Title, Abstract, Keywords, Authors, Venue), Authors (Name, Papers), and Venues (Name, Year, Papers). The cardinalities of these entity-sets in this testing dataset are $|P| = 30{,}000$, $|A| = 52{,}266$ and $|U| = 369$.
(5) We performed blocking on the entity-sets as follows. We used two blocking functions to partition entity-set P into overlapping blocks. The first function partitions the entities based on the first two characters of their titles, whereas the second function partitions them based on the last two characters of their titles. Similarly, we used two blocking functions to partition entity-set A into overlapping blocks. The first function partitions the entities based on the first two characters of their first names, whereas the second function partitions them based on the first two characters of their last names. For entity-set U, we partitioned its entities based on the first two characters of their venue names. The total time needed to divide the entity-sets of the testing dataset into blocks is 56 seconds.

The training dataset was constructed using exactly the same steps above. The number of publications chosen in the first step is 10,000 and the cardinalities of the entity-sets are $|P| = 30{,}000$, $|A| = 50{,}334$ and $|U| = 432$.

We note that we used this MedLine dataset because evaluating the efficacy of our *IterativeTraining* algorithm requires knowing the ground truth of the input testing dataset, which is not available for the CiteSeerX dataset.

For the similarity functions, we use the same set of functions given in Table 3 except for f_2^A and with the addition of one function f_2^U that is defined on the Year attribute and uses the Edit-Distance algorithm to compute the similarity between two year values. The resolve function of each entity-set is implemented as a Naive Bayes classifier.

8.3.1 Number of Iterations. To study the effectiveness of our *IterativeTraining* algorithm, we conducted an experiment in which we ran ProgressER to completion on the testing dataset five times. The *maximum* recall values of the three entity-sets of each run are shown in Table 7. In each run, we used a different set of classifiers as the resolve functions. This set of classifiers was obtained by running the *IterativeTraining* algorithm using the training dataset with the

Table 7. Effects of Number of Training Iterations

Entity-set	Number of iterations (the value n in Algorithm 7)				
	0	1	2	3	4
P	0.96	0.97	0.99	1.0	1.0
A	0.97	0.99	1.0	1.0	1.0
U	0.97	0.99	1.0	1.0	1.0

Table 8. Size of the Training Datasets

	MT-1	MT-4	MT-7	MT-10	MT*-1	MT*-4	MT*-7	MT*-10
Number of publications	1,000	4,000	7,000	10,000	1,000	4,000	7,000	10,000
$\|P\|$	3,000	12,000	21,000	30,000	3,000	12,000	21,000	30,000
$\|A\|$	6,312	21,612	34,632	50,334	8,508	35,094	62,184	86,817
$\|U\|$	90	177	261	432	3,000	12,000	21,000	30,000

corresponding n value. Note that the classifiers used when $n = 0$ are the ones generated by the *BasicTraining* algorithm.

The results show that the recall values increase as we increase the number of iterations and that we needed only three training iterations to reach the maximum recall. In comparison with *BasicTraining*, our training algorithm has improved the recall values for the three entity-sets P, A, and U by 4%, 3%, and 3%, respectively, demonstrating the need for using a more sophisticated training algorithm to train the resolve functions in relational ER.

8.3.2 Size and Characteristics of the Training Dataset. In this section, we study the impact of the size and characteristics of the training dataset on the final recall values of the testing dataset. To this end, we ran the same experiment performed in the previous section eight times; each using a different training dataset. These eight datasets were constructed using the same steps of constructing the testing dataset but with some minor modifications in the first step of that process as we will see next. Table 8 shows the number of publications chosen in the first step for those datasets and the cardinalities of the three entity-sets in each of them.

These datasets can be divided according to their sizes and characteristics into two groups. The first group consists of four datasets *MT-1*, *MT-4*, *MT-7*, and *MT-10*. These datasets differ from each other in size and were constructed in a way that makes them representative of the testing dataset. In other words, each dataset *MT-x* was constructed using exactly the same steps of constructing the testing dataset except that the number of publications chosen in the first step is $x * 1{,}000$.

The second group consists of four datasets *MT*-1*, *MT*-4*, *MT*-7*, and *MT*-10*. Similarly, these datasets differ from each other in size; the number of publications chosen in the first step for each dataset *MT*-x* is $x * 1{,}000$. However, they were constructed in a way that makes their characteristics different from those of the testing dataset. This was achieved by choosing, in the first step, a set of publications that do not share any venue or author entities, minimizing the connectivity among the nodes in the resulting graph. Such a modification in the construction process causes the four datasets to be unrepresentative of the testing dataset.

Table 9 shows the results of these experiments. For each training dataset, the table shows the *maximum* recall achieved for each entity-set in the testing dataset and the number of iterations that *IterativeTraining* needed to produce the resolve functions that achieve those recall values. For

Table 9. Effects of the Size and Characteristic of the Training Datasets

	MT-1	MT-4	MT-7	MT-10	MT*-1	MT*-4	MT*-7	MT*-10
Entity-set P	0.99	1.0	1.0	1.0	0.97	0.97	0.97	0.97
Entity-set A	0.99	1.0	1.0	1.0	0.98	0.98	0.98	0.99
Entity-set U	1.0	1.0	1.0	1.0	0.97	0.98	0.98	0.98
Number of iterations	3	3	3	3	2	3	3	3

Table 10. Parameters of Synthetic Datasets

Parameter	Description	Value
n	Number of entity-sets	4
s	Cardinality of each entity-set	20,000
b	Number of blocks per entity-set	100
d	Fraction of duplicate pairs	0.2
z	Zipfian distribution exponent	0.15
l	Probability of generating an influence	0.3
k	Average number of dependent nodes	2

example, when using *MT*-1*, the maximum recalls for entity-sets P, A, and U are 0.97, 0.98, and 0.97, respectively. Those recalls were achieved by the resolve functions produced after running *IterativeTraining* using two iterations. Increasing the number of iterations further for that dataset did not cause the recall values to improve.

The results of these experiments demonstrate the following. First, ProgressER can achieve high recall values for a given input dataset only if the resolve functions are trained using a dataset that is representative of that input dataset. Second, the size of the training dataset has a lower impact on the results. In general, increasing the training dataset size adds more diversity to the training resolution instances and decreases the chance of overfitting; thus, improving the performance of the resolve functions. Third, training resolve functions with unnecessary additional representative instances does not cause the resolve functions to perform worse. The recall values remain perfect even after we increased the dataset size from *MT-4* to *MT-7* and then to *MT-10*.

In conclusion, the representativity of the training dataset has a higher impact on our specific resolution model than the size of that dataset. The resolve functions trained using the smallest-but-representative dataset (i.e., *MT-1*) outperformed those trained using the largest-but-unrepresentative dataset (i.e., *MT*-10*).

8.4 Synthetic Dataset Experiments

In order to evaluate ProgressER in a wider range of various scenarios, we built a synthetic dataset generator that allows us to generate datasets with different characteristics. The parameters that this generator takes as input along with their default values are shown in Table 10. We will explain below those parameters, and then discuss the effects of varying some of them on the efficacy of ProgressER and the *DepGraph* and the *Static* approaches that we described in Experiment 8.2.1.

In each synthetic dataset, we generate n entity-sets, each contains s entities. The s entities of each entity-set are divided evenly into b non-overlapping blocks. The parameter d is the fraction of duplicate pairs in each entity-set, and it is computed as the number of duplicate pairs divided by the total number of pairs after applying blocking. The duplicate pairs of an entity-set are distributed across the blocks of that entity-set using a zipfian distribution with an exponent of z. The

parameter $l \in [0, 1]$ determines the number of influences in the dataset. For each two entity-sets R and S, there is an influence from entity-set R to entity-set S with probability l. Thus, the higher the value of l is, the higher the number of influences in the dataset will be. Finally, the parameter k represents the average number of direct dependent nodes that each node $v_i \in V(R)$ can have via each influence $L_{R \to S} \in Inf(R)$. We require the direct dependent nodes of a duplicate node to be also duplicate, and the direct dependent nodes of a distinct node to be also distinct.

Each entity-set R has one non-reference twenty-five character long string attribute, and is associated with a single similarity function that is defined on that attribute. This function uses the Edit-Distance algorithm to compute the similarity between the values of that attribute. The resolve function of each entity-set employs a simple decision-making process; it returns 1 as the similarity (dissimilarity) confidence of a pair only if the associated similarity function has been applied on the pair and indicated that the two values of the string attribute are similar (distinct).

8.4.1 Duplicate Distribution. In this experiment, we study the performance of various ER algorithms when varying the value of the zipfian distribution exponent z while fixing the other parameters to their default values. When $z = 0$, all blocks have the same effect on the duplicate recall because the duplicate pairs are uniformly distributed across all blocks of the dataset. However, in the cases where the duplicate pairs are not uniformly distributed, resolving the blocks with high duplicate percentage will have higher influence on the duplicate recall than resolving those with low duplicate percentage. Hence, the higher the value of z is, the smaller the number of blocks that have the highest influence on the duplicate recall.

In Figure 9, we vary the value of z from 0 to 0.3. As anticipated, the higher the value of z is, the better ProgressER performs compared to the other algorithms. This experiment demonstrates the following. First, ProgressER can adapt itself to datasets with various duplicate distributions and therefore quickly identify and resolve the blocks with high duplicate percentage values. Second, ProgressER is more effective when the duplicate pairs are not uniformly distributed across the blocks, which is almost always the case in real-world datasets. Third, ProgressER performs well even when the resolution cost is relatively cheap; it involves applying a single similarity function on two 25-character-long string values.

8.4.2 Number of Influences. In this experiment, we study the effects of the number of influences on the performance of various ER approaches.

In Figure 10, we vary the value of l from 0 to 0.6. As expected, when $l = 0$, ProgressER behaves very similarly to the *Static* approach because there exists no influences that ProgressER can utilize to identify which blocks to load and which nodes to resolve next. In fact, the *Static* approach performs slightly better than ProgressER because it does not need to compute the usefulness values of the blocks and then sort them in each window. However, as the value of l increases, ProgressER starts to perform better than the *Static* approach because the number of influences increases and that therefore implies that declaring a node to be duplicate can guide us toward findings more duplicate nodes in the dataset.

On the other hand, the increase in the number of influences introduces a little overhead in performing the benefit-vs-cost analysis in the two progressive approaches (as more influences are involved when computing the probability and impact values) and in loading the blocks into memory in the three approaches. This, therefore, causes the approaches to run a little slower.

This experiment also emphasizes the importance of employing a proper block selection strategy because even when l is 0.6, the *Static* approach can achieve only around 0.77 recall with the same amount of time that the *DepGraph* algorithm needs to resolve the entire dataset.

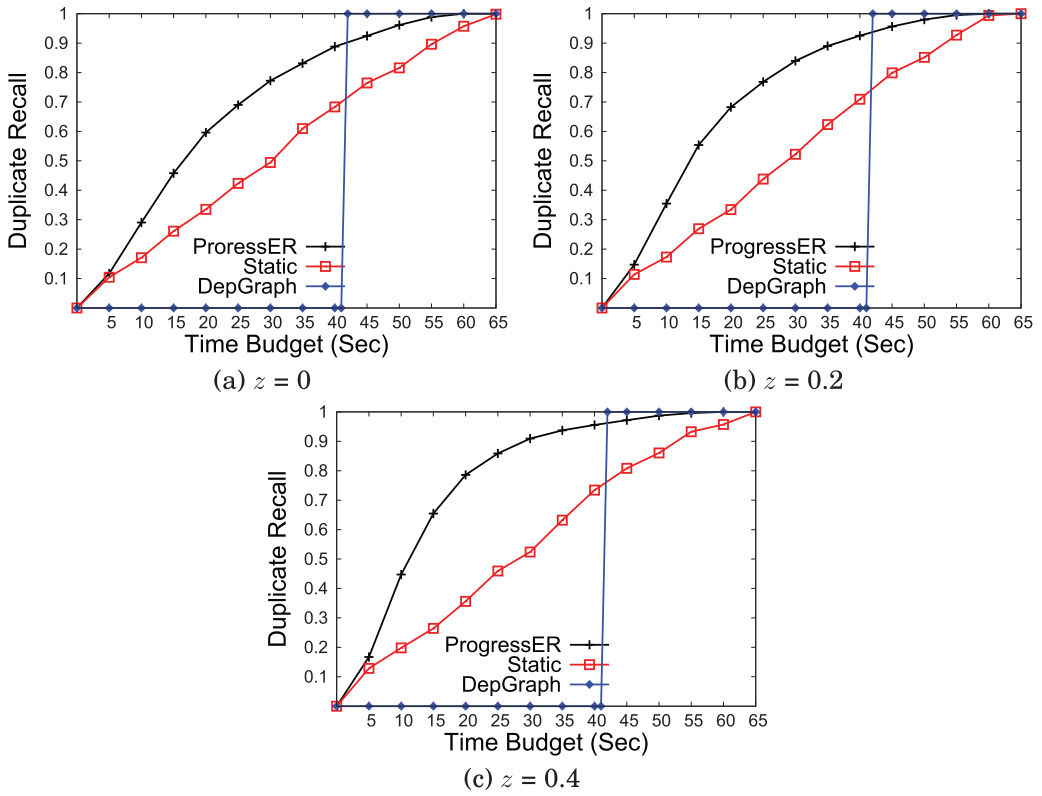

Fig. 9. Effects of zipfian distribution exponent value.

9 RELATED WORK

ER is a well-known data quality problem that has received significant attention in the literature over the past few decades (Yakout et al. 2010a; Benjelloun et al. 2009; Whang and Garcia-Molina 2012; Chen et al. 2009). A thorough overview of the existing work in this area can be found in Elmagarmid et al. (2007), Köpcke and Rahm (2010), Doan et al. (2012), and Getoor and Machanavajjhala (2012). We review below the related work in different aspects of the ER problem.

Progressive entity resolution. The most related work to our proposed approach is those of Whang et al. (2013b) and Papenbrock et al. (2015). In Whang et al. (2013b), the authors propose several concrete ways of constructing hints that can be utilized by a variety of ER algorithms to progressively resolve blocks. Moreover, Papenbrock et al. (2015) proposes two progressive hint-based ER approaches that are based on the *SN* algorithm (Hernández and Stolfo 1995). The first approach is built on the top of the *SN*'s hint proposed in Whang et al. (2013b), whereas the second one first partitions the sorted dataset into disjoint equal-size windows, resolves all entities in each window, and then dynamically/iteratively extends the windows that contain a high number of identified duplicate pairs.

As stated in Section 1, the key difference of our work from Whang et al. (2013b) and Papenbrock et al. (2015) is that we study the problem in the case of *relational* ER, whereas Whang et al. (2013b) and Papenbrock et al. (2015) do not. In fact, this difference raises an interesting question. To resolve a relational dataset $\mathcal{D}$ using a limited budget, should we use ProgressER, or should

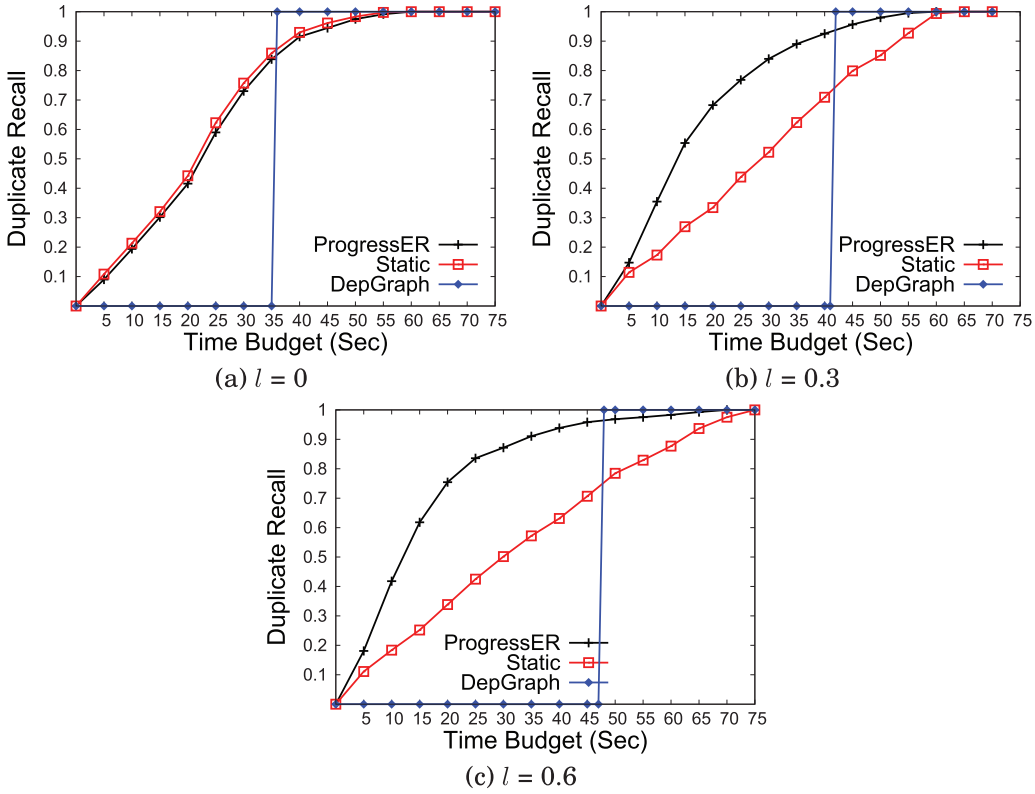

Fig. 10. Effects of the number of influences.

we resolve each entity-set in isolation using a single entity-set hint-based progressive ER algorithm (that does not exploit the relationships in $\mathcal{D}$)? ProgressER is intended for situations where exploiting those relationships is important for resolving $\mathcal{D}$. If those relationships are not important (i.e., the similarity between the attribute values is sufficient to achieve high-quality results), then using ProgressER may not provide any significant advantage over Whang et al. (2013b) and Papenbrock et al. (2015). A full characterization of under what circumstances one should choose which approach is an interesting direction of future work.

Finally, Altowim and Mehrotra (2017) propose a parallel progressive approach to ER using MapReduce. That approach first divides the input datasets into a set of (possibly overlapping) blocks, gathers some statistics about the generated blocks, and then distributes the blocks among the available machines in a way that maximizes the overall rate of duplicate detection. The approach is capable of dealing with severe skewness in block sizes and resolves each pair of entities that exists in multiple blocks only once (as opposed to resolving it redundantly in each of those blocks).

Relational ER. The problem of relational ER (also referred to as collective ER or joint ER) has been previously studied in the literature. The authors in Domingos (2004) and Culotta and McCallum (2005) propose probabilistic models that use Conditional Random Fields (CRF) to capture the dependencies among different entity-sets. In Dong et al. (2005), the algorithm performs the resolution process on different entity-sets simultaneously. The relationships among entity-sets are

leveraged to propagate the similarity increases of some pair to its dependent pairs. Moreover, the authors in Whang and Garcia-Molina (2012) propose a joint ER framework for resolving multiple datasets in parallel using custom ER algorithms. That proposed framework is not designed to be progressive. Thus, the order in which the datasets are resolved is determined at the beginning of the joint ER algorithm. Our solution, however, *dynamically* specifies the order in which the blocks are resolved based on the estimated number of duplicate pairs in those blocks.

Training-based resolve functions. Several approaches have addressed the problem of training resolve functions. Reference (Bilenko and Mooney 2003) shows how a SVM classifier can be trained and used as a resolve function. In addition, the authors in Sarawagi and Bhamidipaty (2002) propose an interactive active learning algorithm for constructing an effective, yet concise, set of training instances that are needed to train resolve functions. In contrast to our *IterativeTraining* algorithm, these techniques do not consider the case where the resolve function depends on the outcomes of other resolve functions as in the case of relational ER. The authors in Whang and Garcia-Molina (2012) address this problem but they use multiple different resolve functions for the same entity-set. The choice of which function to use when resolving an entity-set is determined based on the state of the resolution process, i.e., based on which entity-sets have been resolved so far. That training algorithm cannot be adopted in our case since ProgressER resolves nodes of different entity-sets at the same time.

New ER approaches. In addition to progressive ER, several new approaches, such as incremental ER and analysis-aware ER, have been recently proposed in the literature. For instance, incremental ER such as Whang and Garcia-Molina (2014) and Gruenheid et al. (2014) addresses the problem of maintaining an up-to-date ER result when data updates arrive quickly. Instead of recomputing the ER result each time from scratch, such approaches leverage the previous result to efficiently compute the updated one.

Furthermore, analysis-aware approaches such as Bhattacharya and Getoor (2007b) and Wang et al. (2014) aim to exploit the knowledge of the analysis task to reduce the amount of cleaning required. For instance, the approach in Bhattacharya and Getoor (2007b) answers mention matching queries (e.g., retrieve *all* papers written by author "J. Smith") collectively using a two-phase algorithm. It first retrieves the related records for a query using two expansion operators and then answers the query by only considering the extracted records. In addition, Wang et al. (2014) uses a sampling-based technique to answer aggregate queries on top of a dirty dataset. Their idea is to first apply ER on a relatively small sample of the dataset and then use the resolution result to approximately answer queries.

Contributions of this article over its Conference Version. This article is based on our initial work (Altowim et al. 2014) that appeared in VLDB 2014. Due to the space constraint of VLDB, the preliminary version did not include all of our contributions. In this article, we present a more complete and comprehensive coverage of the materials. More precisely, the new contributions of this article are as follows:

- We present a technique for dealing with correlation among similarity functions when generating workflows for resolving nodes (Section 6.3). In our conference version (Altowim et al. 2014), we simplified the workflow generation process by assuming that the similarity functions (and the attributes on which those functions are defined) are uncorrelated with each other.
- We propose an algorithm that learns the contribution values of similarity functions from a training dataset (Section 6.4). In Altowim et al. (2014), we only defined the concept of the contributions of similarity functions and showed how the cost and contribution values of

those functions can be used to generate workflows. The contribution values used in the experiments of Altowim et al. (2014) were derived using the learning algorithm that we present in this article. Due to the space constraint of VLDB, we could not add that algorithm in the conference version.

—We propose an iterative algorithm, referred to as *IterativeTraining*, for training the resolve functions of entity-sets in relational ER (Section 7). Similarly, the resolve functions used in the experiments of Altowim et al. (2014) were trained using this iterative algorithm.

—We describe how ProgressER can be extended to support the case where the blocks of an entity-set are overlapping (Appendix A.1). Note that our experiments on the publication dataset in Altowim et al. (2014) use overlapping blocks and were conducted using this extended version of the approach.

—We conduct more empirical study of different aspects of our solution. Specifically, we have added an experiment (Section 8.2.3) that studies the effect of the number of similarity functions on the performance of ProgressER, an experiment (Section 8.2.4) that shows the benefit of considering the correlation among similarity functions when generating workflows, and two experiments (Sections 8.3.1 and 8.3.2) that study the effectiveness of our *IterativeTraining* algorithm.

10 CONCLUSIONS

In this article, we have proposed a progressive approach to relational ER, named ProgressER, wherein the input dataset is resolved using only a limited budget with the aim of maximizing the quality of the result. ProgressER follows an adaptive strategy that periodically monitors the resolution progress to determine which parts of the dataset should be resolved next and how they should be resolved. We showed empirically, using real-world and synthetic datasets, that ProgressER can quickly identify and resolve the duplicate pairs in the dataset and can thus generate high-quality results using limited amounts of resolution cost.

APPENDIX

A.1 Supporting Overlapping Blocks

In order to support overlapping blocks, the process of computing the instantiation cost and usefulness values of uninstantiated blocks and the probability values of nodes needs to modified. To illustrate, consider two blocks R_1 and R_2 of entity-set R where $R_1 = \{r_1, r_2, r_3, r_4\}$ and $R_2 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$. The total number of nodes that can be created for the entities of R_1 and R_2 are $\frac{4\times3}{2} = 6$, and $\frac{6\times5}{2} = 15$, respectively. Let us assume that R_1 has been already instantiated, meaning that the six nodes of R_1 have already been created. Since the four entities of R_1 already exist in R_2, then the six nodes of R_1 overlap with the fifteen nodes of R_2. Thus, the uninstantiated block R_2 has six nodes that have already been instantiated. In general, an uninstantiated block might have instantiated nodes, which are the nodes that also belong to other already instantiated blocks. Let us denote the set of uninstantiated nodes of an uninstantiated block R_i as $V^u(R_i)$. Thus, the set $V^u(R_2)$ consists of only nine nodes.

Instantiation cost and usefulness of blocks. Knowing the exact number of the instantiated nodes of an uninstantiated block would affect the computation of the instantiation cost and usefulness of that block as follows:

—The instantiation cost $C^{ins}(R_i)$ of a block R_i would involve creating only the uninstantiated nodes of R_i. Recall that the instantiation cost $C^{ins}(R_i)$ typically consists of the cost of reading the block R_i plus the cost of creating nodes for the pairs of entities of R_i.

Although knowing the exact number of uninstantiated nodes of R_i affects the cost of creating the nodes (in the example above, only nine nodes, instead of fifteen, need to created when instantiating R_2), it does not affect the cost of reading R_i as the entire block needs to be read.

We have stated in Section 5.1 that the UPDATE-INST-COST(.) function in Algorithm 3 computes the instantiation cost of every block in **UB** at the beginning of the first window and that it does not need to update these cost values in any subsequent window. In the case of overlapping blocks, this function does need to update these cost values in subsequent windows. However, it does not need to do so for *all* uninstantiated blocks at the beginning of each subsequent window. That is, the instantiation cost of block R_i is updated in a subsequent window only if its number of instantiated nodes has increased since the previous window, i.e., at least one of the instantiated blocks in the previous window had overlapping uninstantiated nodes with R_i before it was instantiated. Instantiating these overlapping nodes in the previous window will cause the instantiation cost of R_i to be lower as the number of uninstantiated nodes of R_i that need to be created becomes smaller.

— The usefulness value $\mathcal{U}(R_i)$ of a block R_i should be computed based only on the uninstantiated nodes of that block. That is, the benefit of resolving the already instantiated nodes of R_i should not be considered when computing the value of $\mathcal{U}(R_i)$. Thus, the modified equation for computing $\mathcal{U}(R_i)$ will be

$$\mathcal{U}(R_i) = \frac{\sum_{v_j \in V^u(R_i)} Benefit(v_j)}{C^{ins}(R_i) + \sum_{v_j \in V^u(R_i)} C^{res}(v_j)}. \tag{14}$$

Probability of nodes. Since a node v_i can belong to different blocks, the probability value $\mathcal{P}(v_i)$ (as computed in Section 4.1.1) should now be dependent upon the percentage of duplicate nodes in each of those blocks. That is, the set of causes $\mathbf{X}^i$ may now contain more than one block.

In addition to the above modification, the single probability value that the COMPUTE-SINGLE-PROB-AND-COST(.) function in Algorithm 3 assigns to all unaffected uninstantiated nodes of each uninstantiated block R_i will not be always the leak value δ. This is because, if the number of instantiated resolved nodes of R_i constitutes at least α percent of the number of nodes in R_i, that single probability value will be set to $\frac{|V^+(R_i)|}{|V^*(R_i)|}$ (recall that the set of present causes of any unaffected uninstantiated node cannot contain any influencing duplicate node according to the definition of the unaffected nodes). However, assigning the same single probability for all unaffected uninstantiated nodes of R_i means that we consider R_i as the *only* cause of each unaffected uninstantiated node $v_j \in V(R_i)$, which is not always the case. This is because the node $v_j \in R_i$ might belong to other uninstantiated blocks, which indicates that these blocks should also be causes of v_i.

ProgressER, however, does not account for these blocks as causes of v_j for several reasons. First, it is very rare that these uninstantiated blocks can be present causes of the node v_j. Second, computing a probability and a cost value for each individual uninstantiated node can be very inefficient in terms of both space and time requirements, as the set of unaffected uninstantiated nodes usually constitutes a large portion of the nodes in G at most stages of the execution. Third, computing a different probability value for each unaffected uninstantiated node of R_i requires knowing all the entities of R_i (to compute a probability for each possible pair/node) and in which other blocks those entities reside (to consider those blocks as other causes). In ProgressER, such information is only available for few referenced entities, as described in the process of block instantiation in Section 3.

A.2 Parameter Learning

In this section, we discuss how to learn and set the main parameters of ProgressER.

- The cost value c_i^R: The initial value of c_i^R can be specified by a domain analyst or learned from a training or previously resolved dataset as the average execution time of applying the function f_i^R on a pair of entities r_j and r_k. In any case, c_i^R can then be continuously adjusted as ProgressER proceeds forward.
- The positive (resp. negative) contribution value t_i^{R+} (resp. t_i^{R-}): Section 6.4 presents an algorithm that learns the contribution values of similarity functions from a training dataset.
- The probability value $\mathcal{P}_{S\to R}$: This value can be estimated from a training or previously resolved dataset. In our implementation, we used a heuristic that sets this value as follows:

$$\mathcal{P}_{S\to R} = \frac{\sum_{v_i \in V^+(S)} \frac{\vartheta_i}{\mu_i}}{|V^+(S)|}, \tag{15}$$

  where $V^+(S)$ is the set of duplicate nodes of entity-set S, ϑ_i is the number of duplicate nodes of R that are directly influenced by v_i via $L_{S\to R}$, and μ_i is the number of nodes of R that are directly influenced by v_i via $L_{S\to R}$.
- The leak value δ: Similarly, this value can be set by a domain analyst or learned from a training or previously resolved dataset. In our implementation, the leak value δ of entity-set R is set to the total number of duplicate pairs of R in that dataset divided by the total number of pairs of R.
- The threshold α described in Section 4.1: This threshold represents the minimum fraction of nodes of a block that need to be resolved before we can consider that block as a present cause when estimating the probability value of any node in that block. We used this threshold to minimize the chance of having a biased probability estimation since relying on the resolution decisions of only few nodes from a block might not be a good indicator of the percentage of duplicate pairs in the block. Intuitively, the value α of a block should be set based on the block size. The larger the block is, the smaller α should be. In our implementation, we used a heuristic that sets the value α of a block R_i to be $= 2 \times |R_i|^2/|R|$.
- The fan-out degree $degree(L_{R\to S})$: This value depends on the characteristics of the dataset rather than on the ground truth of the dataset or the expected output of the similarity and resolve functions. Hence, its initial value can be learned from another similar dataset (that does not have to be resolved) or can be set by a domain expert. For instance, in the publication domain, the expert can easily determine that a pair of paper entities influences at most one pair of venue entities. (Note that an edge is created between these two pairs of entities only if the two entities of each pair share at least one block.) In any case, this initial value can be continuously adjusted as ProgressER proceeds forward.
- The number of pre-generated workflows ω: Setting this value should depend upon the number of similarity functions and the difference between the positive and negative contribution values of each of those functions. The higher the number of functions and the difference between their positive and negative contribution values are, the higher the value ω should be. In general, we should keep increasing ω as long as the generated workflows are different from each other.

  It is important to note that the processes of generating those ω workflows and associating a workflow with a node can be done efficiently even if ω is large. Generating those workflows are performed offline, whereas associating workflows with nodes is performed using a simple calculation (Equation (11)).

— The values $C^{res+}(W_k^R)$ and $C^{res-}(W_k^R)$: The value $C^{res+}(W_k^R)$ (resp. $C^{res-}(W_k^R)$) represents the average execution time of resolving a duplicate (resp. distinct) node using the workflow W_k^R. Similarly, the initial values of $C^{res+}(W_k^R)$ and $C^{res-}(W_k^R)$ can be learned from a training or previously resolved dataset, and can then continuously adjusted as ProgressER proceeds forward.
— The duration of the plan execution W: This parameter provides a tradeoff between the quality of the individual generated plans and the amount of overhead that our approach needs to incur. Using a small value for W can improve the quality of the generated plans. However, it increases the number of windows, which in turn increases the amount of required-but-unuseful tasks (i.e., updating the cost and benefit values, and generating resolution plans) that our approach needs to perform. In our implementation, we specify the value of W using a piecewise function of the total budget BG. For instance, we set W to BG/x_1 seconds if $lower_1 < BG \leq upper_1$, to BG/x_2 seconds if $lower_2 < BG \leq upper_2$, and so on. We determined the intervals and the subfunction of each interval after experimentally examined several options for them.

It is important to note that learning the parameters of our approach does not necessarily require an *accurately* labeled training dataset. This is because the values of those parameters depend upon the characteristics of the dataset and the provided similarity and resolve functions rather than on the ground truth of the dataset. For instance, to learn the contribution value of a similarity function f_i^R, we need to measure the expected output of the corresponding resolve function $\mathfrak{R}_R$ regardless of whether that output is correct *w.r.t.* the ground truth or not.

REFERENCES

Yasser Altowim, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2014. Progressive approach to relational entity resolution. *Proceedings of the VLDB Endowment*, 7, 11 (2014), 1846–1857.

Yasser Altowim and Sharad Mehrotra. 2017. Parallel progressive approach to entity resolution using MapReduce. In *Proceedings of the IEEE 33rd International Conference on Data Engineering (ICDE'17)*. 909–920.

Rohan Baxter, Peter Christen, and Tim Churches. 2003. A comparison of fast blocking methods for record linkage. In *Proceedings of the ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification*, vol. 3. ACM, 25–27.

O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. 2009. Swoosh: A generic approach to entity resolution. *The VLDB Journal*, 18, 1, 255–276.

Indrajit Bhattacharya and Lise Getoor. 2007a. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data*, 1, 1 (2007), 1–36.

Indrajit Bhattacharya and Lise Getoor. 2007b. Query-time entity resolution. *Journal of Artificial Intelligence Research*, 30 (2007), 621–657.

Mikhail Bilenko and Raymond J. Mooney. 2003. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 39–48.

Jennifer J. Burg, John Ainsworth, Brian Casto, and Sheau-Dong Lang. 1999. Experiments with the Oregon trail knapsack problem. *Electronic Notes in Discrete Mathematics*, 1 (1999), 26–35.

Z. Chen, D. V. Kalashnikov, and S. Mehrotra. 2009. Exploiting context analysis for combining multiple entity resolution systems. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. 207–218.

Gregory F. Cooper. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42, 2 (1990), 393–405.

Aron Culotta and Andrew McCallum. 2005. Joint deduplication of multiple record types in relational data. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*. 257–258.

AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration*. Elsevier.

P. Domingos. 2004. Multi-relational record linkage. In *Proceedings of the KDD-2004 Workshop on Multi-Relational Data Mining*.

X. Dong, A. Halevy, and J. Madhavan. 2005. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. 85–96.

Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19, 1 (2007), 1–16.

Lise Getoor and Ashwin Machanavajjhala. 2012. Entity resolution: Theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5, 12 (2012), 2018–2019.

Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental record linkage. *Proceedings of the VLDB Endowment*, 7, 9 (2014), 697–708.

Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J. Miller. 2009. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment*, 2, 1 (2009), 1282–1293.

Max Henrion. 1987. Practical issues in constructing a Bayes' belief network. In *Proceedings of the 3rd Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI'87)*. 132–139.

M. A. Hernández and S. J. Stolfo. 1995. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, vol. 24, no. 2. ACM, 127–138.

Finn Verner Jensen and Thomas Dyhre Nielsen. 2007. *Bayesian Networks and Decision Graphs*. Springer.

Hanna Köpcke and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69, 2 (2010), 197–210.

Solomon Kullback. 1997. *Information Theory and Statistics*. Courier Corporation.

J. F. Lemmer and D. E. Gossink. 2004. Recursive noisy OR - A rule for estimating complex probabilistic interactions. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34, 6 (2004), 2252–2261.

Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 169–178.

Rabia Nuray-Turan, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2012. Exploiting web querying for web people search. *ACM Transactions on Database Systems*, 37, 1 (2012), 7:1–7:41.

Thorsten Papenbrock, Arvid Heise, and Felix Naumann. 2015. Progressive duplicate detection. *IEEE Transactions on Knowledge and Data Engineering*, 99, 1 (2015), 1316–1329.

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Henry T. Reynolds and H. T. Reynolds. 1977. *The Analysis of Cross-classifications*. Free Press, New York.

Sunita Sarawagi and Anuradha Bhamidipaty. 2002. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 269–278.

Temple F. Smith and Michael S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147, 1 (1981), 195–197.

Norases Vesdapunt, Kedar Bellare, and Nilesh Dalvi. 2014. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment*, 7, 12 (2014), 1071–1082.

Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2012. CrowdER: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5, 11 (2012), 1483–1494.

Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. 2014. A sample-and-clean framework for fast and accurate query processing on dirty data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 469–480.

Steven Euijong Whang and Hector Garcia-Molina. 2012. Joint entity resolution. In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE'12)*. 294–305.

Steven Euijong Whang and Hector Garcia-Molina. 2014. Incremental entity resolution on rules and data. *The VLDB Journal*, 23, 1 (2014), 77–102.

Steven Euijong Whang, Peter Lofgren, and Hector Garcia-Molina. 2013a. Question selection for crowd entity resolution. *Proceedings of the VLDB Endowment*, 6, 6 (2013), 349–360.

Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. 2013b. Pay-as-you-go entity resolution. *IEEE Transactions on Knowledge and Data Engineering*, 25, 5 (2013), 1111–1124.

Mohamed Yakout, Ahmed K. Elmagarmid, Hazem Elmeleegy, Mourad Ouzzani, and Alan Qi. 2010. Behavior based record linkage. *Proceedings of the VLDB Endowment*, 3, 1–2 (2010), 439–448.

Received June 2016; revised September 2017; accepted October 2017