Progressive Approach to Relational Entity Resolution

Yasser Altowim Dmitri V. Kalashnikov Sharad Mehrotra

Department of Computer Science
University of California, Irvine

ABSTRACT

This paper proposes a progressive approach to entity resolution (ER) that allows users to explore a trade-off between the resolution cost and the achieved quality of the resolved data. In particular, our approach aims to produce the highest quality result given a constraint on the resolution budget, specified by the user. Our proposed method monitors and dynamically reassesses the resolution progress to determine which parts of the data should be resolved next and how they should be resolved. The comprehensive empirical evaluation of the proposed approach demonstrates its significant advantage in terms of efficiency over the traditional ER techniques for the given problem settings.

1. INTRODUCTION

Entity resolution (ER) is a well-known data quality challenge that arises when real-world objects are referred to using entities or descriptions that are not always unique identifier of the objects [3, 5, 8, 23, 24]. The task of ER is to determine which entities refer to the same real-world object. A typical ER process consists of two phases: (a) a pruning phase that uses strategies, such as *blocking* [14, 18], to partition the dataset into a set of (possibly overlapping) *blocks* such that entities in different blocks are unlikely to co-refer, and (b) a (potentially expensive) resolution phase that determines duplicate entities within each block.

Traditionally, the ER problem has been studied in the context of data warehousing as an offline pre-processing step prior to data analysis. Such an offline strategy is not suitable for many emerging applications that require (near) real-time analysis, especially if it involves big, fast, or streaming data [22]. To address the needs of such applications, instead of cleaning data fully, a preferred approach is to clean data progressively in a pay-as-you-go fashion, while continually/periodically analyzing the partially cleaned data to progressively compute better analysis results.

Such a progressive approach is a new paradigm for entity resolution. It aims to identify and resolve as many duplicate entities in the dataset as possible while trying to avoid resolving non-duplicate entities. For this purpose, a progressive ER approach prioritizes/ranks which parts of the data to resolve first in order to maximize the number of resolved duplicate entities.

The work on progressive data cleaning [9, 15, 22] is at an early stage with [22] being the first to explore it in the context of ER. In [22], the authors focus on maximizing the quality of the cleaned data given a user-defined resolution budget (e.g., 5 minutes). They propose several concrete ways of constructing resolution "hints" that can be then used by a variety of existing ER algorithms as a guidance for which entities to resolve first.

While [22] addresses the problem of progressive ER, their solution considers resolving only a single entity-set containing duplicates. In this paper, we study the problem in the case of *relational* ER [6, 11, 12, 23]. In relational ER, the dataset consists of multiple entity-sets and relationships among them. In such a dataset, a resolution of some entities might influence the resolution of other entities. For example, deciding that two research papers are the same may trigger the fact that their venues are also the same. Prior research [6, 11, 12] has shown that such decision propagation via relationships can significantly improve data quality in datasets where such relationships can be specified. Our challenge in this paper is to make such ER techniques that use decision propagation (e.g., [11, 12]) progressive.

The dependency between resolution decisions in relational ER influences 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 an adaptive progressive approach to relational ER that aims to generate a high-quality result using a limited resolution budget. To achieve adaptivity, our approach 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

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: A performance ...	...	{File System, Database}	$\{a_1\}$	u_2
	p_4	Berkeley DB: A Retrospective	...	{File System, Database}	$\{a_3\}$	u_4

(a) Entity-set Papers.

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	$\{p_3\}$

(b) Entity-set Authors.

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	$\{p_4\}$

(c) Entity-set Venues.

Table 1: Toy Publication Dataset.

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. Furthermore, our approach 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 the dataset, such a technique must be implemented carefully so that the benefits obtained are not overshadowed by the associated overheads of adaptation. We achieve this by designing efficient algorithms, appropriate data structures, and statistics gathering methods that do not impose high overheads. Our results in Section 7 show how our approach can generate a high-quality result using a limited amount of resolution budget.

Overall, the **main contributions** of this paper are:

- We propose an adaptive progressive approach to the problem of relational entity resolution (Section 3).
- We present a benefit and a 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 similarity functions can be exploited to reduce the resolution cost (Section 6).
- We experimentally evaluate our approach using publication and synthetic datasets and demonstrate the efficiency of the proposed solution (Section 7).

2. NOTATION AND PROBLEM DEFINITION

In the following subsections, we first develop notation, and then define the problem of progressive ER.

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 contains a set of entities of the same type $R = \{r_1, r_2, \dots, r_{|R|}\}$. 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. We denote the attributes in R as $R.A = \{R.a_1, R.a_2, \dots, R.a_{|R.A|}\}$. An attribute $R.a$ can be a *reference* attribute, i.e., its values are references to other entities in other entity-sets.

Table 1 shows an example of a relational publication dataset. It contains three entity-sets: papers P, authors A, and venues U. 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* [14,18] that partitions each entity-set into (possibly overlapping) smaller 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 entity-set to that of applying ER on the small blocks. In our approach, we also assume that each entity-set $R \in \mathcal{D}$ is partitioned into a set of blocks $\mathcal{B}_R = \{R_1, R_2, \dots, R_{|\mathcal{B}_R|}\}$[1].
- *Problem Representation* that maps the data into an internal data representation of the ER algorithm. We represent our problem as a graph, as detailed in Section 2.3.
- *Similarity Computation* that computes the similarity between entities in the same block. This phase is often computationally expensive as it might require resolving $\mathcal{O}(n^2)$ 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, ontology matching, and/or seeking human input through crowdsourcing [19,21].

  Correspondingly, in our approach, each entity-set $R \in \mathcal{D}$ is associated with multiple similarity functions $F_R = \{f_1^R, f_2^R, \dots,\ f_{|F_R|}^R\}$. These similarity functions are assumed to be "black-boxes" [5,8]. 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 from the $[0, 1]$ interval. This score is computed by f_i^R by analyzing the values of r_j and r_k in some of the non-reference attributes $\mathcal{A}(f_i^R)$ of R. For instance, function f_1^P in Table 2 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.

  Each similarity function f_i^R in turn is associated with a cost c_i^R, which represents an average cost of applying f_i^R

[1]For clarity, we will present the paper as if blocks do not overlap. Extending our approach to overlapping blocks is straightforward, see [1] for details. In fact, our experiments on the publication dataset use overlapping blocks.

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	JaroWinkler Distance
f_1^U	U.Name	Edit Distance

Table 2: Description of the Similarity Functions.

on a pair of entities. Our approach is agnostic to the way this cost is set. For instance, we set it as the average execution time of f_i^R learned from the dataset. But it also could be a unit-based cost set by the domain analysts, etc.

In the case where multiple similarity functions are used on an entity-set R, there could be a *resolve* function whose task is to combine the values of these multiple functions into a single value. We explain our resolve functions in Section 2.3.

- *Clustering* that groups duplicate entities into clusters based on the values computed by similarity/resolve functions such that each cluster represents one real-world object, and each real-world object is represented by one cluster [5,12,19].
- *Merging* that combines entities of each individual cluster into a single representative entity that will represent the cluster to the end-user or application in the final result.

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 entity attributes as in traditional ER, relational ER also utilizes the relationships among the entity-sets to further improve the quality of the result [6,12].

To illustrate, consider the paper pair $\langle p_1, p_3\rangle$ in Table 1. 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 1 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 1, 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. *Inf*(R) is the set of all $L_{R\to X}$ influences for any X, and *Dep*(S) is the set of all $L_{X\to S}$ influences for any X. In the example in Table 1, *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 [11,12].

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 represents 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 exits an influence $L_{R\to S}$ and the resolution of v_i can influence the resolution of v_l.

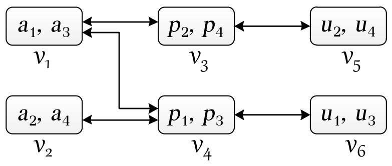


Figure 1: Graph Representation.

Figure 1 shows the graph corresponding to the publication dataset in Table 1. 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 $L_{A\to P}$. Therefore, node v_3 has one *influencing* node v_1 via $L_{A\to P}$. Node v_1 has two *dependent* nodes v_3 and v_4 via $L_{A\to P}$.

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 "black-box" *resolve* function on the node v_i.

Resolve Functions. 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* value $sim_i \in [0,1]$ and a *dissimilarity* 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 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 in itself 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 in $Dep(R)$. For example, consider node v_3 in Figure 1. Suppose that the outputs of applying the 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 above discussed model of resolve functions is generic to capture a wide range of possible implementations of resolve functions. In our experiments on the publication dataset, we implement the resolve function 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.

Developing an *optimal* progressive approach is infeasible in practice as it would require an "oracle" that knows in advance the set of pairs that, when resolved, will have 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 clean 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[2]. Second, it should resolve those identified pairs with the least amount of cost.

3. OVERVIEW OF OUR APPROACH

Our progressive ER approach resolves the dataset D by incrementally constructing the ER graph and resolving its nodes. At any instance of time, the approach 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 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 $\mathbf{RV}$, and *unresolved* nodes, denoted as $\mathbf{UV}$, based of 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 2 depicts an example of a graph G and a partially constructed graph G^p.

Resolution Windows. To incrementally build and resolve the graph G^P, we divide the total budget BG into several resolution windows. Each window consists of two phases: plan generation followed by plan execution. In the plan generation phase, our approach determines the activities that should be performed in 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; e.g., 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 our approach incrementally builds the graph G^p. If a node v_i is chosen to be resolved and it is not currently instantiated in G^p, then it is first instantiated. Instantiation is performed in unit of *blocks*; that is, if a node v_i needs to be instantiated in a window, then we instantiate 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, we also determine, 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 1, we determine 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 helps bring significant savings since the cost of reading the nodes (possibly from disk/storage) is only incurred once per block. Thus, prior to the beginning of a window, the blocks in $\mathcal{D}$ can be classified as *instantiated* blocks, which are the blocks that have been instantiated in previous windows, and *uninstantiated* blocks, denoted as $\mathbf{UB}$.

[2]We operate under the condition that most of the pairs in the dataset are distinct [22], which is often the case in real-world datasets.

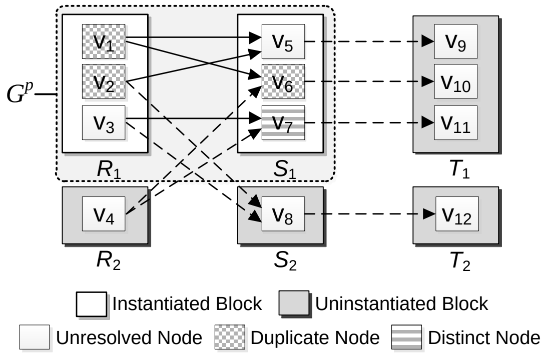


Figure 2: A Partially Constructed Graph G^p.

Plan Generation. During the plan generation phase of a window, our approach generates a resolution plan $\bar{\mathbf{P}}$ that specifies the set of blocks to be instantiated in the window, denoted as $\bar{\mathbf{P}}_{\mathrm{B}}$, and the set of nodes to be resolved in the window, denoted as $\bar{\mathbf{P}}_{\mathrm{V}}$. A resolution plan is said to be *valid* only if, for every uninstantiated node $v_i \in \bar{\mathbf{P}}_{\mathrm{V}}$, the block $BK(v_i) \in \bar{\mathbf{P}}_{\mathrm{B}}$. For example, suppose that the graph G^p at the beginning of the current window is as depicted in Figure 2. If $\bar{\mathbf{P}}_{\mathrm{V}} = \{v_3, v_8\}$ and $\bar{\mathbf{P}}_{\mathrm{B}}$ 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 $\bar{\mathbf{P}}$ is the summation of the *instantiation* cost of every block in $\bar{\mathbf{P}}_{\mathrm{B}}$ and the *resolution* cost of every node in $\bar{\mathbf{P}}_{\mathrm{V}}$. To estimate the resolution cost of a node v_i, we need to associate a *workflow* with that node which specifies how v_i is to be resolved (explained next). The benefit of a plan $\bar{\mathbf{P}}$ measures the value of resolving the nodes in $\bar{\mathbf{P}}_{\mathrm{V}}$. Note that instantiating a block $R_i \in \bar{\mathbf{P}}_{\mathrm{B}}$, 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 plan generation starts by ensuring that every block in $\mathbf{UB}$ is associated with an updated instantiation cost value and every node in $\mathbf{CV}$ 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 in a small amount of cost. The process of generating a valid resolution plan is explained in Section 5.

Plan Execution. The process of executing a plan $\bar{\mathbf{P}}$ starts by instantiating the blocks in $\bar{\mathbf{P}}_{\text{B}}$, and then iterates over all the nodes in $\bar{\mathbf{P}}_{\text{V}}$ to resolve them[3]. One naive strategy for resolving a node $v_i \in V(R)$ is to apply all the similarity functions of R on that node, and then call the resolve function $\mathfrak{R}_R$ on v_i to determine its resolution decision. However, in practice, it is often sufficient to apply a subset of these functions to resolve the node to a `certain` decision. Thus, our approach follows a *lazy resolution* strategy that delays the application of a similarity function on a node until it is needed. To resolve a node $v_i \in V(R)$ using this strategy, we apply the similarity functions of R on v_i in a specific order, referred to as the *workflow* of v_i (the details of how workflows are generated and associated with nodes are presented in Section 6). After the application of each similarity function, we call the resolve function $\mathfrak{R}_R$ on v_i to determine if it is `certain` or not. If the node is `uncertain`, we apply the next similarity function in the workflow on v_i, and then call $\mathfrak{R}_R$ on the node again. This process continues until the node is `certain`, or there is no more similarity function to apply on the node.

After resolving all nodes in $\bar{\mathbf{P}}_{\text{V}}$, their resolution decisions are propagated to their dependent nodes in G^P that are still uncertain. To do so, our approach stores those dependent nodes into a set H, and then iterates over them. For each node $v_j \in H$, our approach 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, we insert those dependent nodes into H only when v_j was resolved to a certain decision, or when the increase in sim_j or dis_j due to the last application of the resolve function on v_j exceeds a small constant.

Early Termination. On the completion of the budget BG, all uncertain and unresolved nodes in the graph G are declared to be `distinct`. Although there can be sophisticated approaches to making decisions about those nodes, such approaches add to the overall computational complexity. Since our approach finds duplicate entities early, declaring those nodes to be distinct is expected to perform well.

4. BENEFIT AND COST MODELS

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

4.1 Benefit Model

We will first describe how we compute the benefit of a node v_i and then show at the end of this section how we estimate 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 (i.e., which nodes have been resolved to duplicate so far), denoted as $\boldsymbol{\mathcal{S}}$. Therefore, the benefit of a node v_i can be determined based on whether v_i is duplicate or not (*direct benefit*) and 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 is defined as follows:

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

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 $\mathit{Imp}(v_i)$ is the impact of v_i which is defined as follows:

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

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 between the nodes (in contrast, we construct G progressively), and is also equivalent to the *belief update* problem in Bayesian Networks, which is known to NP-hard [10]. Second, identifying each of the Top and Top_{v_i} sets can be easily proved to be NP-hard as it contains the traditional *knapsack* problem as a special case. Therefore, we use heuristics to compute $\mathit{Benefit}(v_i)$. The values of $\mathcal{P}(v_i)$ and $\mathit{Imp}(v_i)$ are estimated as below.

Probability Estimation. We estimate the value of $\mathcal{P}(v_i)$ using the Noisy-Or model [20] 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 *causes* $\mathbf{X}^i = \{x_1^i, x_2^i, \dots, x_n^i\}$. These causes in our case correspond to the direct influencing nodes of v_i and the block $BK(v_i)$. That is, the probability 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[4], and therefore computes the conditional probability $\mathcal{P}(v_i) = \mathcal{P}(y_i|\mathbf{X}^i)$ as follows:

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

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 $L_{S\rightarrow R}$, then x_j^i

[3]The actual cost of executing a plan might be different from its estimated cost since we can never accurately determine the exact cost of instantiating a block or resolving a node.

[4]One could also use a more advanced model such as the *Recursive* Noisy-Or model [17] to allow for dependent causes to be used in estimating the probability of an effect. However, such dependency between causes is rarely observed in datasets and often requires significant effort to be identified.

is present only if v_k was resolved to duplicate in a previous window, and thus we set $\mathcal{P}(y_i|x^i_j)$ 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^i_j corresponds to the block $BK(v_i) = R_k$, then we set $\mathcal{P}(y_i|x^i_j)$ 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 previous windows, but we require that x^i_j 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 our approach, but rather helps improve the accuracy of estimating the probability value of nodes.

As an example, consider the graph in Figure 2. Suppose that $\mathcal{P}_{R\to S} = 0.4$ and $\alpha = 0.3$. Then, $\mathbf{X}^5_p$ consists of three causes that correspond to the nodes v_1 and v_2 and the block S_1, whereas $\mathbf{X}^8_p$ 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 can not be duplicate. To overcome this limitation, the *Leaky* Noisy-Or model [13] 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.

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:

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

where the probability values in Equation 4 are computed as in Bayesian Networks (not using our approximation model (Equation 3)). 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 (the parameter β is explained later).

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 $\mathcal{P}(v_j|\boldsymbol{\mathcal{S}}_{v_i})$ is in general an NP-hard problem [10]. Although there are approximation algorithms for this problem [16], such algorithms are still expensive for our real-time settings. Second, identifying $\mathcal{N}(v_i)$ might not be possible in certain cases unless the graph G is fully instantiated.

Thus, we instead compute a single estimated impact value, denoted as $Imp(R)$, for every entity-set R and use 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:

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

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 our approach proceeds forward.

```
COMPUTE-IMPACT($R, \beta$)
1  if $\beta = 0$ then
2      return 0
3  $imp \leftarrow 0$
4  $Visited \leftarrow \{R\}$
5  foreach $L_{R\to S} \in inf(R)$ do
6      $k \leftarrow U(1, L_{R\to S})$ // Equation 5
7      $\langle m, Visited\rangle \leftarrow$ COMPUTE-PROB-INC($R, S, k, Visited, \beta$)
8      $imp \leftarrow imp + m$
9  return $imp$

COMPUTE-PROB-INC($R, S, k, Visited, \beta$)
1  $Visited \leftarrow Visited \cup \{S\}$
2  $n \leftarrow \mathcal{P}_{R\to S} * k$
3  if $\beta - 1 = 0$ then
4      return $\langle n, Visited\rangle$
5  foreach $L_{S\to T} \in inf(S)$ s.t. $T \notin Visited$ do
6      $k \leftarrow U(k, L_{S\to T})$ // Equation 5
7      $\langle m, Visited\rangle \leftarrow$ COMPUTE-PROB-INC($S, T, k, Visited, \beta - 1$)
8      $n \leftarrow n + \mathcal{P}_{R\to S} * m$
9  return $\langle n, Visited\rangle$
```

Figure 3: Impact Computation.

Now, the impact value $Imp(R)$ can be estimated by calling the COMPUTE-IMPACT(.) function in Figure 3 with these values (R, β) (the value of β is explained next). As an example, let $\mathcal{D}$ correspond to the dataset whose ER graph is depicted in Figure 2. 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 we need to compute only a single impact value for each entity-set $R \in \mathcal{D}$) and works very well as we show in Section 7.

The parameter β has the same value for all entity-sets, but that value is updated throughout the execution. Intuitively, such a value 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, we update 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 2 consists of $|\mathcal{D}| = 3$ nodes (that correspond to the entity-sets R, S, and T) and two edges ($R \to S$ and $S \to T$), 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.

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

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

4.2 Cost Model

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

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

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 a local machine, on the cloud, or anywhere else. 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, we first need to estimate the value of $\mathcal{P}(v_j)$, use that value to associate a workflow with v_j, and then estimate 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, our approach first needs to perform a benefit-vs-cost analysis on the uninstantiated blocks and candidate nodes 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, our approach needs to generate a *valid* resolution plan $\bar{\mathbf{P}}$ such that $Cost(\bar{\mathbf{P}}) \leq W$, and $Benefit(\bar{\mathbf{P}})$ is maximized. Generating such a plan can be proved to be NP-hard [7]. Thus, we propose in Section 5.2 an efficient approximate solution for this problem that performs very well in practice.

5.1 Benefit-vs-Cost Analysis

The algorithm that performs the benefit-vs-cost analysis is illustrated in Figure 4.

Instantiation Cost. The algorithm starts by passing the blocks **UB** to the UPDATE-INST-COST(.) function 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

```
PERFORM-ANALYSIS($\mathbf{UV}, \mathbf{UB}, \mathcal{D}, L(\mathcal{D}), \beta$)
1  UPDATE-INST-COST($\mathbf{UB}$)
2  UPDATE-IMPACT($\mathcal{D}, L(\mathcal{D}), \beta$)
3  UPDATE-PROB-AND-COST($\mathbf{UV}$)
4  $\mathbf{AV} \leftarrow$ GET-AFFECTED-NODES($\mathbf{UB}$)
5  UPDATE-PROB-AND-COST($\mathbf{AV}$)
6  for $i \leftarrow 1$ to $|\mathbf{UB}|$ do
7      COMPUTE-SINGLE-PROB-AND-COST($\mathbf{UB}[i]$)
```

Figure 4: Benefit-vs-Cost Analysis.

can overlap. More details on when we should update these values can be found in [1].

Impact. The algorithm then calls the UPDATE-IMPACT(.) function to compute a single impact value for each entity-set $R \in \mathcal{D}$ using the COMPUTE-IMPACT(.) function in Figure 3.

Probability And Cost. Next, the algorithm passes the set of nodes **UV** to the UPDATE-PROB-AND-COST(.) function that updates the probability and resolution cost values of only the nodes whose values may have changed due to the resolution in the last window. That is, the probability value of a node $v_i \in \mathbf{UV}$ is updated only if (a) at least one of its influencing nodes has been resolved to duplicate in the previous window, or (b) some of the nodes resolved in the previous window belong to the block $BK(v_i)$ and that $BK(v_i)$ is a present cause in estimating the probability of v_i. Once the probability of a node $v_i \in \mathbf{UV}$ is updated, we use the new probability to reassociate a workflow with v_i and then recompute 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 need to 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 2 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, our approach 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 2, the set of counter values for v_8 consists of a single counter that corresponds to the influence $L_{R \to S}$ and whose value is 1.

Since the set of present causes of any unaffected uninstantiated node can not 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 our approach, 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 values, our approach does not store any information about the unaffected uninstantiated nodes of block $R_i \in \mathbf{UB}$.

```
GENERATE-PLAN($\mathbf{UV}, \mathbf{UB}, W, \mathcal{D}$)
1   BlockSet $\bar{\mathbf{P}}_{\mathrm{B}} \leftarrow \emptyset$
2   NodeSet $\bar{\mathbf{P}}_{\mathrm{V}} \leftarrow \emptyset$, $M \leftarrow \emptyset$
3   Double $max$, $t$
4   $\langle \bar{\mathbf{P}}_{\mathrm{V}}, max \rangle \leftarrow$ SELECT-NODES($\mathbf{UV}, W, \mathcal{D}$)
5   BlockList $List \leftarrow$ SORT-BLOCKS($\mathbf{UB}, \mathcal{D}$)
6   for $i \leftarrow 1$ to $|List|$ do
7       Block $B \leftarrow List[i]$
8       $\langle M, t \rangle \leftarrow$ SELECT-NODES($\bar{\mathbf{P}}_{\mathrm{V}} \cup V(B)$, $W - C^{ins}(B), \mathcal{D}$)
9       if $t > max$ then
10          $\bar{\mathbf{P}}_{\mathrm{B}} \leftarrow \bar{\mathbf{P}}_{\mathrm{B}} \cup \{B\}$
11          $\bar{\mathbf{P}}_{\mathrm{V}} \leftarrow M$
12          $max \leftarrow t$
13          $W \leftarrow W - C^{ins}(B)$
14      else
15          break
16  return $\langle \bar{\mathbf{P}}_{\mathrm{B}}, \bar{\mathbf{P}}_{\mathrm{V}} \rangle$
```

Figure 5: Generating A Valid Resolution Plan.

5.2 Algorithm

Our algorithm that generates a valid resolution plan is illustrated in Figure 5. 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 $\mathbf{UV}$. 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 their 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. Then, the algorithm sorts the blocks in $\mathbf{UB}$ in a non-increasing order based on their 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:

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

Then, the function sorts the blocks in a non-increasing order based on their usefulness values and 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 8-9). If so, the algorithm adds the block to the set of selected blocks $\bar{\mathbf{P}}_{\mathrm{B}}$, and updates the values of the set $\bar{\mathbf{P}}_{\mathrm{V}}$ and the other helper variables (Lines 10-13). Otherwise, it exists the *while* loop. Finally, the algorithm returns the sets $\bar{\mathbf{P}}_{\mathrm{B}}$ and $\bar{\mathbf{P}}_{\mathrm{V}}$ (Line 16).

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 2. 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 dependency 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 set of output 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 benefit of nodes are restricted to their direct benefit, i.e., their probability values.

Therefore, we do not account for such dependency 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 decreasing 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 the 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, our approach 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 may require that our approach explores several blocks in the initial few windows. To address this requirement, the approach needs to employ a different strategy for generating a resolution plan. Using the algorithm in Figure 5 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, the algorithm in Figure 5 will not consider instantiating the second block unless the budget W is sufficient for instantiating and resolving all nodes of the first block. However, our approach may want to instantiate several blocks in each of the initial few windows and explore those blocks by resolving a few nodes from each of them.

The plan generation algorithm that we use in each of these initial windows is a modification of the algorithm shown in Figure 5. It first starts by sorting the blocks in $\mathbf{UB}$ 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, denoted as $C^r(R_i, k)$, 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 $\bar{\mathbf{P}}_{\mathrm{B}}$ and $\bar{\mathbf{P}}_{\mathrm{V}}$ respectively, and then updates the window budget accordingly; i.e., $W = W - C^i(R_i) - C^r(R_i, k)$. 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 $\bar{\mathbf{P}}_{\mathrm{B}}$ and adds them to the set $\bar{\mathbf{P}}_{\mathrm{V}}$ to fill the budget W. This algorithm can be viewed as an initialization step of our approach and therefore be employed in the initial few windows.

6. WORKFLOWS

Given $|F_R|$ similarity functions associated with entity-set R, there are $|F_R|!$ different orders of function application that could be employed to resolve a node $v_i \in V(R)$. We, 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 this section, we first define our concept of the contribution of similarity functions, then describe how workflows are generated and associated with nodes, and finally show how we can estimate the resolution cost of a node given its associated workflow.

Function Contribution. In order to resolve a node v_i, we need 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 (negative) if it will increase the chance that the resolve function will return 1 as the similarity (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 terms of the amount of evidence that they provide to the resolution decision. Therefore, 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 provide *w.r.t.* the function $\mathfrak{R}_R$. Hence, 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 contribution of a function is the amount of positive evidence that the function is expected to provide when it is applied on a *duplicate* node. Similarly, the negative contribution of a function is the amount of negative evidence that the function is expected to provide when it is applied on a *distinct* node.

Workflow Generation. The process of generating a workflow for a node $v_i \in V(R)$ proceeds as follows. First, we compute for each similarity function $f_j^R \in F_R$ a utility value as follows: $[\theta * t_j^{R+} + (1-\theta) * t_j^{R-}]/c_j^R$, where θ is set to $\mathcal{P}(v_i)$. Then, we sort the functions in F_R in a non-increasing order based on their utility values. This order of functions is the workflow of v_i. This sorting will place the functions with the highest contribution per unit cost of resolution first in the workflow, maximizing the chance of resolving v_i to a `certain` decision with the least amount of cost.

Workflow Association Strategy. As explained in Section 5, to estimate 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 is to instantly generate a workflow for v_i as discussed above. Such a strategy can be inefficient as we will have to sort the similarity functions every time we need to estimate the resolution cost of a node. Thus, we follow a more efficient strategy for associating a workflow with a node. This strategy requires that we maintain a set of w pre-generated workflows $W_1^R, W_2^R, \ldots, W_w^R$ 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 $\frac{k-1}{w-1}$. For example, if $w = 5$, then the θ values that we use to generate these w workflows will be: $0, 0.25, 0.5, 0.75$, and 1.0 respectively. Now, to associate a workflow with a node $v_i \in V(R)$, we can simply map the node v_i to the workflow W_k^R whose θ value, i.e., $\frac{k-1}{w-1}$, is the closet (among all the workflows' of R) to the value of $\mathcal{P}(v_i)$.

Resolution Cost. Given a node $v_i \in V(R)$ and its workflow W_k^R, we estimate the value of $C^{res}(v_i)$ as follows:

$$C^{res}(v_i) = \mathcal{P}(v_i) * C^{r+}(W_k^R) + (1 - \mathcal{P}(v_i)) * C^{r-}(W_k^R) \quad (9)$$

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

7. EXPERIMENTAL EVALUATION

In this section, we evaluate the quality and efficiency of our approach on publication and synthetic datasets.

7.1 Experimental Setup

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 $L_{R\to S} \in Inf(R)$[5]. 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:

$$C^{ins}(R_i) = C^f(R_i) + |V(R_i)| * cc + \sum_{L_{R\to S} \in Inf(R)} C^b(S) \quad (10)$$

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.

Algorithms. In our experiments, we compare our solution with the *DepGraph* algorithm proposed in [12]. Although this algorithm is not a progressive solution, 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. In addition, we compare our solution with three variants of our approach. The first one, referred to as *Static*, differs from our approach only in how the SORT-BLOCKS(.) function in Figure 5 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 [23]. For example, if $Inf(R) = \{L_{R\to T}\}$, $Inf(S) = \emptyset$, and

[5]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 in Table 1).

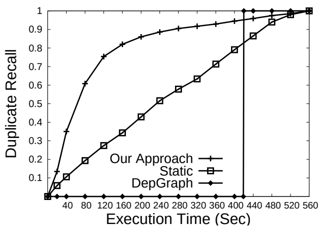
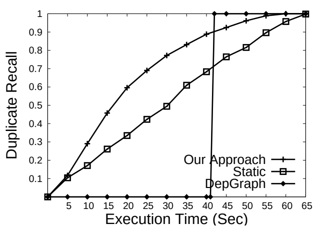
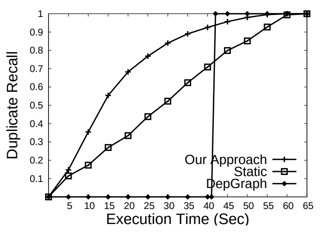


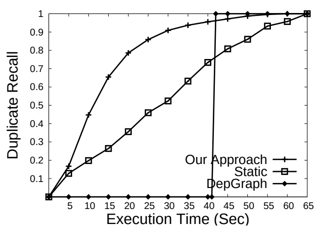
Figure 6: Time vs. Recall.



(a) $z = 0$.



(b) $z = 0.15$.



(c) $z = 0.3$.

Figure 7: Effects of Zipfian Distribution Exponent Value.

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. The second variant, referred to as *All*, 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 third variant, referred to as *Random*, uses the lazy resolution strategy but applies the similarity functions on a node in a random order.

Quality Metric. Since the goal of our approach is to find and resolve as many duplicate pairs as possible using the budget BG, we use the duplicate pairwise recall as our quality metric. Duplicate 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 our approach always achieves more than 0.99 precision.

7.2 Publication Dataset Experiments

In this section, we evaluate the efficacy of our approach on a real publication dataset called CiteSeerX [2]. 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 Venue (Name, Year, Pages, Papers). The cardinalities of the resultant entity-sets are $|P| = 30,000$, $|A| = 83,152$ and $|U| = 30,000$. We computed the ground truth by simply running the *DepGraph* algorithm on the obtained dataset.

Each entity-set is divided into a set of blocks. We use 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 use 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 use 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.

For this dataset, we use the same set of similarity functions given in Table 2 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.

Experiment 1 (Cost-vs-Recall). Figure 6 plots the duplicate recall as a function of the resolution cost (measured as the end-to-end execution time) for various ER algorithms. In the *Static* approach, P's blocks appear first in the sorted list of blocks, then A's blocks, and finally U's blocks. In this experiment, we do not study the benefit of using the lazy resolution strategy in resolving nodes. Therefore, when resolving a pair of entities, all algorithms in Figure 6 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 our approach, we ran the approach with different budget values (correspond to the points on the curve). For each budget value, we ran the approach ten times, recorded the achieved recall of each run, and then reported the average recall of the ten recall values. The curve of the *Static* approach is plotted in the same way as we plotted the curve of our approach. For the *DepGraph* approach, we ran it to completion ten times, recorded the completion time of each run, and then took the average completion time of the ten 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 how our approach can achieve high duplicate recall values using limited budget values. The performance gap between our approach 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 approach 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 resolution.

Experiment 2 (Lazy Resolution and Execution Time Phases). This experiment studies the benefit of resolving the nodes using the lazy resolution strategy with workflows.

	Our Approach	**Random**	**All**
Execution Time (sec)	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 3: Phases of Execution Time.

To conduct this experiment, we ran each of the three approaches shown in Table 3 ten 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 (the first row in Table 3) along with the breakdown of that execution 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 3. For example, we ran our approach with $BG = 310$ seconds ten times, and found that on average our approach finishes in 300.33 seconds and that the plan generation process (Section 5) takes on average 4.76% of the total exact 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 graph, and 82.01% for resolving the nodes). The achieved recall of each approach using the specified time 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 our approach 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., [19, 21], and therefore the overhead of generating resolution plans in such cases can be even lower.

Parameter	n	s	b	d	z	l	k
Value	4	20,000	100	0.2	0.15	0.3	2

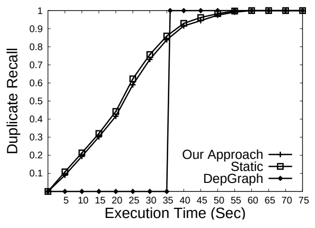
Table 4: Parameters of Synthetic Datasets.

7.3 Synthetic Dataset Experiments

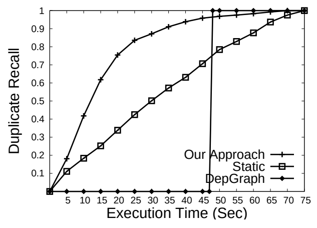
In order to evaluate our approach in a wider range of various scenarios, we built a synthetic dataset generator that allows us to generate datasets with different characteristics. The parameters that this generator takes as input along with their default values are shown in Table 4.

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 $L_{R \to S} \in \mathit{Inf}(R)$. 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



(a) $l = 0$.



(b) $l = 0.6$.

Figure 8: Effects of the Number of Influences.

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).

Experiment 3 (Effects of Duplicate Distribution). In this experiment, we study the performance of various ER algorithms when varying the value of the 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. Therefore, the higher the value of z is, the smaller the number of blocks that have the highest influence on the duplicate recall. In Figure 7, we vary the value of z from 0 to 0.3. Each figure in Figure 7 was plotted in the same way as we plotted Figure 6. As shown in these figures, the higher the value of z is, the better our approach performs compared to the other algorithms.

This experiment demonstrates the following. First, our approach performs well even when the resolution cost is relatively cheap (involves applying a single similarity function on two twenty-five character long string values). Second, our approach can adapt itself to datasets with various duplicate distributions and therefore quickly identify and resolve the blocks with high duplicate percentage values. Third, our approach is more effective when the duplicate pairs are not uniformly distributed across the blocks which is almost always the case in real-world datsets.

Experiment 4 (Effects of Influences). In this experiment, we study the effects of the number of influences on the performance of various ER approaches. In Figure 8, we vary the value of l from 0 to 0.6. The result when $l = 0.3$ is plotted in Figure 7(b). Each figure in Figure 8 was plotted in the same way as we plotted Figure 6.

As expected, when $l = 0$, our approach behaves very similarly to the *Static* approach because there exists no influences that our approach can utilize to identify which blocks to load and which nodes to resolve next. In fact, the *Static* approach performs slightly better than our approach 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, our approach 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 towards 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.

8. RELATED WORK

Entity resolution is a well-known data quality problem and has received significant attention in the literature over the past few decades, e.g., [5, 8, 23, 24]. Most prior work in this area has focused on improving either the efficiency of the ER algorithms [14, 18] or the quality of their results [4, 6, 24]. However, only a few research efforts have considered the problem of exploring a trade-off between the quality of the result and the resolution cost [9, 15, 22].

The most related work to our paper is that of [22]. As stated in Section 1, the context/type of datasets for which the two approaches have been developed differs. In fact, this difference raises an interesting question. To resolve a relational dataset $\mathcal{D}$ using a limited budget, should we use our approach, or should we resolve each entity-set in isolation using a single entity-set ER algorithm (that does not exploit the relationships in $\mathcal{D}$) with the help of one of the hints proposed in [22]? Our approach is intended for situations where exploiting those relationships is important in 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 our approach may not provide any significant advantage over [22]. A full characterization of under what circumstances one should choose which approach is an interesting direction of future work.

Furthermore, the problem of relational entity resolution has been studied in the literature. Reference [11] proposes a probabilistic model that uses Conditional Random Fields (CRF) to capture the dependencies between different entity-sets. In [12], 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 [23] propose a joint ER framework for resolving multiple datasets in a parallel fashion using custom ER algorithms. This proposed framework is not designed to be a progressive solution; 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 amount of data quality issues in these blocks.

9. CONCLUSIONS

In this paper, we have proposed a progressive approach to relational ER wherein the input dataset is resolved using only a limited budget with the aim of maximizing the quality of the result. Our approach 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 how our approach can quickly identify and resolve the duplicate pairs in the dataset, and thus generate a high quality result using limited amounts of resolution budget.

10. ACKNOWLEDGMENTS

This work was supported in part by NSF grants III-1429922, III-1118114, CNS-1059436, and by DHS grant 206-000070. Yasser Altowim was supported by KACST's Graduate Studies Scholarship.

11. REFERENCES

[1] `http://ics.uci.edu/~yaltowim/ProgER.pdf`.

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

[3] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. In *VLDB*, pp. 1846–1857, 2013.

[4] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pp. 586–597, 2002.

[5] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, pp. 255–276, 2009.

[6] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, pp. 1–36, 2007.

[7] J. J. Burg, J. Ainsworth, B. Casto, and S.-D. Lang. Experiments with the oregon trail knapsack problem. *Electronic Notes in Discrete Mathematics*, 1:26–35, 1999.

[8] Z. Chen, D. V. Kalashnikov, and S. Mehrotra. Exploiting context analysis for combining multiple entity resolution systems. In *SIGMOD*, pp. 207–218, 2009.

[9] R. Cheng, E. Lo, X. S. Yang, M.-H. Luk, X. Li, and X. Xie. Explore or exploit?: effective strategies for disambiguating large databases. In *VLDB*, pp. 815–825, 2010.

[10] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

[11] A. Culotta and A. McCallum. Joint deduplication of multiple record types in relational data. In *CIKM*, pp. 257–258, 2005.

[12] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pp. 85–96, 2005.

[13] M. Henrion. Practical issues in constructing a bayes' belief network. In *UAI*, pp. 132–139, 1987.

[14] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pp. 127–138, 1995.

[15] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, pp. 847–860, 2008.

[16] F. V. Jensen and T. D. Nielsen. *Bayesian networks and decision graphs*. Springer, 2007.

[17] J. F. Lemmer and D. E. Gossink. Recursive noisy or - a rule for estimating complex probabilistic interactions. *Trans. Sys. Man Cyber. Part B*, pp. 2252–2261, 2004.

[18] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, pp. 169–178, 2000.

[19] R. Nuray-Turan, D. V. Kalashnikov, and S. Mehrotra. Exploiting web querying for web people search. In *TODS*, pp. 7:1–7:41, 2012.

[20] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[21] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. In *VLDB*, pp. 1483–1494, 2012.

[22] S. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. In *TKDE*, pp. 1111–1124, 2013.

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

[24] M. Yakout, A. K. Elmagarmid, H. Elmelegy, M. Ouzzani, and A. Qi. Behavior based record linkage. In *VLDB*, pp. 439–448, 2010.