Linking Entities across Relations and Graphs

Wenfei Fan[1,2], Liang Geng[3] Ruochun Jin[1], Ping Lu[2], Resul Tugay[1], Wenyuan Yu[3]
University of Edinburgh[1] Shenzhen Institute of Computing Sciences[2] Alibaba Group[3]
{wenfei@inf., ruochun.jin@, resul.tugay@}ed.ac.uk, luping5303@gmail.com, {guanyi.gl, wenyuan.ywy}@alibaba-inc.com

***Abstract*—This paper proposes a notion of parametric simulation to link entities across a relational database $\mathcal{D}$ and a graph G. Taking functions and thresholds for measuring vertex closeness, path associations and important properties as parameters, parametric simulation identifies tuples t in $\mathcal{D}$ and vertices v in G that refer to the same real-world entity, based on topological and semantic matching. We develop machine learning methods to learn the parameter functions and thresholds. We show that parametric simulation is in quadratic-time, by providing such an algorithm. Putting these together, we develop HER, a parallel system to check whether (t, v) makes a match, find all vertex matches of t in G, and compute all matches across $\mathcal{D}$ and G, all in quadratic-time. Using real-life and synthetic data, we empirically verify that HER is accurate with F-measure of 0.94 on average, and is able to scale with database $\mathcal{D}$ and graph G.**

I. Introduction

Consider a relational database $\mathcal{D}$ and a graph G from different sources. Is it possible to determine whether a tuple t in $\mathcal{D}$ and a vertex v in G refer to the same real-world entity?

The need for studying this is evident. While most business data resides in relational databases, it is increasingly common to find graph-structured data, *e.g.,* transaction graphs, knowledge bases and social networks. It is often necessary to correlate the data from different sources for extracting, integrating and querying data in, *e.g.,* data lakes [63].

However, it is hard to correlate data in relations and graphs. It is "a longstanding challenge to the data management community" [39]. Unlike relational databases, real-life graphs may not have a schema, and typically denote entities as vertices. Even in the same graph, entities of the same "type" may have different topological structures, and their properties are often linked via paths, rather than annotated as direct attributes.

Example 1: Consider an enterprise procurement order placed at an e-commerce company A. It contains the quantities and specifications of the ordered items, along with information on suppliers, brands, logistic, etc. While the formats of such orders may vary across enterprises, the orders can be uniformly expressed as relations, *e.g.,* Tables I and II. As shown in Fig. 1, company A maintains a knowledge graph G for the items it carries. Consider the following three scenarios.

(1) Given ordered item t_1 of Table I, company A wants to check whether it is the item represented by vertex v_1 in graph G of Fig. 1. This is nontrivial. The specification of t_1 comes from catalogs/websites of suppliers, which may differ from the information collected in G. Indeed, t_1 and v_1 have different "topological structures", *e.g.,* "Dame Basketball Shoes D7" is the value of **item** attribute of t_1, while in G, it is represented by two vertices v_0 "Dame Basketball Shoes" and v_8 "Dame Gen 7". Moreover, an attribute in a tuple may be encoded by a path in G, *e.g.,* the **made_in** attribute "Can Duoc, VN" of tuple b_1 maps to a path (v_{15}, v_{19}, v_9) in G, bearing edge labels *factorySite*, *isIn* and *isIn*. Worse still, the attribute and the edge labels on the path may not seem closely related.

(2) For item "Dame Basketball Shoes D7", the procurement managers want to find all matching items supplied by company A, and buy the most cost effective one. This requires company A to search the entire graph G to find the matches.

(3) To fulfill the order, company A needs to find all matches from G of all the items that enterprise intends to order.

Cross checking also happens once a period of time, when company A searches all matches across vertices in graph G and all tuples from past orders collected in a large dataset $\mathcal{D}$, to accumulate information about items and orders, and improve the performance of its item recommendation [49]. □

Contributions & organization. We make an effort to link entities across relations and graphs based on their semantics.

(1) System (Section II). We develop a system, denoted by HER (Heterogeneous Entity Resolution), for linking entities in a relational database $\mathcal{D}$ and a graph G. It converts $\mathcal{D}$ to a canonical graph G_D using W3C standard RDB2RDF [86], and supports three modes. (a) SPair: users may enter pair (t, v) of a tuple $t \in \mathcal{D}$ and a vertex $v \in G$. HER checks whether t and v make *a match*, *i.e.,* they refer to the same entity. (b) VPair: users may ask for all vertices in G that match a given tuple $t \in \mathcal{D}$. (c) APair: one may also request HER to find all matches across $\mathcal{D}$ and G. These modes correspond to cases (1)–(3) of Example 1. In particular, VPair conducts real-time analysis as in, *e.g.,* [88], and APair is needed in fine-grained advertising [90].

(2) A new notion (Section III). Underlying HER is a notion of parametric simulation. Given G_D and G, it determines whether a vertex u_t in G_D (denoting a tuple t in $\mathcal{D}$) matches a vertex v_g in G. Since G_D and G may have radically different topological structures, it may not suffice to inspect only local features of u_t and v_g. Hence parametric simulation recursively checks the pairwise semantic closeness of descendants of u_t and v_g, by embedding machine learning (ML) in topological matching.

More specifically, parametric simulation is inductively defined to match (u_t, v_g) and their descendants. It maps paths in G_D to paths in G, to accommodate the semistructured nature of graphs. It is parameterized by score functions to assess the closeness of (a) vertices, (b) properties (descendants linked via paths) of vertices, and (c) associations of pairwise matching descendants of u_t and v_g. It decides that u_t and v_g match only if an aggregate score is above predefined bounds.

(3) Learning parameters (Section IV). As parameters, we

	item	material	color	type	brand	qty
t_1	Dame Basketball Shoes D7	phylon foam	white	Dame 7	b_1	500
t_2	Lightweight Running Shoes	synthetic	red	DD8505	b_1	100
t_3	Mid-cut Basketball Shoes Ultra Comfortable	phylon foam	red	*null*	b_2	200

TABLE I
RELATION item

	name	country	manufacturer	made_in
b_1	Addidas Originals	Germany	Addidas AG	Can Duoc, VN
b_2	Addidas	Germany	Addidas AG	Long An, Vietnam

TABLE II
RELATION brand

define the score functions with BERT-based embedding and metric learning models [73], [30], to quantify the semantic similarity between labels. We select top-k "properties" of a vertex via Long Short Term Memory (LSTM) network [58] for a bound k. Bounds are decided by random search [19], a trade-off between efficiency and accuracy. Moreover, HER interacts with users to improve the parameter functions with feedback, which employs triplet loss function [75] and majority voting to make the fine-tuning robust.

(4) Complexity and algorithm (Section V). We show that parametric simulation takes quadratic time, as opposed to the intractability of graph homomorphism and subgraph isomorphism (cf. [38]). To show this, we develop a quadratic-time algorithm to determine whether a pair (u_t, v_g) makes a match.

(5) Parallel algorithms (Section VI). We also give algorithms to compute all matches of a tuple t and all pairs of matches across G_D (*i.e.,* $\mathcal{D}$) and G with parametric simulation. We show that both algorithms are in quadratic-time. We parallelize these algorithms to scale with large databases and graphs.

(6) Empirical study (Section VII). Experimenting with six real-life datasets and synthetic data, we find the following. On average, (a) HER has F-measure= 0.94; it is 23.3% and 41.6% more accurate than ML-based [36], [48], [62] and rule-based [69] methods alone, respectively. (b) It is also 118 and 6 times faster than ML-based and rule-based methods, respectively. It takes 0.68ms to check whether a pair is a match across databases $\mathcal{D}$ of 52M tuples and graphs G of 202M vertices and edges with a single machine, and 107s to find all matches across $\mathcal{D}$ and G using 16 machines. (c) It scales well with the number n of processors. It is on average 3.2 times faster when n varies from 4 to 16. These verify that integrating topological matching and ML models is promising for entity linking.

Related work. We categorize the related work as follows.

Entity resolution (ER). ER has primarily been studied for relational data that is specified by a schema (see [25] for a survey). The state-of-the-art relational ER systems employ (1) ML models, *e.g.,* [50], [22] exploit graph embedding for ER; and [48], [80], [62] propose ML-based ER architectures; (2) transfer and active learning, *e.g.,* [96], [47], learn ER models that are transferable to new tasks; [71] learns ER rules via active learning; [42], [24] employ active learning to adaptively process ER workloads. Different from previous work, (a) we study ER across relations and graphs, beyond merely on relational data (*e.g.,* [42], [22], [48]); and (b) "globally" assess

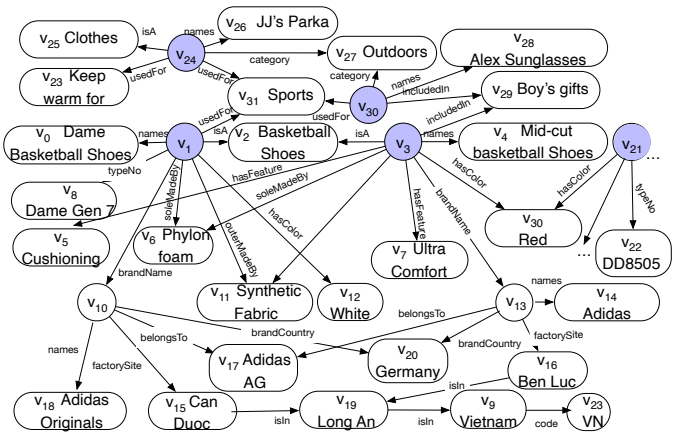

Fig. 1. An e-commerce knowledge graph

the entity semantic closeness by recursively inspecting properties (descendants), while previous work considers only local attributes or features of a single entity (*e.g.,* [24], [96], [80]).

ER has also been studied for graphs, based on (1) *rules*, *e.g.,* keys [32] and graph differential dependencies [52]; (2) *ML models*, *e.g.,* [74] adopts unsupervised clustering and matches vertices in the same cluster; [81] employs deep neural networks; [31] and [97] make use of vertex embedding based on heterogeneous skip-gram and co-occurrence; [89] encodes edge information into graph embedding via interactions; [94] exploits multi-view embedding and cross-data training; [64] exploits triangle inequality for blocking, and conducts metric-based ER; [43] aggregates value similarities via link conditions; [44], [92], [51], [87] compute SimRank scores for vertex matching; [77] aligns ontology via probability estimation.

There has also been work on heterogeneous ER [67], [69], [53], [95], [79], [37]. However, their notions of heterogeneity are quite different from ours. In [53], it means the (relational) schema heterogeneity. [79] and [37] target heterogeneous networks with multiple typed objects and links. In [67], it comes from loose schema binding. JedAI [69] considers various data formats, *e.g.,* RDF and CSV, by converting input entities into a set of profiles in the form of name-value pairs, and then checking labels and attributes as in [64]. While [95] links entities in Web tables and knowledge bases, it takes only local information (*e.g.,* edit distance and vertex description) as features.

Unfortunately, none of the prior methods (*e.g.,* [89], [94], [77]) works well across relations $\mathcal{D}$ and graphs G. Relational ER methods rely on schema information, and do not apply to schema-agnostic graphs. In particular, entities are denoted as vertices v in G, and its properties are linked from v via paths. Relational methods do not explore such properties. To extend these methods, one has to use joins to traverse paths and incur cost way beyond quadratic time. Moreover, prior methods explore only local properties, *e.g.,* "local embeddings" [22], [89] collect local information of neighbors within limited hops. However, to accurately identify a tuple t in $\mathcal{D}$ and a vertex v in G, one has to recursively check the pairwise semantic closeness of important descendants of t and v.

Related to this work are also [82], [66], which map a cell in a table to an entity in a knowledge base. However, their settings differ from ours as follows: (1) we match tuples in

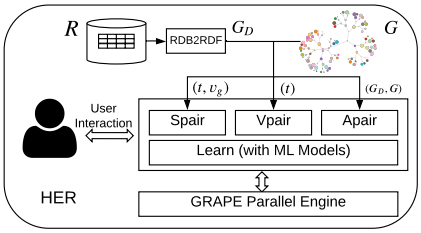

Fig. 2. HER Architecture

database to vertices in graphs, rather than cells to vertices as in SemTab; and (2) SemTab employ external query APIs or other information from the Web, while we do not assume these.

This work differs from the prior work as follows. (a) We study *disjoint* graph G and (canonical graph representation of) database $\mathcal{D}$, which are essentially heterogeneous. (b) We propose parametric simulation that is beyond conventional ML methods, and cannot be expressed as existing rules, since (i) parametric simulation is recursively defined, beyond the expressive power of first-order logic, and (ii) it "globally" assesses the semantic closeness by recursively inspecting properties (descendants), as opposed to checking attributes and close neighbors of vertices as ML models. (c) Matches found by parametric simulation are explainable, showing why two vertices match based on matching vertex pairs and the accumulated score. In contrast, embedding-based methods (*e.g.,* [89] and [94]) just train black-box model without explanation.

Graph simulation. Proposed for program analysis [61], this notion has been extended to map edges to paths, *e.g.,* bounded [33] and strong simulation [57]. Other notions for graph matching, *e.g.,* subgraph isomorphism and homomorphism, are too strong to match entities with different topological structures; worse yet, they incur intractability (cf. [38]).

As will be seen in Section III, parametric simulation radically differs from graph (bounded, strong) simulation as follows. (1) It is parameterized with score functions and closeness thresholds learned via ML models. Neither (aggregate) scores nor ML models are used in (bounded, strong) simulation. (2) It may map paths in one graph to paths in another. It does not require every edge of u to find a match in G, to cope with schemaless graphs in which missing links are common.

II. Heterogeneous Entity Resolution

We next outline HER (Heterogeneous Entity Resolution).

Preliminaries. We start with a review of basic notations. Assume three infinite alphabets Υ, Θ and Φ, for relation attributes, graph vertex labels and edge labels, respectively.

Relational databases. Consider a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, where R_i is a relation schema $(A_1, \ldots, A_k)$, and $A_i \in \Upsilon$ is an attribute. A *relation of schema* R is a set of tuples with attributes A_i of R ($i \in [1, k]$). A *database* $\mathcal{D}$ *of* $\mathcal{R}$ is $(D_1, \ldots, D_n)$, where D_i is a relation of R_i for $i \in [1, n]$.

Graphs. We consider *directed labeled graphs* $G = (V, E, L)$, where (a) V is a finite set of vertices, (b) $E \subseteq V \times V$ is a set of edges, and (c) for each vertex $v \in V$ (resp. edge $e \in E$), $L(v)$ (resp. $L(e)$) is a label in Θ (resp. Φ). The graphs encode attributes (properties) as edges, like in RDF.

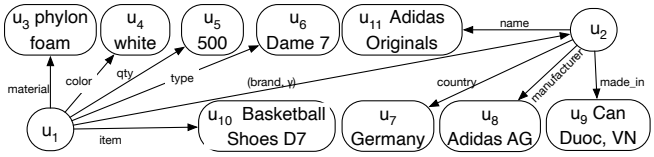

Fig. 3. Graph representation of t_1 and b_1 in $\mathcal{D}$

Intuitively, edge labels of Φ typify predicates, and vertex labels of Θ represent values. As will be seen in Section IV, we treat labels of Φ and Θ with different ML models.

Architecture. As shown in Fig. 2, HER operates on a database $\mathcal{D}$ of schema $\mathcal{R}$ and a graph G. It consists of five modules.

(1) RDB2RDF. This module converts $\mathcal{D}$ to a canonical graph G_D offline by, *e.g.,* direct mapping of RDB2RDF [86], which yields an 1-1 mapping f_D from the tuples and their attributes in $\mathcal{D}$ to the vertices and their edges in G_D, respectively.

(2) Learn. It learns score functions and bounds, *i.e.,* parameters for parametric simulation. It also interacts with users to inspect the matches, and improves the bounds based on feedback.

After these, users may issue requests in one of three modes.

(3) SPair. In this mode, users iteratively provide pairs (t, v_g) for tuples t in $\mathcal{D}$ and vertices v_g in G. Given (t, v_g), module SPair first finds the vertex u_t of G_D denoting t, via mapping f_D. It then checks whether (u_t, v_g) makes a match via parametric simulation. It returns true if so, and false otherwise.

(4) VPair. Users may enter a single tuple t. Module VPair finds all pairs (t, v_g) for all vertices $v_g \in G$ such that (u_t, v_g) is a match, where u_t is the vertex in G_D that denotes tuple t.

(5) APair. Alternative, users may request to find all pairs (t, v) that make matches for all tuples $t \in \mathcal{D}$ and vertices $v \in G$.

Here SPair, VPair and APair compute matches based on parametric simulation, using the learned parameters. We will define parametric simulation in Section III, learn parameters in Section IV, and develop algorithms underlying SPair, VPair and APair in Sections V and VI. The algorithms run on top of GRAPE [35], [9], an open-source parallel graph engine.

RDB2RDF. We next present RDB2RDF. Several methods are in place for converting relations to graphs, *e.g.,* [59]. Here we take RDB2RDF [85] for simplicity; HER allows user to plug in other methods for representing relations as graphs.

Following direct mapping rules of RDB2RDF [20], for a database schema $\mathcal{R}$, we define a *canonical mapping* f_D. Given a database $\mathcal{D}$ of $\mathcal{R}$, it returns a *canonical graph* $G_D = f_D(\mathcal{D})$ in which (1) each tuple t of relation schema R is mapped to a unique vertex u_t in G_D labeled R; (2) each attribute A in t is mapped to a unique vertex $u_{t.A}$ such that $L(u_{t.A})$ is the value of $t.A$ and there is an edge $(u_t, u_{t.A})$ with label A in G_D; and (3) for each attribute A of a foreign key in tuple t referencing another tuple t', there exists an edge $(u_t, u_{t'})$ with a pair (A, γ) of labels, where distinct γ indicates foreign key.

Example 2: Figure 3 shows the canonical graph G_D converted by canonical mapping f_D from tuples t_1 and b_1 in $\mathcal{D}$, *i.e.,* f_D maps t_1 and b_1 to vertices u_1 and u_2 in G_D, respectively, and the foreign key is mapped to an edge from

u_1 to u_2. Each attribute is mapped to a vertex with an edge from u_1 or u_2, where the vertex label is its value and the edge label is its name; *e.g.,* "phylon foam" in t_1 is mapped to u_3 and attribute "**material**" is the edge label (see Fig. 3). □

III. Parametric Simulation

We next introduce the notion of parametric simulation. Given two graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$, it is to identify their vertices that refer to the same entity.

Paths. We use the following notations.

A *path* ρ *from a vertex* v_0 in G is a list $\rho = (v_0, v_1, \ldots, v_l)$ such that (v_{i-1}, v_i) is an edge in G for $i \in [1, l]$. The *length of* ρ, denoted by $\mathsf{len}(\rho)$, is l, *i.e.,* the number of edges on ρ. A path is *simple* if $v_i \neq v_j$ for $i \neq j$, *i.e.,* a vertex appears on ρ at most once. We consider simple paths in the sequel.

We refer to v_2 as a *child* of v_1 if (v_1, v_2) is an edge in E, and as a *descendant* if there exists a path from v_1 to v_2. A vertex is a called *leaf* if it has no children.

Parameters. To determine whether a vertex u_0 in G_1 matches a vertex v_0 in G_2, parametric simulation inductively considers the "closeness" of descendants of u_0 and descendants of v_0.

Given a descendant u' of u_0 (resp. v' of v_0) connected by path ρ_1 (resp. ρ_2), we define score functions h_v and h_ρ:

$$h_v(u', v') = \mathcal{M}_v(L_1(u'), L_2(v')) \quad (1)$$

$$h_\rho(\rho_1, \rho_2) = \frac{\mathcal{M}_\rho(L_1(\rho_1), L_2(\rho_2))}{\mathsf{len}(\rho_1) + \mathsf{len}(\rho_2)} \quad (2)$$

As will be seen in Section IV, $\mathcal{M}_v$ is a function that assesses how close u' and v' are to each other, based on their labels (types and values), and $\mathcal{M}_\rho$ inspects how close the association of u' to u_0 and that of v' to v_0 is, based on the labels on paths ρ_1 and ρ_2. Intuitively, the longer a path is, the weaker the association is; hence $\mathcal{M}_\rho(\rho_1, \rho_2)$ is divided by $\mathsf{len}(\rho_1) + \mathsf{len}(\rho_2)$. Both $h_v(u', v')$ and $h_\rho(\rho_1, \rho_2)$ are in [0, 1].

Note that types (*e.g.,* item and person) and values (*e.g.,* red and 500) are just labels of a node, and different nodes can have the same types or values, *i.e.,* type and value do not uniquely identify a vertex in a graph like the tuple id in relational database [72]. We do not use node id in our algorithms.

To identify u_0 and v_0 in practice, it often suffices to inspect a small number of their important properties (descendants; *e.g.,* 18 in Section VII). In light of this, we adopt an ML-based ranking function $h_r(\cdot, \cdot)$ and a bound k such that given a vertex u, $h_r(u, k)$ ranks the descendants of u and selects top-k ones along with a path for each, which represent characteristic features of u; similarly for $h_r(v, k)$ (see Section IV). Denote by V_v^k the set of top-k descendants of v picked by $h_r(v, k)$.

Using $h_r(\cdot, \cdot)$ is to strike a balance between the complexity and accuracy of entity linking. Indeed, there are exponentially many paths to descendants of u, and it is impractical to enumerate them, especially when G_1 or G_2 is dense.

Example 3: Consider vertices u_6 in canonical graph G_D of Fig. 3 and v_8 in graph G of Fig. 1. The closeness of vertices u_6 and v_8 is assessed by $h_v(u_6, v_8) = \mathcal{M}_v(L_D(u_6), L(v_8)) = \mathcal{M}_v(\text{Dame 7}, \text{Dame Gen 7})$. For paths $\rho_1 = (u_2, u_9)$ in G_D and $\rho_2 = (v_{10}, v_{15}, v_{19}, v_9)$ in G, their closeness is computed by $h_\rho(\rho_1, \rho_2) = \mathcal{M}_\rho(\text{made\_ in}, (\text{factorySite, isIn, isIn}))/(1+3)$.

Let k=5. Function h_r may select descendants **item** u_{10}, **material** u_3, **color** u_4, **type** u_6 and **brand** u_2 as properties of u_1 in G_D. Similarly, it selects soleMadeBy v_6, names v_0, brandName v_{10}, typeNo v_8 and hasColor v_{12} for v_1 in G. □

Symbol	Notation
$\mathcal{R}$, $\mathcal{D}$	database $\mathcal{D}$ of schema $\mathcal{R}$
G_D	RDB2RDF canonical graph of $\mathcal{D}$
$G = (V, E, L)$	labeled directed graph
h_v, h_ρ, h_r	score functions h_v, h_ρ and ranking function h_r
σ, δ, k	thresholds (vertex & path associations, # of properties)
V_u^k	the top-k descendants picked by h_r
$S_{(u,v)}$	lineage set of pair (u, v) of vertices
$\Pi(u, v)$	match of (u, v) via parametric simulation
$\Gamma(u_t, v_g)$	schema match pertaining to t and v_g

TABLE III
Notations

We use bounds σ for h_v and δ for h_ρ to assess the closeness of vertex labels and associations of labels on paths, respectively. We will show how to determine σ, δ, k in Section IV.

Parametric simulation. Taking functions (h_v, h_ρ, h_r) and thresholds (σ, δ, k) as parameters, *parametric simulation* is to check whether (u_0, v_0) is a match, for $u_0 \in V_1$ and $v_0 \in V_2$.

Given (u_0, v_0), parametric simulation computes a binary relation $\Pi(u_0, v_0) \subseteq V_1 \times V_2$ satisfying the following conditions:

(1) $(u_0, v_0) \in \Pi(u_0, v_0)$; and
(2) for each pair $(u, v) \in \Pi(u_0, v_0)$,
  (a) $h_v(u, v) \geq \sigma$; and
  (b) if u is not a leaf, then there exists a set $S_{(u,v)}$ of (u', v') that is a partial injective (1-to-1) mapping from V_u^k to V_v^k such that its aggregate score $\sum_{(u',v') \in S_{(u,v)}} h_\rho(\rho_{(u,u')}, \rho_{(v,v')}) \geq \delta$; and for each $(u', v') \in S_{(u,v)}$, $(u', v') \in \Pi(u_0, v_0)$.

Here $\rho_{(u,u')}$ is the path selected by $h_r(u, k)$ for u'; similarly for $\rho_{(v,v')}$. We call $S_{(u,v)}$ a *lineage set* of (u, v).

We say that (u_0, v_0) is a *match* by simulation parameterized with $(h_v, h_\rho, h_r, \sigma, \delta, k)$ if there exists such a nonempty $\Pi(u_0, v_0)$. There are possibly many such sets; to check whether (u_0, v_0) makes a match, we only need to check the existence of such a set, referred to as a *witness* of (u_0, v_0).

Intuitively, (u_0, v_0) is a match if (1) u_0 and v_0 are close enough, measured by function h_v based on their types and values; (2) there exists a lineage set $S_{(u_0,v_0)}$ of pairwise matching pairs (properties) such that their associations to (u_0, v_0) are close enough, measured by the aggregated score with function h_ρ; and (3) for a pair (u, v), $S_{(u,v)}$ is a set of pairs (u', v') such that each important property u' of u finds the "best" match v' if it exists (hence a partial 1-to-1 mapping) in terms of h_ρ scores on paths found by h_r. That is, (u_0, v_0) is a match if their "values" and important properties are close enough.

Example 4: Let σ=0.7, δ=1.5 and k=5. Vertices u_1 in Fig. 3 and v_1 of Fig. 1 match by parametric simulation, as follows.

(1) Vertices u_1 and v_1 make a match since they carry the same label, *i.e.,* $h_v(u_1, v_1) = \mathcal{M}_v(\text{item}, \text{item}) \geq \sigma$. Moreover, there exists a lineage set $S_{(u_1,v_1)} = \{(u_2, v_{10}), (u_3, v_6), (u_4, v_{12}),$

$(u_6, v_8), (u_{10}, v_0)\}$ that has an aggregate score above δ. Intuitively, $S_{(u_1,v_1)}$ confirms that u_1 and v_1 have the same material, color and brand, and similar names and types. We will see how to compute $\mathcal{M}_v()$ and aggregate scores $h_\rho()$ in Section IV, and how to pick lineage sets in Section V.

Note that it is not necessary for all properties of u_1 to find a match in $S_{(u_1,v_1)}$, *e.g.,* **qty** u_5 has no match in G; in other words, properties in $S_{(u_1,v_1)}$ suffices to match u_1 and v_1.

(2) To verify that $S_{(u_1,v_1)}$ is indeed a lineage set, inductive checking is needed: (a) (u_3, v_6) is valid since they bear the same label "Phylon foam", and u_3 is a leaf; similarly for $(u_4, v_{12}), (u_6, v_8)$ and (u_{10}, v_0); in contrast, (b) (u_2, v_{10}) has to be verified inductively itself since u_2 is not a leaf; a lineage set is $S_{(u_2,v_{10})} = \{(u_7, v_{20}), (u_8, v_{17}), (u_9, v_9), (u_{11}, v_{18})\}$.

(3) It confirms that pairs in $S_{(u_2,v_{10})}$ match and $S_{(u_2,v_{10})}$ has aggregate score above σ, since u_7 and v_{20} have similar labels and u_7 is a leaf; similarly for other pairs in $S_{(u_2,v_{10})}$. Intuitively, u_2 and v_{10} have the same name and manufacture, and carry similar country and made_in attributes.

(4) At this point, (u_1, v_1) is confirmed a match, which is witnessed by $\Pi(u_1, v_1) = \{(u_1, v_1)\} \cup S_{(u_1,v_1)} \cup S_{(u_2,v_{10})}$.

To check if (u_1, v_1) is valid, we (a) compare their labels via *ML model* h_v, (b) aggregate scores of their *outgoing paths*, and (c) *recursively check* if their descendants (u_2, v_{10}) match; this is needed for, *e.g.,* cycle detection in transaction graphs [93] for fraud detection [40], which cannot be detected by embeddings generated from random walks with bounded length [22]. These are key ingredients of parametric simulation. □

It is shown that for any $u_0 \in G_1$ and $v_0 \in G_2$, there exists a unique maximum $\Pi(u_0, v_0)$ by simulation with parameters $(h_v, h_\rho, h_s, \sigma, \delta, k)$ (see [16] for a proof). That is, parametric simulation retains the uniqueness of graph simulation [61].

The notations of the paper are summarized in Table III.

IV. Parameter Functions and Bounds

We next present module Learn of HER. We show how to learn parameters for parametric simulation, *i.e.,* score functions (h_v, h_ρ), ranking function h_r, thresholds (σ, δ) and bound k.

Given graphs $G_1{=}(V_1, E_1, L_1)$ and $G_2{=}(V_2, E_2, L_2)$, we define functions (h_v, h_ρ, h_r) pertaining to $u \in V_1$ and $v \in V_2$.

Vertex model $\mathcal{M}_v$. Function $h_v(u, v) = \mathcal{M}_v(L_1(u), L_2(v))$ takes two vertex labels as inputs, and returns their semantic similarity. We implement $\mathcal{M}_v(\cdot, \cdot)$ with a sentence embedding model [73], since it captures both sequential sentence information, such as descriptions of a movie, and words in vertex labels. The model takes string $L_1(u)$ (resp. $L_2(v)$) as input, and embeds it as a vector representation x_u (resp. x_v). The semantic similarity between $L_1(u)$ and $L_2(v)$ is assessed by:

$$\mathcal{M}_v(L_1(u), L_2(v)) = (|\cos(x_u, x_v)| + \cos(x_u, x_v))/2,$$

where $|\cdot|$ is the absolute value, such that $h_v(u, v){\in}[0, 1]$ and $\cos(x_u, x_v)$ computes the cosine similarity between x_u and x_v.

Edge model $\mathcal{M}_\rho$. Different from $\mathcal{M}_v$, $\mathcal{M}_\rho$ takes as input strings $L_1(\rho_1)$ and $L_2(\rho_2)$ of edge labels on paths, and quantifies their similarity. It sends $L_1(\rho_1)$ (resp. $L_2(\rho_2)$) to embedding model BERT [30] that captures sequential information of edge labels on paths, and gets its vector representation x_{ρ_1} (resp. x_{ρ_2}). A metric learning model compares x_{ρ_1} and x_{ρ_2}, and outputs their similarity score in [0, 1]. For example, $\mathcal{M}_\rho$ obtains embedding vectors x_{ρ_1} and x_{ρ_2} of "made_in" and "(factorySite, isIn, isIn)" by using BERT, respectively, and the learning model gives the similarity score of x_{ρ_1} and x_{ρ_2}.

We have to train the embedding and metric learning models in $\mathcal{M}_\rho$ instead of employing pre-trained NLP models, since edge labels are often special relation tokens for predicates, *e.g.,* "/akt:has-author" in a publication graph [28]. More specifically, (1) we construct a corpus C by randomly walking in G and collecting edge labels on the paths. (2) We then pre-train BERT model on C, driven by the unsupervised Masked Language Model task [30]. This enables BERT to capture sequential information in $L_1(\rho_1)$ (resp. $L_2(\rho_2)$) and embed it as vector x_{ρ_1} (resp. x_{ρ_2}). (3) We jointly train the metric learning and BERT models using annotated matching pairs (ρ_1, ρ_2). Thus, BERT fine-tunes the embeddings of x_{ρ_1} and x_{ρ_2}, and the learning model measures their semantic similarity.

An ideal $\mathcal{M}_v$ (resp. $\mathcal{M}_\rho$) scores similar pairs above 0.5 and dissimilar ones below 0.5 [91]; this guides model training and fine-tuning (see more details in Section VII).

Example 5: Consider the candidate match (u_7, v_{20}) in the lineage set $S_{(u_2,v_{10})}$ of Example 4. Its vertex similarity is $\mathcal{M}_v(\mathsf{Germany}, \mathsf{Germany}){=}1{\geq}\sigma$, and the closeness of the association of u_7 to u_2 (represented by the path $\rho_1{=}(u_2, u_7)$) and that of v_{20} to v_{10} (represented by the path $\rho_2{=}(v_{10}, v_{20})$) is measured by $h_\rho(L_D(\rho_1), L(\rho_2)){=}\mathcal{M}_\rho(L_D(\rho_1), L(\rho_2))$ $/(\mathsf{len}(\rho_1) + \mathsf{len}(\rho_2)) = \mathcal{M}_\rho(\mathsf{country}, \mathsf{brandCountry})/2 = 0.75/2 = 0.375$. Here $\mathcal{M}_\rho(\mathsf{country}, \mathsf{brandCountry}){=}0.75$ is computed using the embedded vectors of strings "country" and "brandCountry". After considering all pairs in $S_{(u_2,v_{10})}$, we compute the sum of their associations to (u_2, v_{10}), and find that the aggregate score is 1.6, which is greater than δ. □

Remark. We employ Bert with fine-tuning as an exemplary implementation. HER is open to other models in place of Bert. Sentence Bert is adopted since it effectively embeds long descriptions of entities, and performs well in fine-tuning embeddings [78]. With Bert, HER achieves good accuracy (see Section VII), since parametric simulation computes scores by multiple matching paths, and a single failure of embedding-based similarity measure has little impact on the result.

Ranking function h_r. Given vertex v and bound k, function h_r returns top-k descendants of v together with a path for each such descendant, representing important properties of v. It works in two steps: (1) it selects a set of m paths from v by using a language model $\mathcal{M}_r$, where m is the number of the children of v; and (2) it ranks the m paths by using a path resource allocation (PRA) algorithm, and returns top-k ones.

(1) For each outward edge e_i of v, function h_r selects a path ρ_i from v guided by language model $\mathcal{M}_r$, and adds ρ_i to a set P. For instance, starting at edge e_1 from v to v_1, h_r initiates

$\rho_1 = (v, v_1)$, presents e_1 to $\mathcal{M}_r$, and obtains a list E_{p_1} of all edges from v_1 with their possibility of following "word" e_1. Then from all outward edges of v_1, h_r chooses an edge e_2 from v_1 to v_2 with the highest possibility in E_{p_1}, appends v_2 to ρ_1 and feeds e_2 to $\mathcal{M}_r$ for predicted list E_{p_2}. The iteration proceeds until (a) $\mathcal{M}_r$ returns the "stop signal", *i.e.,* the end of sentence tag "<eos>"; (b) there is no outward edge to choose; or (c) the path forms a cycle (it is then abandoned).

Here we use LSTM network as $\mathcal{M}_r$ since it can model the semantics of labels on paths in knowledge graphs [54], [55], [56]. Given an edge label, LSTM can generate a path following the edge label with reasonable semantic meanings [58].

(2) Function h_r ranks paths in P as follows. Given a path ρ=$(v_0, v_1, \ldots, v_l)$, we extend resource allocation [56] and propose PRA to measure whether ρ is a meaningful connection by

$$R(\rho) = \prod_{i=0}^{l-1} \frac{1}{|ch(v_i)|},$$

where $ch(v_i)$ denotes the set of v_i's children. Intuitively, PRA assumes that a resource "flows" from the starting vertex of a path, and equally divides at each vertex in the middle. After propagation, PRA quantifies the semantic association of ρ in terms of the amount of resource that reaches v_l from v_0 via ρ.

Example 6: Taking v_1 in Fig. 1 and $k = 5$ as input, $\mathcal{M}_r$ selects paths starting from each outward edge of v_1. For example, given edge (v_1, v_{10}) with the edge label "brandName", $\mathcal{M}_r$ returns the end of sentence tag "<eos>", which terminates path selection; this is because the trained language model prefers to select paths with fewer branches and stronger semantic associations. Thus, it stops after v_{10}, since v_{10} has many descendants that will diverge and weaken the semantic association of longer paths. Finally, it outputs path (v_3, v_{13}).

After picking 8 paths via model $\mathcal{M}_r$ (*i.e.,* (v_1, v_0), (v_1, v_2), (v_1, v_6), (v_1, v_8), (v_1, v_{10}), (v_1, v_{11}), (v_1, v_{12}) and (v_1, v_{31})), h_r ranks them with PRA, drops (v_1, v_2), (v_1, v_{11}) and (v_1, v_{31}) for low scores, and gets 5 descendants v_0, v_6, v_8, v_{10} and v_{12}, with an associated path for each. □

Training. We prepare training data for $\mathcal{M}_r$ as follows. (1) For each vertex v, we first find the set V_r of all reachable vertices of v. Then we inspect the label of each vertex v' in V_r and remove those whose labels are machine codes, *e.g.,* URL or ID. This process is automatic as pre-trained embedding models (*e.g.,* GloVe [70]) recognize machine codes as unknown words. (2) For each vertex v' in V_r, we find all simple paths from v to v', quantify each by PRA and add the one with the maximum value to the training dataset. This preparation process does not take long, since we can practically collect enough paths by clustering and inspecting representative entities only.

Thresholds σ, δ and bound k. The objective of selecting σ, δ and k is to maximize F-measure (for accuracy) defined with precision and recall. Here precision, recall and F-measure are (1) the ratio of true matches to the matches returned, (2) the ratio of true matches to the annotated matches, and (3) $2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$, respectively.

We choose σ, δ and k by random search [19], rather than grid search that is computationally expensive for enumerating all combinations of the three parameters. More specifically, we construct a sampling validation set consisting of 15% of all annotated vertex pairs (u_t, v_g) randomly taken from G_D and G, which participate in neither model training nor testing. Then we evaluate the model on this validation set using random combinations of σ, δ and k. We pick the values that maximize the F-measure after limited number of trials.

Interaction and refinement. HER allows users to inspect and annotate matching decisions. It collects false positive (FP) and false negative (FN) pairs to refine $\mathcal{M}_v$ and $\mathcal{M}_\rho$. Given an FP (resp. FN) feedback, we mark its vertex matches and path-path matches as dissimilar (resp. similar) samples with similarity score 0 (resp. 1) to fine-tune $\mathcal{M}_v$ and $\mathcal{M}_\rho$.

To handle false feedback, we demonstrate the results to multiple users and conduct majority voting to reduce noise, which is a common practice for annotation quality control [46]. Moreover, we employ triplet loss function [75] to ensure robust model fine-tuning, which has proven effective in suppressing the negative influence of (possible) remaining false feedback.

Complexity. Once the training completes, it takes linear time for h_v and h_ρ to measure the similarity. It takes $O(|V||E|)$ time for function h_r to select top-k descendants and associated paths for each vertex v in a graph $G = (V, E, L)$.

V. Parametric Simulation Algorithm

We now show that parametric simulation is in quadratic-time. As a proof, we develop such an algorithm for module SPair of HER, denoted by ParaMatch. It takes functions (h_v, h_ρ, h_r) and bounds (σ, δ, k) as parameters. Given a tuple $t \in \mathcal{D}$ and a vertex v_g in G, it checks whether (u_t, v_g) is a match in $O(|G||G_D|)$ time, where G_D is the canonical graph of $\mathcal{D}$, and u_t is the vertex in G_D denoting t via mapping f_D.

Overview. ParaMatch is recursive. Given a pair (u, v) of vertices, it finds a lineage set $S_{(u,v)}$ of top-k descendants of u and v, and recursively checks pairs of the descendants. For $(u', v') \in S_{(u,v)}$ that makes a match, it sums up the associations between (u, v) and (u', v'), and checks whether the aggregate score reaches δ. It returns true if so. Otherwise it backtracks and examines other lineage sets. It returns false if no lineage set witnesses (u, v) as a match.

This is nontrivial. (1) When inspecting a pair (u_1, v_1), it has to select top-k descendants of u_1 and v_1; special care has to be taken to avoid picking the same vertex during repeated different recursive calls. (2) Candidate matches (u_1, v_1) and (u_2, v_2) may depend on each other, *e.g.,* when they are in a strongly connected component. This makes it tricky to backtrack and decide when to return false (see [16] for an example).

To cope with these we employ two hashmap structures:

(1) ecache, to record V_u^k, the top-k selected descendants for each vertex u, and avoid repeated descendant selection; and

(2) cache, to record the current states of candidate matches and dependencies among candidates. For each candidate (u, v),

cache$[u, v]$ is a pair $[\varphi, \mathcal{W}]$, which is either [false, $\emptyset$] or [true, $\mathcal{W}$], where $\mathcal{W}$ is a set of candidate matches, and φ is a Boolean value indicating whether (u, v) is invalid (false) or valid (true) under the condition that all candidates in $\mathcal{W}$ are valid.

Observe the following. (a) If (u, v) and (u', v') are interdependent, (u, v) and (u', v') are marked $[\text{true}, \mathcal{W}_1]$ and $[\text{true}, \mathcal{W}_2]$ in cache, respectively, and if $(u', v') \in \mathcal{W}_1$ and $(u, v) \in \mathcal{W}_2$, then both (u, v) and (u', v') are matches by the definition of parametric simulation. (b) We only need to store matches for vertices of V_u^k in cache$[u, v]$, *i.e.,* $|\mathcal{W}| \leq k$; moreover, the interdependence can be deduced from such $\mathcal{W}$.

In addition, we adopt the following strategies.

(3) For each top-k descendant u' of u, we sort the vertices v' in V_v^k in the descending order of the association between (u', v') and (u, v). When we search a candidate match v' for u', we follow the order in V_v^k. Intuitively, this helps us decide earlier whether we may not get a lineage set with aggregate score reaching δ and safely return false, since backtracking in the descending order always yields smaller scores.

(4) When candidate match (u, v) is invalided, we first identify candidates (u', v') that directly depend on (u, v), *i.e.,* $(u, v) \in$ cache$[u', v'].\mathcal{W}$. We then call ParaMatch to recheck whether (u', v') is still valid. Observe that this suffices to deal with interdependent candidates; indeed, if (u', v') is also invalid, the candidates that indirectly depend on (u', v') are rechecked when recursive ParaMatch backtracks.

Algorithm. Putting these together, we present ParaMatch in Fig. 4. It returns true for vertices $u_t \in G_D$ and $v_g \in G$ if u_t matches v_g and false otherwise. It works in three steps.

(1) Initial stage (lines 1-11). ParaMatch starts with two steps. (a) It first checks whether (u, v) can be a match by inspecting their labels (line 1-2), and whether u is a leaf (line 3-4). (b) It then constructs a set of candidate matches for each descendant of u (lines 6-11). If the top-k descendants of u or v are stored in ecache, it simply initializes V_u^k and V_v^k with ecache$[u]$ and ecache$[v]$, respectively. Otherwise it calls function h_r to pick top-k descendants of u and v (lines 6-10). After these, it builds a set $l_{u'}$ of candidate matches for each descendant u' of u (*i.e.,* $v' \in l_{u'}$ if $v' \in V_v^k$ and $h_v(u', v') \geq \sigma$), and sorts $l_{u'}$ in the descending order of associations (line 11).

(2) Matching stage (line 12-27). At this stage, ParaMatch inductively checks top-k descendants of u. At first, it adopts an early termination strategy and checks whether the maximum score among all possible lineages sets $S_{(u,v)}$ of (u, v) can reach δ; if not, (u, v) is confirmed invalid and false is returned (line 12-14); here $v'_{j,1}$ is the vertex having the maximum h_ρ score among all matches of u'_j. Otherwise for each selected descendant u', it finds a candidate for u', by checking V_v^k following the descending order of $l_{u'}$ (line 16). For a vertex v' in $l_{u'}$, it first checks whether (u', v') has been validated. If so, it directly uses the previous result. Otherwise, it checks (u', v') by recursively calling ParaMatch (lines 17-19). If (u', v') is valid, it accumulates its association to (u, v) in a variable sum, and adds (u', v') to the set $\mathcal{W}$ (line 21). Then it checks whether the value of sum reaches δ. If so, it marks (u, v) as $[\text{true}, \mathcal{W}]$ and returns true (lines 22-23). Otherwise, it checks whether we can find a match of u' in the remaining vertices of $l_{u'}$ such that the maximum score can reach δ (lines 25-27).

(3) Cleanup stage (lines 28-32). ParaMatch performs necessary cleanup to entries in cache after (u, v) is confirmed invalid. It first sets cache$[u, v]$ to [false, $\emptyset$] (line 28), and then reruns ParaMatch to update stale cache entries that directly depend on (u, v) (lines 29-31). Finally, it returns false (line 32).

Example 7: Recall Example 4. We show how ParaMatch finds that items u_1 and v_1 make a match as follows.

(1) In the first stage, the hashmap is set: cache$[u_1, v_1] = [\text{true}, \emptyset]$. The top-$k$ descendants of u_1 and v_1 are selected by h_r: $V_{u_1}^k = \{u_2, u_3, u_4, u_6, u_{10}\}$ and $V_{v_1}^k = \{v_0, v_6, v_8, v_{10}, v_{12}\}$. The sorted lists are $l_{u_2} = \{v_{10}\}$, $l_{u_3} = \{v_6\}$, $l_{u_4} = \{v_{12}\}$, $l_{u_6} = \{v_8, v_0\}$ and $l_{u_{10}} = \{v_0, v_8\}$ based on their label similarity.

(2) During the matching stage, matches are recursively identified for descendants of u_1 (*i.e.,* u_2, u_3, u_4, u_6 and u_{10}).

(a) Since u_3, u_4, u_6 and u_{10} are leaves, (u_3, v_6), (u_4, v_{12}), (u_6, v_8) and (u_{10}, v_0) are valid (lines 3-4) during the recursive calls. By now the aggregate score is below δ, *i.e.,* $h_\rho(\text{material}, \text{soleMadeBy}) + h_\rho(\text{color}, \text{hasColor}) + h_\rho(\text{type}, \text{typeNo}) + h_\rho(\text{item}, \text{names}) = 1.4 < \delta$. Thus, it checks (u_2, v_{10}).

(b) Vertex u_2 has four outgoing edges: (u_2, u_7), (u_2, u_8), (u_2, u_9) and (u_2, u_{11}). Hence ParaMatch is recursively called to match u_7, u_8, u_9 and u_{11} when processing (u_2, v_{10}). It finds that (u_7, v_{20}), (u_8, v_{17}), (u_9, v_9) and (u_{11}, v_{18}) are matches. Since their aggregate score $h_\rho(\text{country}, \text{brandCountry}) + h_\rho(\text{manufacturer}, \text{belongsTo}) + h_\rho(\text{made in}, \text{factorSiteisIn}) + h_\rho(\text{name}, \text{type}) = 1.6 \geq \delta$, (u_2, v_{10}) is confirmed to be a match, and thus the hashmap is updated: cache$[u_2, v_{10}] = [\text{true}, \{(u_7, v_{20}), (u_8, v_{17}), (u_9, v_9), (u_{11}, v_{18})\}]$.

(c) Now ParaMatch checks the aggregate score of descendant matches of (u_1, v_1). It finds that $h_\rho(\text{material}, \text{soleMadeBy}) + h_\rho(\text{color}, \text{hasColor}) + h_\rho(\text{type}, \text{typeNo}) + h_\rho(\text{item}, \text{IsA}) + h_\rho(\text{brand}, \text{brandName}) = 1.87 \geq \delta$. Thus, (u_1, v_1) is valid. Then ParaMatch updates cache$[u_1, v_1] = [\text{true}, \{(u_2, v_{10}), (u_3, v_6), (u_4, v_{12}), (u_6, v_8), (u_{10}, v_0)\}]$, and returns true. □

Analyses. Algorithm ParaMatch is correct and takes quadratic time in the worst case, the same as graph simulation [41]. Indeed, (I) the algorithm takes $O(|V_D||E_D|+|V||E|)$ time to select top-k descendants for each pair (u, v) (see Section IV); and (II) checking whether $(u_t, v_g) \in \Pi(u_t, v_g)$ takes $O(|V_D||V|)$ time in the worst case, since the total number of recursive calls is bounded. For (II), (a) there exist at most $O(|V_D||V|)$ candidate matches; (b) for each (u, v), ParaMatch is called at most $k^2 + 1$ times, by the use of hashmap cache and moreover, (i) the cleanup stage can only be called once for each candidate in cache$[u, v].\mathcal{W}$ and (ii) $|\text{cache}[u, v].\mathcal{W}| \leq k^2$; (c) line 1 of Fig. 4 takes $O(|V_D||V|)$ times in total; and (d) during each recursive call, all lines of Fig. 4 take $O(1)$ time except line 1 and recursive calls (line 16, lines 23-25). Thus ParaMatch takes at most $O((|V_D|+|E_D|)(|V|+|E|))$ time.

Input: $G_D = (V_D, E_D, L_D)$, $G = (V, E, L)$, $(u, v) \in V_D \times V$, and a set FP of functions h_v, h_ρ, h_r and parameters σ, δ, k.
Output: true if (u, v) is a match, and otherwise false.

1. **if** $h_v(u, v) < \sigma$ **then** /*initial stage*/
2. cache$[u, v]$:= [false, $\emptyset$]; **return** false;
3. **if** u is a leaf **then**
4. cache$[u, v]$:= [true, $\emptyset$]; **return** true;
5. cache$[u, v]$:= [true, $\emptyset$]; $\mathcal{W} := \emptyset$; sum := 0;
6. **if** $u \notin$ ecache **then**
7. extract the set V_u^k of selected vertices of u; ecache$[u]$:=V_u^k;
8. **if** $v \notin$ ecache **then**
9. extract the set V_v^k of selected vertices of v; ecache$[v]$:=V_v^k;
10. V_u^k := ecache$[u]$; V_v^k := ecache$[v]$;
11. construct a sorted list $l_{u'} = \{v' \mid v' \in V_v^k \wedge h_v(u', v') \geq \sigma\}$ for each $u' \in V_u^k$ following descending order of h_ρ score;
12. MaxSco:= $\sum_j h_\rho(\rho(u, u'_j), \rho(v, v'_{j,1}))$; /*matching stage*/
13. **if** MaxSco $< \delta$ **then**
14. cache$[u, v]$:= [false, $\emptyset$]; **return** false;
15. **for each** $u' \in V_u^k$ **do**
16. **for each** $v' \in l_{u'}$ in the order of $l_{u'}$ **do**
17. **if** $(u', v') \in$ cache **then**
18. match := cache$[u', v'].\varphi$;
19. **else** match := ParaMatch$(G_D, G, (u', v'),$ FP);
20. **if** match **then**
21. sum += $h_\rho(\rho_{(u,u')}, \rho_{(v,v')})$; $\mathcal{W} := \mathcal{W} \cup \{(u', v')\}$;
22. **if** sum $\geq \delta$ **then**
23. cache$[u, v]$:= [true, $\mathcal{W}$]; **return** true;
24. **break**;
25. MaxSco := MaxSco $- h_\rho(\rho(u, u'), \rho(v, v'))$ $+ h_\rho(\rho(u, u'), \rho(v, v'_n))$
26. **if** MaxSco $< \delta$ **then**
27. **break**;
28. cache$[u, v]$:= [false, $\emptyset$]; /*cleanup stage*/
29. **for each** (u^p, v^p) such that $(u, v) \in$ cache$[u^p, v^p].\mathcal{W}$ **do**
30. **unset** cache$[u^p, v^p]$;
31. ParaMatch$(G_D, G, (u^p, v^p),$ FP);
32. **return** false;

Fig. 4. Algorithm ParaMatch

In contrast, bounded simulation and strong simulation take $O(|V|(|V_D|+|E_D|)(|V|+|E|))$ time [33], [57].

Theorem 1: *Given graphs (G_D, G) and a pair (u_t, v_g) of vertices for $u_t \in G_D$ and $v_g \in G$,* ParaMatch *takes $O((|V_D|+|E_D|)(|V|+|E|))$ time to decide whether (u_t, v_g) is valid.* □

VI. Computing All Matches

We now develop algorithms to compute (1) all matches (u_t, v_g) for a given tuple t in database $\mathcal{D}$ (*i.e.,* module VPair), where u_t is the vertex denoting t in the canonical graph G_D of $\mathcal{D}$, and v_g is a vertex in graph G, and (2) all matches across G_D and G (*i.e.,* APair), based on parametric simulation.

We first develop algorithms for VPair and APair (Section VI-A). We then parallelize these algorithms (Section VI-B). We defer details and examples to [16] for the lack of space.

A. Algorithms for VPair and APair

VPair. We first present algorithm VParaMatch for VPair. It takes functions (h_v, h_ρ, h_r) and bounds (σ, δ, k) as parameters, and a tuple $t \in \mathcal{D}$ as input. It computes the set $\Pi(u_t)$ of (u_t, v_g) based on parametric simulation for v_g in G, defined as

$$\Pi(u_t) = \{(u_t, v_g) \mid v_g \in G,\ \Pi(u_t, v_g) \neq \emptyset\}.$$

As opposed to ParaMatch, vertex v_g is not given as input.

Input: $G_D = (V_D, E_D, L_D)$, $G = (V, E, L)$, a vertex u_t in G_D, and a set FP of functions h_v, h_ρ, h_r and parameters σ, δ, k.
Output: The set $\Pi(u_t)$ of matches.

1. $\Pi(u_t) := \emptyset$; $\mathcal{C}(u_t) := \emptyset$; initialize the hashmap cache;
2. **for each** $v_g \in V$ such that $h_v(u_t, v_g) \geq \sigma$ **do**
3. add $(u_t, v_g) \in \mathcal{C}(u_t)$;
4. sort matches (u, v) in $\mathcal{C}(u_t)$ in increasing orders of degrees;
5. **for each** $(u, v) \in \mathcal{C}(u_t)$ **do**
6. remove (u, v) from $\mathcal{C}(u_t)$;
7. **if** $(u, v) \in$ cache and cache$[u, v].\varphi =$ true **then**
8. add (u, v) to $\Pi(u_t)$;
9. **else** match := ParaMatch$(G_D, G, (u, v),$ FP);
10. **if** match = true **then**
11. add (u, v) to $\Pi(u_t)$;
12. **return** $\Pi(u_t)$;

Fig. 5. Algorithm VParaMatch

A brute-force approach to computing $\Pi(u_t)$ is to run ParaMatch for each (u_t, v_g) with $h_v(u_t, v_g) \geq \sigma$. It is, however, not very efficient. Hence we develop another algorithm.

Algorithm VParaMatch. As shown in Fig. 5, VParaMatch first selects all vertices v_g in G with $h_v(u_t, v_g) \geq \sigma$, and initializes a set $\mathcal{C}(u_t)$ with such candidates (u_t, v_g) (lines 2-3). It then sorts the pairs in $\mathcal{C}(u_t)$ following the increasing order of degrees of vertices in $\mathcal{C}(u_t)$ (line 4). Intuitively, starting from vertices with smaller degrees, VParaMatch can find more candidate matches to be valid or invalid earlier, and reduce runtime. After that VParaMatch iteratively checks each (u, v) following its order in $\mathcal{C}(u_t)$ (lines 6-11). More specifically, it first checks whether (u, v) had been confirmed valid (lines 7-8); if so, it adds it to $\Pi(u_t)$. Otherwise, it calls ParaMatch on (u, v) to verify its validity (lines 9-11). VParaMatch constructs inverted indices [98] on critical information as blocking strategies; for example, papers of the same year are in the same block.

APair. We next present AllParaMatch for APair. It computes the set Π of all matches across database $\mathcal{D}$ and graph G:

$$\Pi = \{(u_t, v_g) \mid u_t \in G_D,\ v_g \in G,\ \Pi(u_t, v_g) \neq \emptyset\},$$

where u_t (resp. v_g) is a vertex in G_D (resp. G). As opposed to ParaMatch and VParaMatch, none of u_t and v_g is input.

Algorithm AllParaMatch. Extending VParaMatch, the algorithm initializes a set $\mathcal{C}$ of candidate pairs (u_t, v_g) across G_D and G, for all $u_t \in V_D$ and $v_g \in V$ such that $h_v(u_t, v_g) \geq \sigma$. After this, it works just like algorithm VParaMatch.

Analyses. Algorithms VParaMatch and AllParaMatch do not increase the worst-case complexity bound of ParaMatch. Intuitively, to check whether $(u_t, v_g) \in \Pi(u_t, v_g)$, ParaMatch may already check all u (resp. v) reachable from u_t (resp. v_g), *i.e.,* in the worst case, it checks all pairs across G_D and G.

Corollary 2: VParaMatch *finds all matches pertaining to a vertex u_t, and* AllParaMatch *computes Π across database $\mathcal{D}$ and graph G, both in $O((|V_D|+|E_D|)(|V|+|E|))$ time.* □

B. Parallelization

When G_D and G are large, quadratic-time could still be expensive. To scale with large graphs, below we parallelize AllParaMatch, denoted by PAllMatch. Algorithms ParaMatch and VParaMatch can be parallelized along the same lines.

Setting. We adopt the following parallel setting.

(1) Algorithms run with n shared-nothing workers $P_1, \ldots, P_n$, under the Bulk Synchronous Parallel (BSP) model [84]. The computation is divided into multiple supersteps.

(2) Graph G_D is partitioned into n fragments $F_1^D, \ldots, F_n^D$ via edge-cut [21]. Each fragment F_i^D is defined as $(V_i^D \cup O_i^D, E_i^D, L_i^D)$, where (a) $(V_1, \ldots, V_n)$ is a partition of V^D, *i.e.,* $V_1^D \cup \ldots \cup V_n^D = V$ and $V_i^D \cap V_j^D = \emptyset$ for any $i \neq j$; (b) O_i^D is the set of *border nodes* that are not in V_i^D but have incoming edges from vertices in V_i^D; and (c) F_i^D is the subgraph of G_D induced by $V_i^D \cup O_i^D$. We will see that the vertices in O_i^D are used to synchronize computation between fragments.

(2) Graph G is also partitioned into n fragments $F_1^G, \ldots, F_n^G$ via edge-cut [21], where $F_i^G = (V_i^G \cup O_i^G, E_i^G, L_i^G)$. To reduce communication cost, special care is taken such that for each vertex u in fragment F_i^D, we assign all vertices v in G with their adjacent edges to fragment F_i^G if (u, v) is a candidate, *i.e.,* $h_v(u, v) \geq \sigma$. This is done by using inverted indices.

Below we denote by F_i both fragments F_i^D and F_i^G.

Fixpoint computation. Given fragmented graphs G_D and G, PAllMatch computes Π in parallel. It adopts the fixpoint model of GRAPE [35], [9]. It starts with a procedure PPSim at each worker, and then iteratively runs procedure IncPSim to incrementally refine the result, as follows (see [16] for details).

(1) PPSim. In the first superstep, each worker P_i starts by setting cache$[u, v']$ as $[\mathsf{true}, \emptyset]$ for each border node $v' \in O_i^G$ and each vertex $u \in F_i^D$, *i.e.,* it assumes that border node v' of F_i^G could match all vertices in F_i^D, due to the absence of the data of v' from local fragment F_i. Workers P_i then run AllParaMatch to compute partial result R_i^0 in F_i, *in parallel*.

(2) Messages. To synchronize the workers, the newly deduced invalid matches (u, v) (*i.e.,* cache$[u, v].\varphi$ is changed from true to false in the last superstep) are exchanged as messages. More specifically, for each $v \in V_i$, we define a status variable v.status, which stores invalid matches (u, v) deduced. Initially, v.status is $\emptyset$. Recall that border nodes $v \in O_i$ are associated with edges across different fragments. At the end of each superstep, the changes to v.status of border nodes in O_i are sent to other workers as messages, following the cross edges.

(3) IncPSim. Upon receiving message M_i, each worker P_i incrementally refines partial result R_i^j of superstep j at P_i by treating M_i as updates. More specifically, (a) it first initializes a set $\mathcal{U}$ of invalid matches $(u, v) \in M_i$ for border nodes $v \in O_i$; that is, PAllMatch improves R_i^j by using the results of other workers. (b) It then follows the *cleanup stage* of ParaMatch; for each $(u, v) \in \mathcal{U}$, it updates cache$[u, v]$ to be $[\mathsf{false}, \emptyset]$; it then calls ParaMatch for all entries in cache whose $\mathcal{W}$ overlaps with $\mathcal{U}$ to re-check the affected candidates.

At the end of each superstep, each worker generates messages and communicates with other workers as in (2) above.

(4) Termination. The process proceeds until it reaches a fixpoint *i.e.,* when $R_i^{r^*} = R_i^{r^*+1}$ for all $i \in [1, n]$ at some r^*. The match Π is the union of all partial results, *i.e.,* $\Pi = \cup_{i \in [1,n]} R_i^{r^*}$.

Correctness. One can verify the following (see [16]).

Theorem 3: *Given fragmented graphs G_D and G as above,* PAllMatch *correctly computes the match Π of (G_D, G).* □

Remarks. (1) PAllMatch can work asynchronously. Under the conditions of [34] we can show that PAllMatch correctly computes Π under the adaptive asynchronous parallel model [34].

(2) *IncPSim* can be extended to incrementally link entities in response to updates to $\mathcal{D}$ and G, in parallel.

(3) We adopt the fixpoint model of GRAPE to parallelize algorithms in Section V, without developing parallel algorithms starting from scratch. GraphScope [9], an open source version of GRAPE, is efficient and popular at GitHub; hence building HER atop GRAPE is within the reach of a large group of users.

(4) While blocking speeds up ER, it may miss matches when matching entities are in different blocks. It alone does not fit parametric simulation, which needs to recursively check descendants that may be in different blocks. Hence we use data-partitioned parallelism in place of blocking. For large datasets, we construct inverted indices [98] on critical information (*e.g.,* years of papers in DBLP for "blocking"; see also Section VII).

VII. Experimental Study

Using real-life and synthetic datasets, we experimentally evaluated HER for its (1) accuracy, (2) efficiency, (3) scalability and (4) the impact of user interactions on the accuracy.

Experimental Setting. We start with the setting.

Datasets. We used five real-life datasets shown in Table IV: (1) UKGOV, a collection of Camden Council data (Commercial Contracts, Parking Charges, Schools, Air Quality and Trees), exported in CSV and RDF formats from the Web sites [83]. (2) DBpediaP, subsets of DBPedia knowledge base about athletes and politicians in relations [4] and graphs [5]. (3) DBLP, publication data in relations [29] and graphs [28]. (4) IMDB, movie data in relations [13] and graphs [12]. (5) FBWIKI, consisting of (a) part of FreeBase knowledge graph [18] and (b) entries of people extracted from the wikidata [15].

We also tested on the Cell-Entity Annotation (CEA) task of SemTab 2020 challenge [45] that matches cells of tables to entities in Wikidata knowledge graph. The "Tough Tables" (2T) dataset [27], [26] from SemTab 2020 was used for this test due to its high difficulty. External Wikidata query APIs are allowed following the instructions of the SemTab challenge.

Based on the TPC-H data generator, we designed a graph generator to produce synthetic graphs G, controlled by the number of vertices (up to 36M) and edges (up to 305M), with vertex labels drawn from a set of 1.1M words and edge labels from a set of 100 words. We generated databases $\mathcal{D}$ with 70 columns (*i.e.,* edge labels in G_D).

ML models. For $\mathcal{M}_v$, we employed Sentence-bert [73], a pre-trained sentence embedding model for its high accuracy. For $\mathcal{M}_\rho$, we first constructed an edge label corpus (see Section IV) for pre-training BERT [30], which contains 195K labels in 1845 categories from all datasets. The pre-training

Dataset	$\|V_D\|$	$\|E_D\|$	$\|V\|$	$\|E\|$	Type
UKGOV	12.3M	11.9M	25.8K	76.9K	public services
DBpediaP	2M	5.4M	4.8M	11.7M	celebrity base
DBLP	36M	58.6M	15.9M	31M	citation network
IMDB	7.5M	34.2M	2.3M	5.4M	movies
FBWIKI	4.0M	7.4M	60.8M	362.2M	knowledge base
2T	937K	742K	1.11M	3.68M	semantic web

TABLE IV
DATASETS FOR EVALUATION

of the BERT-base takes 122.2min with 30 epochs. After that, we utilized 50% of all match/mismatch pairs to train other modules in $\mathcal{M}_\rho$, which takes 325.1s. The similarity model in $\mathcal{M}_\rho$ is implemented as a 3-layer neural network with width 1536, 256 and 1 in each layer. We adopted LSTM for ranking model $\mathcal{M}_r$ with the default configuration in [11], except using 650 and 650 hidden units per layer. When collecting training paths for $\mathcal{M}_r$, we took paths of at most 4 edges following [56] since longer paths usually yield weaker associations. For all datasets, we collected 118K paths as the training data for $\mathcal{M}_r$.

Evaluation. For accuracy tests (the F-measure, see Section IV), we manually annotated 5000 matches (*i.e.,* confirmed matches, which are tuple-vertex pairs) as ground truth for all datasets, assisted by a mapping provided by Google [7]. In addition, we randomly sampled tuples and vertices and obtained 5000 mismatches, which were also manually verified. Thus, the match/non-match ratio is 1. We used 50% of annotated pairs in each dataset to train $\mathcal{M}_\rho$, 15% as validation sets to select bounds σ, δ and k, and the rest for accuracy tests. For efficiency and scalability evaluations, the default σ, δ and k are set as 0.8, 2.1 and 20, respectively, unless stated otherwise.

Baselines. We used nine baselines. (1) MAGNN [37], a GNN-based model that learns vertex embeddings for similarity, with vertex attributes and meta-paths. We applied default configuration of [37], with random parameter search using the validation set. (2) Bsim, bounded simulation [33], based on vertex labels and topological matching. (3) JedAI [68], a rule-based ER toolkit implemented in Java. We configured JedAI with the "budget- and schema-agnostic workflow", including rules of "character 4-grams with TF-IDF weights and cosine similarity" [68]. This configuration "requires no parameter fine-tuning" and has been verified highly effective for different datasets [68]. (4) MAG [48] (Magellan), a state-of-the-art ML-based system for ER on relations. We adopted its random forest model with feature tables [10], and random parameter search on validation set. MAG is mainly implemented in Python, with functions optimized via C++. (5) DEEP [62], a deep-learning-based ER Python package under Magellan, configured with the best hybrid model snapshot and classifier based on the validation set [6]. We compared with top-4 challengers on 2T dataset of SemTab 2020 [45]: MTab [65], bbw [76], LP (LinkingPark) [23] and LexMa [82]. Since only LexMa is open-source [14], we also tested and compared with it on the five real-life tuple-vertex matching datasets.

The baselines represent (a) ML systems (MAGNN, MAG and DEEP), (b) non-ML rule-based method (JedAI), (c) topological matching (Bsim), and (d) Web-based cell matching (MTab, bbw, LP and LexMa). We did supervised training

F-measure	HER	MAGNN	Bsim	JedAI	MAG	DEEP	LexMa
UKGOV	0.94	0.78	OM	0.76	0.84	0.87	0.03
DBpediaP	0.96	0.73	OM	0.64	0.95	0.91	0.01
DBLP	0.94	0.65	OM	0.53	0.57	0.66	0.34
IMDB	0.93	0.71	OM	0.62	0.65	0.72	0.01
FBWIKI	0.96	0.74	OM	0.79	0.86	0.89	0.13

F-measure	HER	MTab	bbw	LP	LexMa
2T	0.615	0.907	0.863	0.81	0.587

TABLE V
ACCURACY (F-measure) ON TUPLE MATCHING AND CELL MATCHING

for ML models of MAGNN, MAG and DEEP with the same training data as for HER. MAGNN and Bsim used RDB2RDF to convert relations to graphs. In order for MAG and DEEP to take vertex v from graph G as input, we took v along with its 2-hop neighbors and flattened them into a tuple t_v, *i.e.,* we packed v into t_v with important features in its close neighbors as commonly practiced by ER methods. Then we flattened G into a relation $\mathcal{D}_G$. Given a tuple t in $\mathcal{D}$ and a vertex v in G, we compared t with t_v for SPair. Similarly, we conducted VPair and APair to find matches of an input tuple t and all matches, respectively. Bsim takes G_D as a graph pattern and computes its "match" in G for APair; however, it does not support SPair and VPair since it is based on pattern matching. External Wikidata query API was used to construct graph G for 2T dataset.

For a fair comparison, we tested the baselines and HER with a single machine using an Intel Xeon 2.5 GHz CPU and 192 GB memory, since only Bsim has a parallel solution among the five baselines. We also tested the parallel scalability of HER and Bsim on GRAPE [35], [9], using an HPC cluster of up to 16 machines connected by 10 Gbps links; each machine has an Intel Xeon 2.5 GHz CPU and 192 GB memory. Each experiment was run 5 times and the average is reported here.

Experimental Results. We next report our findings.

Exp-1: Accuracy. We first tested the F-measure on all datasets in Table IV. As shown in Table V, it is 0.94 for HER on average (except for 2T dataset), consistently outperforming all the baselines. Bsim ran out of memory (OM) for all datasets.

(1) HER is on average 31%, 22% and 17% more accurate than MAGNN, MAG and DEEP, respectively; this shows that parametric simulation is more accurate than using ML methods alone, by embedding ML models in topological matching and checking "global" properties. In particular, it quantifies entity similarities better than meta-path-based measures (MAGNN).

(2) HER beats JedAI by 42% on average, justifying that linking entities across relations and graphs needs both inductive topological matching and ML models, beyond existing rules.

(3) LexMa fails to match tuples to entities, as it just considers each cell value without semantic relations among them. Hence, cells in the same tuple may be mapped to disconnected and different entities. For example, a cell "London" in a tuple may be mapped to different "London" (*e.g.,* cities in UK, US and Canada) in the graph. Given such "independent" cell matches of one tuple, one can hardly decide to which entity the tuple should be mapped, resulting in rather low precision.

As for the CEA task (cell matching), the accuracy of HER

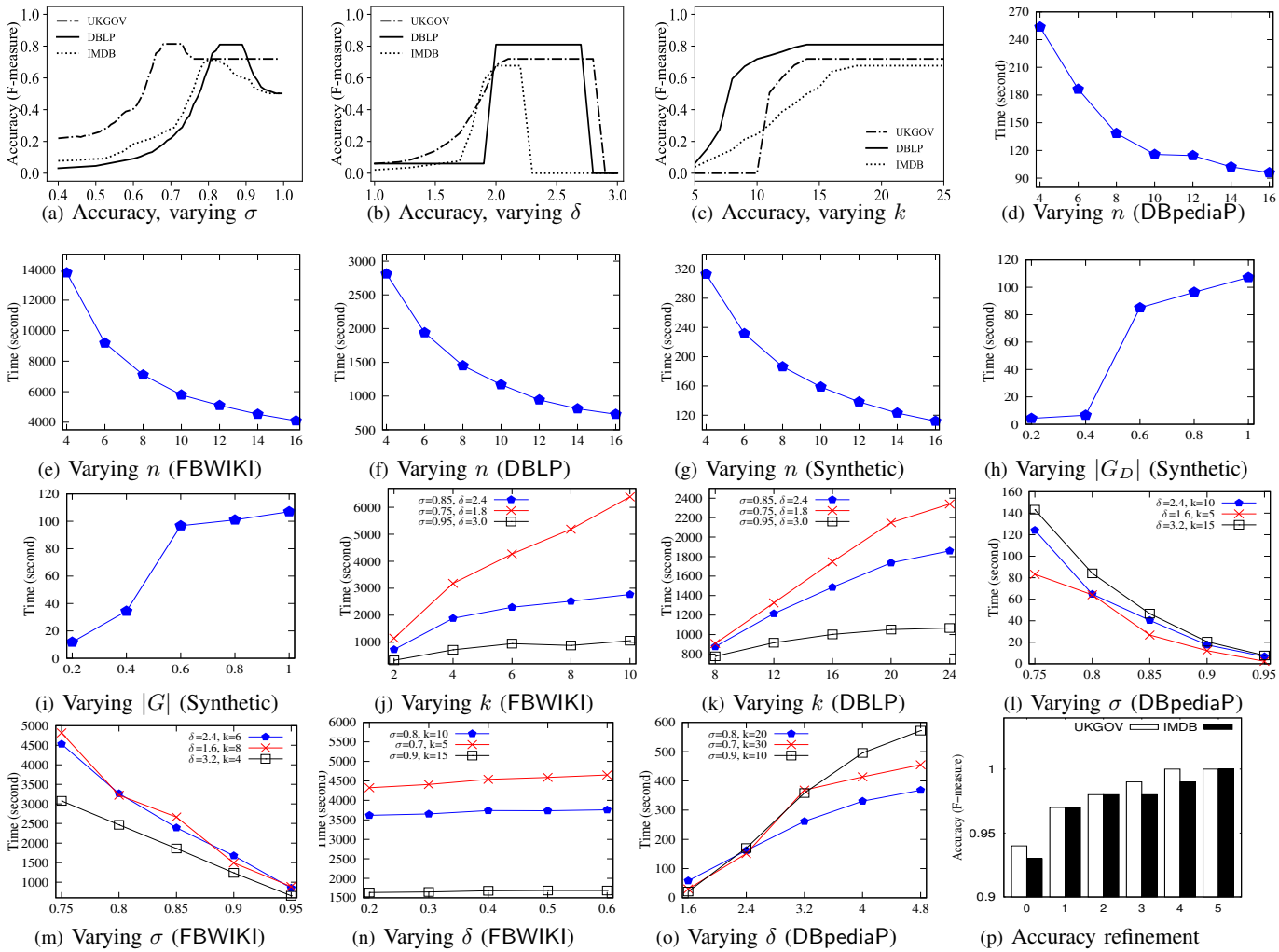

Fig. 6. HER accuracy and efficiency with varying parameters, HER scalability

on 2T is 0.615 (Table V), which is only higher than LexMa (ranked the 4th in SemTab 2020 [2]). This is because the main challenge of 2T is to correct its heavy misspelling and typos before actual matching, and the top-3 challengers are all assisted with specifically designed spell checkers that excel at correcting such string noises. HER is developed for matching tuples and entities, not for spell checking and cell matching.

Varying σ and δ. Fixing $\delta = 2.4$, $k = 20$, we varied σ from 0.4 to 0.99, to study the impact of σ on F-measure with 3 datasets for which optimal parameters are close. As shown in Fig. 6(a), F-measure first grows steadily when σ increases; it reaches the peak and then drops sharply with larger σ. This is because F-measure is the harmonic mean of precision and recall and the threshold is a trade-off between them.

Fixing $\sigma = 0.85$, $k = 20$, we varied δ from 1 to 3. As shown in Fig. 6(b), the impact of δ is similar for the same reason.

Varying k. Fixing δ=2.4 and σ=0.85, we varied bound k on descendants from 5 to 25, to test the impact of k on F-measure with the same datasets as above. As shown in Fig. 6(c), F-measure first increases and then remains stable after k reaches a value around 18. Both precision and recall increase at the beginning since more properties (path-path pairs) are inspected. However, when the pairs already accumulate sufficient scores, increasing k no longer improves F-measure.

Exp-2: Efficiency. Over real-life datasets DBpediaP and DBLP, we report in Table VI the efficiency of modes SPair and VPair of HER versus their competitors, using a single machine for fair comparison, since most of the baselines are not parallel. The results on the other datasets are consistent (not shown). As remarked earlier, Bsim supports neither modes. Since SemTab challenges rely on external APIs whose efficiency heavily depends on the Internet speed, SemTab baselines and 2T are omitted for a fair efficiency comparison.

SPair. Given a pair (t, v_g), HER checks whether (t, v_g) makes a match in 0.03ms and 0.12ms on DBpediaP and DBLP, respectively. On average it outperforms MAGNN, JedAI, MAG and DEEP by 20.6, 288, 2262 and 5629 times, respectively. That is, HER works well in the SPair mode.

VPair. Given a tuple t, VPair finds all its matching vertices in 1.43s and 15.9s over DBpediaP and DBLP, respectively. For a fair comparison, since HER firstly applies inverted index to search for candidate matching pairs before parametric simulation, we also supported the blocking step in JedAI, MAG and DEEP, using person names in DBpediaP and author names in DBLP to generate candidate tuple pairs before matching. On average HER outperforms the four baselines

by 199.7, 6.0, 44.7 and 110.7 times, respectively. Again, this shows that the response time of VPair is reasonably short.

APair. We find the following. (a) On DBpediaP, it takes 93.4s to convert data between relations and graphs, and 405.3s to finish matching in the APair mode, while the other baselines could not terminate within hours. (b) APair takes much longer than SPair and VPair, although the three have the same worst-case complexity. This is because APair has to check all candidates across G_D and D. While we can run APair offline on a single machine, we will see in Exp-3 that APair runs much faster in parallel when given multiple processors. (c) Bounded simulation Bsim takes much longer than HER and exceeds memory limit even on small graphs, since it takes the entire G_D as a graph pattern and computes the maximum match. In contrast, HER only checks vertices reachable from (u_t, v_g).

Exp-3: Scalability. We next evaluated the (parallel) scalability using large real-life datasets and synthetic data.

Varying n. Taking the entire $\mathcal{D}$ and G as input, we varied the number n of workers from 4 to 16 to test the parallel scalability of HER. As shown in Figures 6(d)–6(g), on average APair is 2.6, 3.4, 3.8 and 2.8 times faster on DBpediaP, FBWIKI, DBLP and synthetic data ($|G_D| = 342M$ and $|G| = 202M$), respectively. The results are similar for SPair and VPair.

Synthetic data. Fixing $n = 16$, we tested APair mode using large synthetic graphs, where $|V| = 30$M and $|E| = 172$M for G, and $|V_D| = 36.5$M and $|E_D| = 305.5$M for G_D. We also tested SPair and VPair modes with $n = 1$ using the entire synthetic data, which takes 0.68ms and 15.3s, respectively.

(1) Varying $|G_D|$. Taking the entire G as input and varying the size of G_D, we tested the performance of HER. Figure 6(h) shows that the execution time increases with larger $|G_D|$. HER takes 107s when $|G|$=202M and $|G_D|$=342M.

(2) Varying $|G|$. Figure 6(i) reports results over the entire G_D by varying $|G|$. The results are consistent with Figure 6(h).

Varying k. Fixing $n = 16$ and varying k from 2 to 10 and 8 to 24 with different σ and δ, we studied the impact of k over FBWIKI and DBLP, respectively. The values of k on FBWIKI are smaller because each vertex has less descendants on average. As shown in Figures 6(j) and 6(k) for APair, HER takes longer as k increases, as expected. This is because with larger k, more path-path pairs need to be inspected. Results are consistent for SPair and VPair, and on other graphs.

Varying σ and δ. We tested the impact of thresholds σ and δ with 16 machines. Varying σ from 0.75 to 0.95 with different configurations of δ and k, as shown in Figures 6(l) and 6(m) over DBpediaP and FBWIKI, respectively, APair takes less time as σ increases. This is because more invalid match candidates are removed in the early stage given higher σ; this reduces candidate checking and accelerates the matching. However, it takes longer as δ increases from 0.2 to 0.6 on FBWIKI and from 1.6 to 4.8 on DBpediaP, with various configurations of σ and k (see Figures 6(n) and 6(o)). The reason for this is that in order to reach higher matching threshold δ, the algorithm needs to check more path-path pairs. Note that the impact of varying δ for FBWIKI is smaller than that for other datasets as its matching paths are much longer.

Results are consistent for SPair and VPair on other graphs.

	DBpediaP		DBLP	
	SPair	VPair	SPair	VPair
HER	2.8×10^{-5}	1.4	1.2×10^{-4}	15.9
MAGNN	9.6×10^{-4}	357.1	8.3×10^{-4}	2374.3
Bsim	NA	NA	NA	NA
JedAI	1.3×10^{-2}	11.5	1.1×10^{-2}	62.0
MAG	1.0×10^{-1}	84.6	9.8×10^{-2}	480.5
DEEP	2.6×10^{-1}	209.8	2.5×10^{-1}	1188.2

TABLE VI
SEQUENTIAL EXECUTION TIME (S)

Exp-4: Refinement. We next tested the impact of user interaction on the accuracy, using UKGOV and IMDB. In each round, 50 pairs were given to five users. The users inspected them and annotated each pair either match or mismatch as feedback. Then we applied majority voting to the feedback to reduce the number of false annotations. These annotated pairs are collected to fine-tune the ML models as outlined in Section IV. As shown in Figure 6(p), F-measure goes up by 3% and 4% on UKGOV and IMDB, respectively, in the first round, and 5 rounds suffice for HER to reach 100% accuracy. Note that the 100% accuracy is achieved since human feedback is used to fine-tune the models and verify the matches. The results on other datasets are consistent (not shown).

Summary. We find the following. (1) Except on 2T dataset, HER is more accurate than ML-based and rule-based methods. On average HER beats MAGNN, JedAI, MAG and DEEP by 31%, 42%, 22% and 17% in accuracy, respectively. (2) HER performs the best in efficiency. When running in the VPair mode with a single machine, it is 90 times faster than the baselines on average. When $|G|$=202M and $|G_D|$=342M, SPair and VPair take 0.68ms and 15.3s with a single machine, respectively, and APair takes 107s using 16 machines, while all the baselines could not finish in hours. (3) HER scales well with the processor number n. It is on average 3.2 times faster when n varies from 4 to 16. (4) At most 5 rounds of user interaction suffice to fine-tune the ML models in HER.

VIII. CONCLUSION

We have developed system HER to semantically link entities across relational databases and graphs. We have proposed parametric simulation that embeds ML in global topological matching, and shown that it is in quadratic-time, the same as relational entity resolution. We have also developed ML models for learning parameters and parallel algorithms underlying HER. Our experimental study has shown that HER is promising in terms of its accuracy, scalability and efficiency.

One topic for future work is to extend HER to other data formats such as JSON, CSV and arrays. Another topic is to employ HER to extract, integrate and query data of different sources, which are open challenges to data lakes [63]. A third topic is to query relations and graphs in SQL by semantically extending the join operator of SQL via HER.

REFERENCES

[1] Semtab challenge. https://www.cs.ox.ac.uk/isg/challenges/sem-tab/.
[2] Semtab challenge 2020. https://www.cs.ox.ac.uk/isg/challenges/sem-tab/2020/index.html.
[3] Semtab challenge 2021. https://www.cs.ox.ac.uk/isg/challenges/sem-tab/2021/index.html.
[4] DBpedia as tables, 2020. https://wiki.dbpedia.org.
[5] DBpedia version 2016-10, 2020. https://wiki.dbpedia.org.
[6] Deepmatcher, 2020. https://github.com/anhaidgroup/deepmatcher.
[7] Freebase/wikidata mappings, 2020. *https://developers.google.com/freebase*.
[8] GRAPE. https://github.com/alibaba/libgrape-lite.git, 2020.
[9] Graphscope. https://graphscope.io/, 2020.
[10] Basic EM workflow 1 (Restaurants data set), 2020.
[11] Word-level language modeling RNN, PyTorch example, 2020.
[12] IMDB graph dataset, 2020. https://www.cs.toronto.edu.
[13] IMDB relational dataset, 2020. https://datasets.imdbws.com/.
[14] Open source implementation of LexMa, 2020. https://github.com/shaliniktyagi/TabularData_to_Knowledge_graph.
[15] Wikidata, 2020. *https://www.wikidata.org/*.
[16] Full version, 2021. https://homepages.inf.ed.ac.uk/s1837143/HER_full.pdf.
[17] Pre-trained GloVe word embedding of different dimensions., 2021. https://nlp.stanford.edu/data/glove.6B.zip.
[18] H. Bast, F. Bäurle, B. Buchhold, and E. Haußmann. Easy access to the freebase dataset. In *WWW*, page 95–98, 2014.
[19] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(1):281–305, 2012.
[20] T. Berners-Lee. Relational databases and the semantic Web (in design issues). *World Wide Web Consortium*, 1998.
[21] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.
[22] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *SIGMOD*, pages 1335–1349, 2020.
[23] S. Chen, A. Karaoglu, C. Negreanu, T. Ma, J.-G. Yao, J. Williams, A. Gordon, and C.-Y. Lin. Linkingpark: An integrated approach for semantic table interpretation. In *SemTab@ ISWC*, pages 65–74, 2020.
[24] Z. Chen, Q. Chen, F. Fan, Y. Wang, Z. Wang, Y. Nafa, Z. Li, H. Liu, and W. Pan. Enabling quality control for entity resolution: A human and machine cooperation framework. In *ICDE*, pages 1156–1167. IEEE, 2018.
[25] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. End-to-end entity resolution for big data: A survey. *ACM Comput. Surv.*, 53(6), 2019.
[26] V. Cutrona, F. Bianchi, E. Jiménez-Ruiz, and M. Palmonari. Tough Tables: Carefully Benchmarking Semantic Table Annotators [Data set]. Zenodo. https://zenodo.org/record/4246370#.YWUB9dpBz-g.
[27] V. Cutrona, F. Bianchi, E. Jiménez-Ruiz, and M. Palmonari. Tough tables: Carefully evaluating entity linking for tabular data. In *International Semantic Web Conference*, pages 328–343. Springer, 2020.
[28] DBLP. DBLP RDF data, 2020. http://dblp.rkbexplorer.com.
[29] DBLP. DBLP relational data, 2020. https://dblp.org.
[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.
[31] Y. Dong, N. V. Chawla, and A. Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*, 2017.
[32] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *PVLDB*, 8(12):1590–1601, 2015.
[33] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1), 2010.
[34] W. Fan, P. Lu, W. Yu, J. Xu, Q. Yin, X. Luo, J. Zhou, and R. Jin. Adaptive asynchronous parallelization of graph algorithms. *TODS*, 2020.
[35] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu. Parallelizing sequential graph computations. *TODS*, 43(4), 2018.
[36] J. Feng, M. Huang, Y. Yang, and X. Zhu. GAKE: Graph aware knowledge embedding. In *COLING*, pages 641–651, 2016.
[37] X. Fu, J. Zhang, Z. Meng, and I. King. MAGNN: metapath aggregated graph neural network for heterogeneous graph embedding. In *WC*, 2020.
[38] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
[39] B. Golshan, A. Y. Halevy, G. A. Mihaila, and W. Tan. Data integration: After the teenage years. In *PODS*, 2017.
[40] T. H. E. Helmy, M. zaki Abd-ElMegied, T. S. Sobh, and K. M. S. Badran. Design of a monitor for detecting money laundering and terrorist financing. *International Journal of Computer Networks and Applications*, 1(1):15–25, 2014.
[41] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[42] B. Hou, Q. Chen, Y. Wang, Y. Nafa, and Z. Li. Gradual machine learning for entity resolution. *TKDE*, 2020.
[43] R. Isele, A. Jentzsch, and C. Bizer. Silk server - adding missing links while consuming linked data. In *COLD*, volume 665, 2010.
[44] G. Jeh and J. Widom. Simrank: A measure of structural-context similarity. In *KDD*, pages 538–543, 2002.
[45] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, K. Srinivas, and V. Cutrona. Results of semtab 2020. In *CEUR Workshop Proceedings*, volume 2775, pages 1–8, 2020.
[46] D. R. Karger, S. Oh, and D. Shah. Iterative learning for reliable crowdsourcing systems. In *NIPS*, pages 1953–1961, 2011.
[47] J. Kasai, K. Qian, S. Gurajada, Y. Li, and L. Popa. Low-resource deep entity resolution with transfer and active learning. *arXiv preprint arXiv:1906.08042*, 2019.
[48] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12), 2016.
[49] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.
[50] C. Koutras, M. Fragkoulis, A. Katsifodimos, and C. Lofi. REMA: graph embeddings-based relational schema matching. In *EDBT/ICDT*, volume 2578, 2020.
[51] M. Kusumoto, T. Maehara, and K.-i. Kawarabayashi. Scalable similarity search for simrank. In *SIGMOD*, pages 325–336, 2014.
[52] S. Kwashie, L. Liu, J. Liu, M. Stumptner, J. Li, and L. Yang. Certus: An effective entity resolution approach with graph differential dependencies (GDDs). *PVLDB*, 12(6):653–666, 2019.
[53] B. Li, W. Wang, Y. Sun, L. Zhang, M. A. Ali, and Y. Wang. Grapher: Token-centric entity resolution with graph convolutional neural networks. In *AAAI*, 2020.
[54] M. Li, Q. Zeng, Y. Lin, K. Cho, H. Ji, J. May, N. Chambers, and C. Voss. Connecting the dots: Event graph schema induction with path language modeling. In *EMNLP*, pages 684–695, 2020.
[55] X. V. Lin, R. Socher, and C. Xiong. Multi-hop knowledge graph reasoning with reward shaping. *arXiv preprint arXiv:1808.10568*, 2018.
[56] Y. Lin, Z. Liu, H. Luan, M. Sun, S. Rao, and S. Liu. Modeling relation paths for representation learning of knowledge bases. In *EMNLP*, 2015.
[57] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *TODS*, 39(1):4:1–4:46, 2014.
[58] G. Melis, C. Dyer, and P. Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.
[59] F. Michel, J. Montagnat, and C. F. Zucker. A survey of RDB to RDF translation approaches and tools, 2014.
[60] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
[61] R. Milner. *Communication and concurrency*. Prentice hall, 1989.
[62] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, pages 19–34, 2018.
[63] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data lake management: Challenges and opportunities. *PVLDB*, 12(12), 2019.
[64] A. N. Ngomo and S. Auer. LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, 2011.
[65] P. Nguyen, I. Yamada, N. Kertkeidkachorn, R. Ichise, and H. Takeda. Mtab4wikidata at semtab 2020: Tabular data annotation with Wikidata. In *SemTab@ ISWC*, pages 86–95, 2020.
[66] P. Nguyen, I. Yamada, N. Kertkeidkachorn, R. Ichise, and H. Takeda. Mtab4wikidata at semtab 2020: Tabular data annotation with wikidata. In *SemTabISWC*, volume 2775, pages 86–95, 2020.
[67] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *TKDE*, 25(12):2665–2682, 2012.
[68] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. Three-dimensional entity resolution with JedAI. *Inf. Syst.*, 2020.
[69] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. The return of JedAI: End-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

[70] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.

[71] K. Qian, L. Popa, and P. Sen. Active learning for large-scale entity resolution. In *CIKM*, pages 1379–1388, 2017.

[72] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD*, pages 681–694, 2009.

[73] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP-IJCNLP*, pages 3980–3990, 2019.

[74] A. Saeedi, E. Peukert, and E. Rahm. Using link features for entity clustering in knowledge graphs. In *ESWC*, pages 576–592, 2018.

[75] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.

[76] R. Shigapov, P. Zumstein, J. Kamlah, L. Oberländer, J. Mechnich, and I. Schumm. bbw: Matching csv to wikidata via meta-lookup. In *CEUR Workshop Proceedings*, volume 2775, pages 17–26. RWTH, 2020.

[77] F. M. Suchanek, S. Abiteboul, and P. Senellart. PARIS: probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3):157–168, 2011.

[78] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune bert for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer, 2019.

[79] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, 4(11):992–1003, 2011.

[80] N. Tang, J. Fan, F. Li, J. Tu, X. Du, G. Li, S. Madden, and M. Ouzzani. RPT: relational pre-trained transformer is almost all you need towards democratizing data preparation. *PVLDB*, 14(8):1254–1261, 2021.

[81] R. Trivedi, B. Sisman, J. Ma, C. Faloutsos, H. Zha, and X. L. Dong. Linknbed: Multi-graph representation learning with entity linkage. In *ACL*, 2018.

[82] S. Tyagi and E. Jimenez-Ruiz. Lexma: Tabular data to knowledge graph matching using lexical techniques. In *CEUR Workshop Proceedings*, volume 2775, pages 59–64, 2020.

[83] UK. Government open data. https://opendata.camden.gov.uk.

[84] L. G. Valiant. A bridging model for parallel computation. *CACM*, 1990.

[85] W3C. R2RML: RDB to RDF mapping language, Sept. 2012.

[86] W3C. Relational databases to RDF (RDB2RDF), Sept. 2012.

[87] Y. Wang, Z. Wang, Z. Zhao, Z. Li, X. Jian, H. Xin, L. Chen, J. Song, Z. Chen, and M. Zhao. Effective similarity search on heterogeneous networks: A meta-path free approach. *TKDE*, 2020.

[88] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *TKDE*, 25(5):1111–1124, 2013.

[89] Y. Wu, X. Liu, Y. Feng, Z. Wang, R. Yan, and D. Zhao. Relation-aware entity alignment for heterogeneous knowledge graphs. In *IJCAI*, pages 5278–5284, 2019.

[90] B. Yan, L. Bajaj, and A. Bhasin. Entity resolution using social graphs for business applications. In *ASONAM*, pages 220–227, 2011.

[91] Y. Yang. A study of thresholding strategies for text categorization. In *SIGIR*, pages 137–145, 2001.

[92] W. Yu, X. Lin, W. Zhang, J. Pei, and J. A. McCann. Simrank*: effective and scalable pairwise similarity search based on graph topology. *The VLDB Journal*, 28(3):401–426, 2019.

[93] H. Yun and B. Hwang. A pessimistic concurrency control algorithm in multidatabase systems. In *DASFAA*, volume 4, pages 379–386, 1993.

[94] Q. Zhang, Z. Sun, W. Hu, M. Chen, L. Guo, and Y. Qu. Multi-view knowledge graph embedding for entity alignment. In *IJCAI*, pages 5429–5435, 2019.

[95] S. Zhang, E. Meij, K. Balog, and R. Reinanda. Novel entity discovery from web tables. In *WWW*, pages 1298–1308, 2020.

[96] C. Zhao and Y. He. Auto-em: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning. In *WWW*, pages 2413–2424, 2019.

[97] Z. Zhao, X. Zhang, H. Zhou, C. Li, M. Gong, and Y. Wang. Hetnerec: Heterogeneous network embedding based recommendation. *Knowledge-Based Systems*, 204, 2020.

[98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *TODS*, 23(4):453–490, 1998.

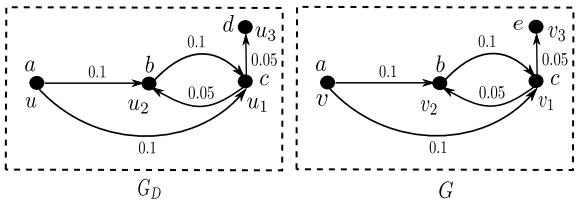

Fig. 7. An example for rechecking

APPENDIX

A. The Uniqueness of Parametric Simulation

There exists a unique maximum $\Pi(u_0, v_0)$ witnessing match (u_0, v_0), *i.e.,* $\Pi'(u_0, v_0) \subseteq \Pi(u_0, v_0)$ for all possible $\Pi'(u_0, v_0)$. We refer to the maximum $\Pi(u_0, v_0)$ as *the match of* (u_0, v_0) *in* (G_1, G_2) by simulation with parameters $(h_v, h_\rho, h_r, \sigma, \delta, k)$.

Proposition 4: *For graphs G_1 and G_2 and pair (u_0, v_0) for $u_0 \in G_1$ and $v_0 \in G_2$, there exists a unique maximum $\Pi(u_0, v_0)$ by simulation with parameters $(h_v, h_\rho, h_s, \sigma, \delta, k)$.* □

Proof: The existence of a match is ensured by Theorem 1. While the set $\Pi(u_0, v_0)$ of matches computed by algorithm ParaMatch may not be maximum, we can always extend the set $\Pi(u_0, v_0)$ to a maximum one since the number of possible matches are finite. That is, the maximum match always exists.

Below we show the uniqueness of the maximum match by contradiction. Assume that there exist two distinct maximum matches $\Pi_1(u_0, v_0)$ and $\Pi_2(u_0, v_0)$. Let $\Pi_3(u_0, v_0) = \Pi_1(u_0, v_0) \bigcup \Pi_2(u_0, v_0)$. Since $\Pi_1(u_0, v_0)$ and $\Pi_2(u_0, v_0)$ are distinct, $\Pi_3(u_0, v_0)$ has a larger size than both $\Pi_1(u_0, v_0)$ and $\Pi_2(u_0, v_0)$, *i.e.,* $\Pi_1(u_0, v_0) \subset \Pi_3(u_0, v_0)$ and $\Pi_2(u_0, v_0) \subset \Pi_3(u_0, v_0)$. We next show that $\Pi_3(u_0, v_0)$ witnesses the match (u_0, v_0), which contradicts that $\Pi_1(u_0, v_0)$ and $\Pi_2(u_0, v_0)$ are maximum. Therefore, the maximum match is unique.

It remains to show that $\Pi_3(u_0, v_0)$ witnesses the match (u_0, v_0). To this end, we prove that (1) $(u_0, v_0) \in \Pi_3(u_0, v_0)$; and (2) for each pair $(u, v) \in \Pi_3(u_0, v_0)$, the two conditions of parametric simulation (see Section III), denoted by $\mathcal{P}(\Pi_3(u_0, v_0), u, v)$, hold: (a) $h_v(u, v) \geq \sigma$; and (b) if u is not a leaf, then there exists a set $S^3_{(u,v)}$ of (u', v') that is a partial injective (1-to-1) mapping from V^k_u to V^k_v such that $\sum_{(u',v',e_i,\rho) \in S^3_{(u,v)}} h_e(e_i, \rho) \geq \delta$, and for each pair $(u', v') \in S^3_{(u,v)}$, $(u', v') \in \Pi_3(u_0, v_0)$. To simplify the proof, we define $\mathcal{P}(\Pi_1(u_0, v_0), u, v)$ and $\mathcal{P}(\Pi_2(u_0, v_0), u, v)$ similarly.

For (1), since both $\Pi_1(u_0, v_0)$ and $\Pi_2(u_0, v_0)$ witness the match (u_0, v_0), we know that both $(u_0, v_0) \in \Pi_1(u_0, v_0)$ and $(u_0, v_0) \in \Pi_2(u_0, v_0)$. From $\Pi_3(u_0, v_0) = \Pi_1(u_0, v_0) \bigcup \Pi_2(u_0, v_0)$, we have that $(u_0, v_0) \in \Pi_3(u_0, v_0)$.

For (2), it suffices to show that given any $(u, v) \in \Pi_3(u_0, v_0)$, condition $\mathcal{P}(\Pi_3(u_0, v_0), u, v)$ holds. Since $(u, v) \in \Pi_3(u_0, v_0)$ we have that $(u, v) \in \Pi_1(u_0, v_0)$ or $(u, v) \in \Pi_2(u_0, v_0)$. If $(u, v) \in \Pi_1(u_0, v_0)$, the conditions for $\mathcal{P}(\Pi_1(u_0, v_0), u, v)$ hold. Because $\Pi_3(u_0, v_0) = \Pi_1(u_0, v_0) \bigcup \Pi_2(u_0, v_0)$, we can verify that conditions for $\mathcal{P}(\Pi_3(u_0, v_0), u, v)$ hold. Indeed, for condition (a) we have that $h_v(u, v) \geq \sigma$, since the condition for $\mathcal{P}(\Pi_1(u_0, v_0), u, v)$ holds; for condition (b), since the condition for $\mathcal{P}(\Pi_1(u_0, v_0), u, v)$ holds, there exists a lineage set $S^1_{(u,v)}$ with aggregate score that is at least δ; then we can define the set $S^3_{(u,v)}$ as $S^1_{(u,v)}$, and verify that the condition (b) holds. Therefore, the conditions for $\mathcal{P}(\Pi_3(u_0, v_0), u, v)$ hold. The proof for the case that $(u, v) \in \Pi_2(u_0, v_0)$ is similar.

Putting these together, the maximum match is unique. □

B. Proof of Theorem 1

For the correctness of ParaMatch, we show that for any pair (u, v), $\mathsf{cache}(u, v) = [\mathsf{true}, \mathcal{W}]$ if and only if (u, v) is valid and $\mathcal{W}$ is a lineage set of (u, v) with the maximal aggregate score. If this holds, we can construct a witness $\Pi(u_0, v_0)$ for (u_0, v_0) by taking union of all pairs (u, v) with $\mathsf{cache}(u, v) = [\mathsf{true}, \mathcal{W}]$.

($\Rightarrow$) Assume that $\mathsf{cache}(u, v) = [\mathsf{true}, \mathcal{W}]$. Consider the following two cases. (a) When u is a leaf in G_D, we have that $h_v(u, v) \geq \sigma$ (line 1); then (u, v) is valid and the lineage set of (u, v) is $\emptyset$ by the definition of parametric simulation; in this case, the empty set $\emptyset$ is the one with maximum score; otherwise, (b) observe that ParaMatch searches the match of u following the descending order of l_u (line 16), and sets $\mathsf{cache}(u, v) = [\mathsf{true}, \mathcal{W}]$ once the aggregate score of $\mathcal{W}$ reaches δ (line 22); therefore, (u, v) is valid and $\mathcal{W}$ is a lineage of (u, v) with the maximum aggregate score.

($\Leftarrow$) Assume that (u, v) is valid and $\mathcal{W}$ is a lineage set of (u, v) with the maximum aggregate score. (i) If $\mathcal{W}$ is $\emptyset$, then u is a leaf in G_D and $h_v(u, v) \geq \sigma$ (condition (b) of parametric simulation in Section III). Thus $\mathsf{cache}(u, v) = [\mathsf{true}, \emptyset]$ (line 4). (ii) When $\mathcal{W} \neq \emptyset$, as (u, v) is valid and ParaMatch searches the match of u following the descending order of l_u (line 16), $\mathcal{W}$ will be finally identified by ParaMatch (line 23). □

C. Example for the Challenges of Parametric Simulation

Given G_D and G in Fig. 7, let σ=1, δ=0.1. Assume that all edges are picked by h_r, and are labeled with association scores, *e.g.,* 0.1 is the closeness between (u, v) and (u_1, v_1).

To check whether (u, v) is a match, one may want to first recursively check whether (u_1, v_1) is a match; to do this we in turn have to inspect candidates (u_2, v_2) and (u_3, v_3). Note that (u_1, v_1) and (u_2, v_2) form a strongly connected component and depend on each other. When checking (u_2, v_2), pair (u_1, v_1) has to be examined again. Checking these directly would be inefficient and may not even terminate.

To this end, we record the state of (u_1, v_1) and reuse it to avoid repeated checking. We initialize the state of (u_1, v_1) as true (*i.e.,* a match) and rectify it when it is invalidated. Note that rectification is necessary. Indeed, when (u_1, v_1) is assumed true, (u_2, v_2) becomes true since u_2 and v_2 bear the same label and the association between (u_2, v_2) and (u_1, v_1) is 0.1 = δ. However, later on (u_3, v_3) is found a non-match (*i.e.,* false), since they have distinct labels; then the state of (u_1, v_1) has to be changed to false. At this point, it is necessary to "clean up" the true state of (u_2, v_2) that was deduced from

Input: $G_D = (V_D, E_D, L_D)$, $G = (V, E, L)$, and a set FP of functions h_v, h_ρ, h_r and parameters σ, δ, k.
Output: The set Π of matches.
1. $\Pi := \emptyset$; $\mathcal{C} := \emptyset$; initialize the hashmap cache;
2. **for each** $u_t \in V_D$ and $v_g \in V$ such that $h_v(u_t, v_g) \geq \sigma$ **do**
3. add (u_t, v_g) to $\mathcal{C}$;
4. sort matches (u, v) in $\mathcal{C}$ in increasing orders of the degrees;
5. compute the set Π as VParaMatch;

Fig. 8. Algorithm AllParaMatch

the initial true of (u_1, v_1). The cleanup is a must since actually none of (u_2, v_2), (u_1, v_1) and (u, v) makes a match.

To implement these we employ a hashmap structure cache, to record both the current states of candidates and the dependencies among candidate matches. For each candidate match (u, v), cache$[u, v]$ is a pair $[\varphi, \mathcal{W}]$, which is either [false, $\emptyset$] or [true, $\mathcal{W}$], indicating whether (u, v) is invalid or valid, respectively. Here $\mathcal{W}$ is a set of candidate matches, and [true, $\mathcal{W}$] means that (u, v) is valid (*i.e.,* denoted by true) under the condition that all candidate matches in $\mathcal{W}$ are valid.

Using cache, we can get over the complications above as follows. (a) We first record the states of candidates (u_1, v_1) and (u_2, v_2) by setting cache$[u_1, v_1]$=[true, $\mathcal{W}_1$] and cache$[u_2, v_2]$ =[true, $\mathcal{W}_2$]=[true, $\{(u_1, v_1)\}$], respectively; here cache$[u_2, v_2]$ =[true, $\{(u_1, v_1)\}$] is to record the fact that the validity of (u_2, v_2) depends on that of (u_1, v_1); note that we can directly reuse these results during recursive calls; and (b) when (u_1, v_1) is confirmed invalid, we need to clean up the state of (u_2, v_2), since $(u_1, v_1) \in \mathcal{W}_2$ (*i.e.,* (u_2, v_2) depends on (u_1, v_1)).

D. Schema Matches

In addition to entity matches, HER can compute schema matches. Below we formulate schema matches and show how to extend algorithm ParaMatch to compute schema matches.

When it comes to graph $G = (V, E, L)$ and the canonical graph $G_D = (V_D, E_D, L_D)$ of a relational database $\mathcal{D}$ (Section II), we can deduce what paths in G represent important attributes A of a tuple t in $\mathcal{D}$. If u_t matches v_g, we deduce a set $\Gamma(u_t, v_g)$ of pairs (e, ρ) using score function h_ρ, where e is an edge from u_t encoding attribute A, and ρ is a path from v_g, such that path ρ encodes e (see Section V). We refer to $\Gamma(u_t, v_g)$ as the *schema matches* pertaining to (t, v_g).

Algorithm ParaMatch in Section V can be extended to compute $\Gamma(u_t, v_g)$, the schema matches pertaining to (u_t, v_g).

Observe the following. When ParaMatch returns true on (u_t, v_g), we get cache$(u_t, v_g) = [\text{true}, \mathcal{W}]$, where $\mathcal{W}$ is a set of pairwise matching properties of (u_t, v_g), *i.e.,* it consists of matches (u, v) for $u \in V_{u_t}^k$ and $v \in V_{v_g}^k$, along with paths ρ_D from u_t to its top-k descendant u and ρ_G from v_g to its top-k descendant v. Paths ρ_D are computed by function h_r and start with an edge e from u_t to its children (see Section IV). Here e may represent an attribute A of tuple t denoted by u_t, and the attribute is encoded by a prefix ρ_e of ρ_G.

For each such attribute A of t, if it is denoted by such an edge e, we deduce its "match" ρ_e from ρ_G as follows. We use function $\mathcal{M}_\rho$ (see Section IV) to pick ρ_e such that $\mathcal{M}_\rho(L_D(e), L(\rho_e))$ is the maximum among all prefixes of ρ_G. The path ρ_e is a "match" of e (attribute A).

Note that when an attribute B of t is picked by h_r as one of the top-k properties, it may not find a match in G. This is not surprising since graph G is heterogeneous from database $\mathcal{D}$ and it is not guaranteed to contain all properties of each entity in $\mathcal{D}$. Moreover, if B is not picked by h_r, it indicates that B is not a very important property of the entity after all.

Example 8: Continuing with Example 7, when ParaMatch terminates, schema matches $\Gamma(u_2, v_{10})$ is computed as follows. Since cache$[u_2, v_{10}]$=[true, $\mathcal{W}$] with $\mathcal{W}$=$\{(u_7, v_{20}), (u_8, v_{17}), (u_9, v_9), (u_{11}, v_{18})\}$, and u_7, u_8, u_9 and u_{11} are children of u_2, we can identify the "matches" of edges (attributes) (u_2, u_7), (u_2, u_8), (u_2, u_9) and (u_2, u_{11}) as follows.

(1) Edges (u_2, u_7), (u_2, u_8) and (u_2, u_{11}) in canonical graph G_D are mapped to edges (v_{10}, v_{20}), (v_{10}, v_{17}) and (v_{10}, v_{18}) in graph G, respectively, since $(u_7, v_{20}) \in \mathcal{W}$, $(u_8, v_{17}) \in \mathcal{W}$, $(u_{11}, v_{18}) \in \mathcal{W}$, and v_{20}, v_{17} and v_{18} are children of v_{10}.

(2) For edge $e_3 = (u_2, u_9)$, since $(u_9, v_9) \in W$, e_3 is mapped to path $\rho_G = (v_{10}, v_{15}, v_{19}, v_9)$. We check the prefixes of ρ_G, *i.e.,* $\rho_1 = (v_{10}, v_{15})$, ρ_2=(v_{10}, v_{15}, v_{19}) and ρ_3=$(v_{10}, v_{15}, v_{19}, v_9)$. Since $\mathcal{M}_\rho(L_D(e_3), L(\rho_1))$=0.46, $\mathcal{M}_\rho(L_D(e_3), L(\rho_2)) = 0.68$ and $\mathcal{M}_\rho(L_D(e_3), L(\rho_3)) = 0.71$, we know that $\mathcal{M}_\rho(L_D(e_3), L(\rho_3))$ is the maximum among all prefixes of ρ_G, and hence add (e_3, ρ_3) to $\Gamma(u_1, v_1)$. □

E. Examples for Algorithm VParaMatch

Continuing with Example 1, given an item "Dame Basketball Shoes D7" (tuple t_1), module VPair is then triggered to find all vertex matches of tuple t_1 in G. Assuming the same parameters $h_v, h_e, h_r, \sigma, \delta, k$ as in Example 7, VParaMatch (1) first finds items (*i.e.,* vertices) in G with names similar to "Dame Basketball Shoes D7" (*i.e.,* v_1 and v_3), and initializes $\mathcal{C}(u_1)$ with candidate matches (*i.e.,* (u_1, v_1) and (u_1, v_3)). (2) It then inspects candidates in $\mathcal{C}(u_1)$ along the same lines as Example 7, and returns (u_1, v_1).

We remark the following. (1) The verification starts from (u_1, v_1) since v_1 has the smaller degree than v_3; it confirms the validity of (u_2, v_{10}), which can be reused to verify other candidate matches. One can verify that inspecting candidates following the increasing degree order reduces comparisons. When G is large, we group vertices in G using inverted indices [98] on vertex attribute values for quick vertex search. Given vertex u_1 in G_D, the candidate matching vertices in G are v_1 and v_3. If we built inverted indices on the hasColor attribute of items in G, we can quickly locate the most similar vertex v_1, since its color is "White", which matches the color of u_1; while v_3 has color "Red", and then cannot match u_1.

F. Algorithm AllParaMatch

As shown in Fig. 8, AllParaMatch extends ParaMatch along the same lines as VParaMatch. The only difference is in the initialization phase (lines 2-3). That is, AllParaMatch initializes a set $\mathcal{C}$ of candidate pairs (u_t, v_g) across G_D and G, ranging over all $u_t \in V_D$ and $v_g \in V$ such that $h_v(u_t, v_g) \geq \sigma$. Then it extends Π in the same way as in VParaMatch (line 5).

Example 9: Continuing with Example 1, the e-commerce company runs AllParaMatch offline after (h_v, h_ρ, h_r) and (σ, δ, k) have been substantially improved, to identify items more accurately. When the process is triggered, AllParaMatch first finds all items from G_D and G, *i.e.,* u_1 for t_1 and u_{12} for t_3 (not shown) in G_D, along with v_1, v_3, v_{21}, v_{24} and v_{30} in G. It initializes $\mathcal{C}$ with candidate pairs, *e.g.,* (u_1, v_1), (u_1, v_3), (u_{12}, v_1) and (u_{12}, v_3); note that (u_1, v_{24}) and (u_{12}, v_{24}) are not in $\mathcal{C}$ due to the different labels of vertices. It then checks candidates in $\mathcal{C}$ along the same lines as Example 7.

Note that (u_1, v_3) and (u_{12}, v_1) are invalid and are not in Π, due to the difference between "Dame Basketball Shoes D7" and "Mid-cut Basketball Shoes Ultra Comfortable". That is, AllParaMatch distinguishes "Basketball Shoes" (*i.e.,* v_2 in G) denoted by v_1 and the one denoted by v_3. □

Proof of Corollary 2. We only prove the correctness and complexity of VParaMatch; AllParaMatch can be shown similarly.

(1) For the correctness, observe that the set $\mathcal{C}(u_t)$ consists of all possible candidates (lines 2-3) and VParaMatch verifies all candidates in $\mathcal{C}(u_t)$ along the same line as ParaMatch.

(2) For the complexity, the analysis is similar to the counterpart for ParaMatch. Observe the following: (a) algorithm VParaMatch takes $O(|V_D||E_D|+|V||E|)$ time to select top-k descendants; (b) it takes $O(|V_D||V|)$ time to check whether $(u_t, v) \in \Pi(u_t, v_g)$ for all candidates (u_t, v) in $\mathcal{C}(u_t)$; this is because VParaMatch uses the hashmap cache, and each pair (u, v) will be checked at most once; and (c) there exist at most $O(|V_D||V|)$ pairs; note that although VParaMatch only identifies all matches for a given vertex u_t, in the worst case all possible candidates need to be verified. Therefore, VParaMatch takes at most $O((|V_D|+|E_D|)(|V|+|E|))$ time. □

G. Fixpoint computation

Given fragmented graphs G_D and G, PAllMatch computes matches Π in parallel. It adopts the fixpoint model of GRAPE [35], [9], [8]. Under BSP, all workers perform APair on its local data in parallel. At the end of each superstep, all workers exchange messages, *i.e.,* the changed status of border nodes. By treating the messages as updates, all workers *incrementally* refine their local matches in parallel. The process proceeds until no more changes can be made. It can be formulated as fixpoint computation in supersteps, as follows:

$$R_i^0 = \mathsf{PPSim}(F_i, \sigma, \delta, k), \tag{3}$$

$$R_i^{j+1} = \mathsf{IncPSim}(R_i^j, F_i, \sigma, \delta, k, M_i). \tag{4}$$

Here R_j^i denotes the partial result at worker P_i after j rounds of computation; it consists of candidate matches identified at fragment F_i; and (2) M_i is the message sent to P_i from other workers. Algorithm PAllMatch starts with a procedure PPSim at each worker, and then iteratively runs IncPSim to incrementally refine the result, as shown in Section VI.

Proof of Theorem 3. This can be proved by constructing a tree $\mathcal{T}$ to represent the dependencies among candidates.

The tree $\mathcal{T}$. We start with the construction of $\mathcal{T}$, where the root is the given pair (u_0, v_0), and other nodes on $\mathcal{T}$ are candidate matches (u, v). The tree $\mathcal{T}$ is constructed top-down from (u_0, v_0). Given any (u, v) in $\mathcal{T}$, assume that $\mathsf{cache}[u, v] = [\mathsf{true}, \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}]$. Then we add (u_1, v_1), $(u_2, v_2), \ldots, (u_n, v_n)$ to $\mathcal{T}$ as the children of (u, v); intuitively, the children of (u, v) witness the match (u, v). The construction stops when either u is a leaf in G_D or a pair (u, v) has been verified before, *i.e.,* (u, v) has already appeared on the path from the root (u_0, v_0). The tree $\mathcal{T}$ is finite, since there exists at most $O(|G_D||G|)$ candidates, and each candidate can appear at most twice on each path from the root (u_0, v_0).

By algorithm ParaMatch shown in Fig. 4, we can verify that given a candidate (u_0, v_0), ParaMatch returns true if and only there exists such a tree $\mathcal{T}$ rooted at (u_0, v_0).

Correctness. It suffices to show that for each pair (u_t, v_g), sequential algorithm ParaMatch returns true if and only if the parallel algorithm PAllMatch returns true.

Before showing this, we first establish a connection between ParaMatch and PAllMatch. Assume that $\mathcal{T}$ is the tree constructed for (u_t, v_g) when ParaMatch runs on G_D and G; and $\mathcal{T}_i^1, \mathcal{T}_i^2, \ldots, T_i^{m_i}$ are trees constructed when PAllMatch runs on fragment F_i $(i \in [1, n])$; observe that $\mathcal{T}_i^1, \mathcal{T}_i^2, \ldots, T_i^{m_i}$ may not be connected, since G and G_D have been partitioned via edge-cut. But due to the use of border nodes in O_i^D (see Section VI), $\mathcal{T}$ can be obtained by merging $\mathcal{T}_i^1, \mathcal{T}_i^2, \ldots, T_i^{m_i}$ $(i \in [1, n])$. Note that when PAllMatch terminates, the cached values $\mathsf{cache}[u, v]$ in different trees $\mathcal{T}_i^1, \mathcal{T}_i^2, \ldots, T_i^{m_i}$ are consistent, since these values are synchronized during the running of PAllMatch via messages (see Section VI).

Then the correctness can be verified as follows. ParaMatch returns true if and only if $\mathcal{T}$ can be constructed from ParaMatch if and only if $\mathcal{T}$ can be obtained by merging $\mathcal{T}_i^1, \ldots, T_i^{m_i}$ $(i \in [1, n])$ if and only if PAllMatch returns true. □

F-measure	GloVe 100d	GloVe 200d	GloVe 300d
DBpediaP	0.807	0.835	0.852
DBLP	0.881	0.902	0.915
IMDB	0.872	0.891	0.899

TABLE VII
ACCURACY OF HER WITH DIFFERENT EMBEDDINGS (F-measure)

H. *Experimental Results on* IMDB

We report the scalability and efficiency of HER in the APair mode on IMDB in Figure 9. From the figure, we find the following. (1) As the number n of workers increases from 4 to 16, APair becomes 2.3 times faster on IMDB (see Fig. 9(a)); and (2) Figures 9(b)-9(d) show the efficiency of APair on IMDB with 16 workers and various parameter settings, which show that larger k or δ increases the execution time while larger σ decreases the matching time. These results are consistent with those on the other datasets in Section VII.

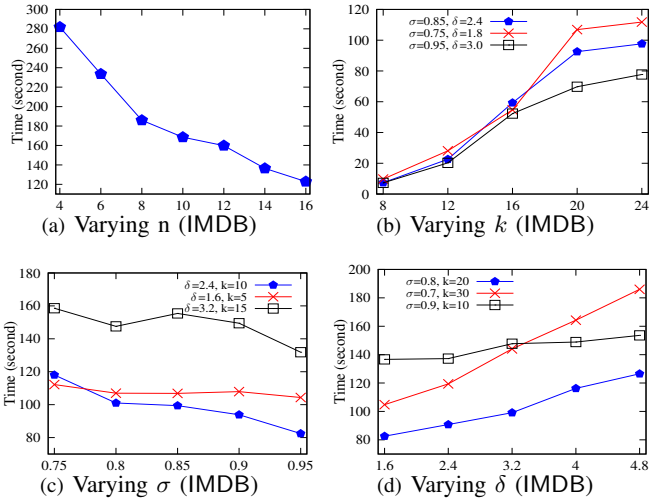

Fig. 9. HER scalability and efficiency on IMDB

I. *Impact of Different Embeddings on* HER *Accuracy*

In order to study the influence of different embeddings in the vertex model $\mathcal{M}_v$ on HER matching accuracy, we replaced the sentence Bert in $\mathcal{M}_v$ with a series of GloVe word embeddings [70] of different vector dimensions, and tested the accuracy of HER accordingly. More specifically, given the pre-trained GloVe embedding, we adopted the average embedding vector of each word in a vertex attribute as the vertex similarity measure, and tested the matching accuracy on IMDB, DBLP and DBpediaP with the same HER parameter configuration. A series of four GloVe word embeddings pre-trained on Wikipedia 2014 and Gigaword 5, with vector dimensions of 100, 200 and 300, respectively, were employed [17]. The reason for choosing these embeddings to compare is to control other influencing factors such as training corpus and hyper-parameter configurations, and leave the word similarity measurement accuracy of an embedding (determined by the dimension of word embeddings since GloVe embedding with higher dimensions have higher accuracy [70]) as the only varying factor. The overall accuracy of 300-dimensional GloVe word embedding on word analogy task of Mikolov et al. (a variety of word similarity tasks) [60] is 71.7%, while that of 100-dimensional embedding is 60.3% [70].

As shown in Table VII, the higher word similarity measurement accuracy (*i.e.,* higher dimensions of embeddings) is, the higher matching accuracy HER can achieve. Thus, embeddings with high word similarity measurement accuracy are desirable for HER. This said, the accuracy gap between high and low dimensional embeddings is quite small (at most 5% difference in accuracy as shown in Table VII), which means that the matching accuracy of HER is rather insensitive to the choice of embeddings. This is because HER computes scores from multiple matching paths, and a single failure of the embedding similarity has little impact on the matching result.