THE UNIVERSITY *of* EDINBURGH

Learning, Deducing and Linking Entities

Resul Tugay

Doctor of Philosophy

Data Science and Artificial Intelligence

School of Informatics

University of Edinburgh

2023

Abstract

Improving the quality of data is a critical issue in data management and machine learning, and finding the most representative and concise way to achieve this is a key challenge. Learning how to represent entities accurately is essential for various tasks in data science, such as generating better recommendations and more accurate question answering. Thus, the amount and quality of information available on an entity can greatly impact the quality of results of downstream tasks. This thesis focuses on two specific areas to improve data quality: (i) learning and deducing entities for data currency (*i.e.,* how up-to-date information is), and (ii) linking entities across different data sources.

The first technical contribution is GATE (Get the lATEst), a framework that combines deep learning and rule-based methods to find up-to-date information of an entity. GATE learns and deduces temporal orders on attribute values in a set of tuples that pertain to the same entity. It is based on creator-critic framework and the creator trains a neural ranking model to learn temporal orders and rank attribute values based on correlations among the attributes. The critic then validates the temporal orders learned and deduces more ranked pairs by chasing the data with currency constraints; it also provides augmented training data as feedback for the creator to improve the ranking in the next round. The process proceeds until the temporal order obtained becomes stable.

The second technical contribution is HER (Heterogeneous Entity Resolution), a framework that consists of a set of methods to link entities across relations and graphs. We propose a new notion, 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.

Rather than solely concentrating on rule-based methods and machine learning algorithms separately to enhance data quality, we focused on combining both approaches to address the challenges of data currency and entity linking. We combined rule-based methods with state-of-the-art machine learning methods to represent entities, then used representation of these entities for further tasks. These enhanced models, combination of machine learning and logic rules helped us to represent entities in a better way (i) to find the most up-to-date attribute values and (ii) to link them across relations and graphs.

Acknowledgements

I am deeply grateful to my primary supervisor, Professor Wenfei Fan, for his unwavering support, guidance, and encouragement throughout my PhD journey. Working with him has been an incredible privilege, and I consider myself lucky to have had such a dedicated and accomplished mentor. He has always amazed me for his exceptional academic achievements, strong work ethic, and commitment to excellence. I am truly appreciative of his kindness, support, and productive discussions. Thank you, Professor Fan, for your invaluable guidance and mentorship.

I would like to express my gratitude to my second supervisor, Yang Cao. I can not explain in words but Dr Yang Cao is and was more than a second supervisor for me. His calmness and wisdow gave me the strength when l fell down. He has patiently guided me through personal and academic challenges. Thank you, Yang Cao, for your guidance and mentorship throughout my academic journey.

I would also like to thank all members in my research group including Dr. Rouchun jin, Muyang Liu, Dr. Yuanhao Li, Dr. Ping Lu, Wenzhi Fu, as they are very kind and helpful.

I was fortunate enough to have, Nick McKenna, Asif Khan, Eric Munday, Ibrahim Abu Farha, Nikita Moghe, Ondrej Bohdal, Adarsh Prabhakaran, Faheem Kirefu, Maurice Bailleu as my colleagues in the office. I appreciate the pleasant working atmosphere and enjoyable lunch times we shared. Also, l want to thank to my friend, Mehmet Aygun, for making my time in Edinburgh fun and joyable.

I especially thank the examiners of my thesis: Dr. Milos Nikolic and Dr.Nikos Ntarmos, for taking time to read my thesis and providing valuable and constructive feedbacks.

I would also like to thank the Republic of Türkiye, Ministry of National Education for their full support, which made my studying abroad possible.

Lastly, I would like to thank my all family for their support. I am especially grateful to my wife, Burcu Tugay for her unconditional love, encouragement and support. Words cannot express how much her support has meant to me. I dedicate this thesis to her and our beloved son Abdullah with all my love. Thank you both for being the light in my life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Resul Tugay*)

To my wife and son,

Table of Contents

Chapter 1

Introduction

The amount of data being generated is increasing at an unprecedented rate, which highlights the importance of considering both its quantity and quality. As of June 2022, YouTube receives over 500 hours of video content every minute on their platform (Youtube, 2022), while Facebook and TikTok have more than 2.96 billion and 1 billion monthly active users, respectively (Meta, 2022; TikTok, 2022). As the volume of data continues to grow at an astonishing rate, it's becoming increasingly crucial to ensure that the data is of high quality and accuracy.

In practice, data obtained from real-world sources can often be of poor quality, exhibiting various issues such as inconsistency, duplication, staleness, inaccuracy, and incompleteness. As a result, relying on such data can lead to poor decision-making, increased costs, and lost opportunities. For example, outdated customer data can result in missed sales opportunities and reduced customer satisfaction, ultimately leading to lost revenue. According to a study by Gartner (gar, 2022), poor data quality is a significant factor in the success or failure of 40% of business initiatives. It is also estimated that inaccurate customer data costs organizations 6% of their annual revenues (Royal Mail, 2018). These facts highlight the importance of data quality. There are several methods for enhancing data quality, such as ensuring consistency, accuracy, removing duplicates, addressing incompleteness, and so forth.

We studied two methods to improve data quality, data currency and entity resolution. The former refers to the timeliness or recency of the information in a dataset. It addresses the question of how up-to-date and relevant the data is at a given point in time. For example, consider a customer database for an e-commerce company. If the information in the database is not regularly updated, it might contain outdated details about customers' preferences, contact information, or purchasing behavior. This can

lead to issues like sending promotional offers to inactive email addresses or mailing catalogs to outdated physical addresses Royal Mail (2018). The data currency problem becomes especially critical in applications where timely information is crucial for decision-making, such as in financial transactions, healthcare, emergency services, and real-time analytics. Addressing data currency involves implementing strategies to ensure that data remains accurate and relevant over time. The latter is Entity resolution, also known as record linkage or deduplication, is a process used in data management to identify and merge duplicate or related records within a dataset. The goal is to create a unified and accurate view of the entities (e.g., individuals, companies, products) represented in the data. For example, consider a database containing information about customers. Due to various factors like data entry errors, multiple entries may exist for the same person. Entity resolution helps identify which records actually pertain to the same real-world entity. This is crucial for tasks such as customer relationship management, fraud detection, and data integration. The process involves comparing attributes (like names, addresses, etc.) across different records to determine their similarity. Entity resolution is a fundamental task in data quality management and is applied in various domains, including healthcare (matching patient records), finance (detecting duplicate transactions), and e-commerce (eliminating duplicate product listings). It's crucial for ensuring accurate decision-making and analysis based on clean, consolidated data. Next, we will provide a more detailed description of each of the methods.

1.1 Data Currency

Data currency is a fundamental technique for enhancing data quality. Its primary goal is to identify the most up-to-date values for entities represented by tuples (Fan et al., 2013a, 2014a). The issue of data currency would be simple if valid timestamps were available for data values, but in practice, timestamps are frequently absent or incorrect. Two approaches have been proposed to solve this problem in the literature: (1) applying ranking models based on deep learning (Burges et al., 2005; Han et al., 2020; Pasumarthi et al., 2019; Tay et al., 2017) or reinforcement learning (Hu et al., 2018; Yao et al., 2021)), and (2) using logic rules such as currency constraints (CCs) to deduce temporal orders (Ding et al., 2020; Duan et al., 2020; Fan et al., 2014b, 2012; Li et al., 2018; Liang et al., 2019; Wang et al., 2018).

Ranking models focus on training models to efficiently order a set of items based on their relevance to a given query. (Freund et al., 2003) introduced a novel approach

to learning to rank, leveraging gradient descent techniques to optimize ranking models. In this approach, the RankBoost algorithm formulates ranking as a binary classification problem and uses boosting techniques to improve the ranking quality. This approach has been widely adopted and extended in subsequent research. Another important approach in this domain is the RankNet algorithm proposed by Burges et al. (Burges et al., 2005). RankNet employs neural networks to learn a ranking function, demonstrating impressive performance in various applications, including information retrieval and recommendation systems. Furthermore, RankNet's success paved the way for the development of deep learning-based ranking models. Techniques like LambdaRank and LambdaMART (Burges, 2010) have further advanced the field by incorporating advanced neural network architectures and optimization strategies. However, it is difficult to determine if the ranking aligns with the temporal orders in the real world, which is important for reliable data-driven decision making. Additionally, these models cannot explain the ranking of objects that have a complex interlinked structure (*e.g.,* job title depends on many attributes including salary, company etc.).

On the other hand, using CCs as logic rules can help to deduce temporal orders among attribute values. As an example of a logic rule, one can observe that an individual's monthly income typically rises as their years of professional experience. Another example can be direct inference such that marital status only changes from single to married, rather than in the opposite direction (Canada, 2022). The study of currency constraints began with the research in (Fan et al., 2012), which initially utilized partial currency orders. This concept was later expanded upon in (Fan et al., 2013b, 2014b) to address conflict resolution, taking into account both CCs and conditional functional dependencies (CFDs) (Fan et al., 2008). In (Li and Li, 2016), a category of rules known as currency repairing rules (CRRs) was introduced, combining logic rules with statistical techniques. Another method, outlined in (Ding et al., 2017), established a two-step process to assess the currency of dynamic data, employing a topological graph.

Furthermore, UncertainRule (Li et al., 2018) introduced a set of rules considering both temporal sequences and data certainty, contributing to the treatment of uncertain currency. The frameworks ImproveC (Ding et al., 2018) and Imp3C (Ding et al., 2020) take a comprehensive approach to data cleaning, encompassing consistency, completeness, and currency. They employ a metric to rectify noisy data using CFDs (Fan et al., 2008) and currency orders. Additionally, (Duan et al., 2020) explores parallel incremental updating algorithms by applying traditional currency rules.

Although these approaches offer valuable strategies to enhance data quality, it can be difficult to find enough rules to deduce the orders for all possible pairs of values. Furthermore, it is challenging to generalize rules to handle values that are similar in meaning but different in wording, especially when the tuples are extracted from different sources. For example, ”married” and ”wedded” both refer to the same marital status but may appear differently in different data sources. The use of either machine learning or logic rules alone is insufficient in solving the problem of deducing temporal orders. Hence it raises the question of whether a uniform framework can be developed by combining both approaches.

In this thesis, we combine logic rules with deep learning techniques to determine the currency of an entity with the help of determining temporal orders on attributes. We propose a creator-critic framework that not only learns temporal orders via deep learning, but also adopts rules to deduce more temporal orders and help the models learn better. The creator learns temporal orders based on contexts and iteratively improves its model with augmented training data from the critic. Then the critic employs rules and provide more training data to the creator. Our results demonstrate a substantial increase in accuracy compared to models solely employing either deep learning or rule-based approaches. Specifically, when considering performance against rule-based methods, our system called GATE showcases superiosr generalizability by effectively learning from previously unseen data. This is attributed to the interaction between deep learning's capacity for feature extraction and logic rules. Furthermore, in comparison to models on machine learning, GATE improves the results with incremental training data by logic rules.

1.2 Entity Resolution

Entity Resolution (ER), also known as record linkage or entity linking, is another technique to improve data quality, is the task of identifying and linking entities in a dataset that refer to the same real-world entity (Herzog et al., 2007; Winkler, 2014). The goal of entity linking is to eliminate duplicate records, resolve inconsistencies, and merge related data from multiple sources. ER has primarily been studied for relational data that is specified by a schema. One widely used approach for relational data is rule-based matching, where predefined rules are applied to compare attributes and determine if two records refer to the same entity (Hernández et al., 2013). State-of-the-art relational ER systems also employ machine learning (Konda et al., 2016)

and deep learning models (Mudgal et al., 2018) to improve their accuracy. These approaches, both supervised and unsupervised, have shown promise in entity resolution. Supervised methods learn from labeled data to predict matches, while unsupervised methods use clustering techniques to group similar records (Getoor and Machanavajjhala, 2012). In addition to relational data, ER has also been studied for graphs and it has mainly relied on rule-based (Fan et al., 2015) or ML-based (Kwashie et al., 2019) approaches. Although ER has been studied on graphs and relational data separately, it is harder to correlate entities when the data sources are different, especially one from a relational database $\mathcal{D}$ and the other from a graph G. 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. With all that, the need for studying this problem 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 (Nargesian et al., 2019). We propose an algorithm called parametric simulation with score functions learned from machine learning models to link entities across relations and graphs. It can assess the semantic closeness of entities by recursively inspecting properties linked to entities via paths.

Specifically, we use language models like BERT (Devlin et al., 2019a) to capture the sequential information embedded in vertex and edge labels. We offer three distinct modes in parametric simulation algorithm for entity linking, namely Spair, Vpair, and Apair. In Spair mode, we determine whether a given pair of entities match. Vpair mode involves examining all potential matches between a given entity e in relational data D and all entities in graph G. Apair mode extends this evaluation to inspect all possible matches between "D" and "G." Additionally, we provide a parallelized version of parametric simulation for the Apair mode to handle large graphs efficiently. By incorporating parametric simulation and machine learning in global topological matching, HER achieves quadratic-time efficiency between relational and graph data. Furthermore, we have developed parallel parametric simulation algorithm for large datasets. Through rigorous experimentation, we have demonstrated that HER offers a highly promising solution, excelling in terms of accuracy, scalability, and overall efficiency.

1.3 Thesis Structure

The structure of this thesis is organized as follows. Chapter 2 provides the necessary background and an overview of prior works related to data currency and linking entities. Section 2.1 presents an overview of data currency of entities, including both rule-based and machine learning-based methods, and discusses temporal orders in this aspect. In Section 2.2, the background of entity resolution and graph simulation is provided, which are required for understanding Chapter 4.

Chapter 3 focuses on data currency. We identify the latest attribute values of an entity by determining temporal orders on attribute values in a set of tuples that pertain to the same entity, in the absence of complete timestamps. We propose a creator-critic framework to learn and deduce temporal orders by combining deep learning and rule-based deduction, referred to as GATE (Get the lATEst). This framework has been open-sourced at https://github.com/resultugay/GATE.

Chapter 4 focuses on entity linking. We propose 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 use machine learning methods to learn the parameter functions and thresholds. We also develop a system, denoted by HER, 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. This system has been open-sourced at https://github.com/resultugay/her-toy.

1.4 List of Publications

The main contributions of this thesis are published in the following top conferences.

1. Chapter 3 introduces GATE and is based on the following paper:
   Wenfei Fan, Resul Tugay, Yaoshu Wang, Min Xie, Muhammad Asif Ali. Learning and Deducing Temporal Orders. *International Conference on Very Large Data Bases (VLDB)*, 2023
   As a co-author, I played a significant role in the development of the parametric simulation algorithm, curated the UKGOV dataset, assisted in conducting experiments, and made contributions to the paper's writing and participated in various discussions and proofread the paper.

2. Chapter 4 introduces HER and is based on the following paper:
   Wenfei Fan, Liang Geng, Ruochun Jin, Ping Lu, Resul Tugay, Wenyuan Yu. Linking Entities across Relations and Graphs. *International Conference on Data Engineering (ICDE)*, 2022
   As a co-author, I introduced the learning to rank approach to address the data currency problem. I personally developed all the code for the creator component and carried out initial experiments. Additionally, I was responsible for finding and cleaning the datasets (except COM dataset), made contributions to the paper's writing and participated in various discussions and proofread the paper.

Chapter 2

Background and Related Work

In this chapter, we provide the necessary background and related work for understanding two studied problems: data currency and entity resolution. To tackle data currency problem, various approaches and techniques have been proposed, ranging from the application of ranking models based on deep learning to leveraging logic rules like currency constraints. We delve into these methods and their respective strengths and limitations, setting the stage for a comprehensive exploration of data currency enhancement. Then we provide background and related work for entity resolution. We discuss the significance of entity resolution across various domains, and present an overview of the techniques employed, including rule-based matching and advanced ML approaches. By delving into both data currency and entity resolution, we do groundwork for the subsequent chapters, where we propose our approaches to address these challenges.

2.1 Data Currency

The data currency is the problem of identifying the current value of an entity in a database in the absence of reliable timestamps. For instance, in a retail company's customer database, if the contact details of customers are not routinely updated, the database may contain outdated email addresses or phone numbers. This could lead to missed communications, failed deliveries, and ultimately, dissatisfied customers. Hence, the data currency problem highlights the necessity of implementing strategies and processes to keep information current, thus enhancing the overall quality and utility of the dataset. Consider a set of tuples that pertain to the same entity e. Is it possible to determine the temporal orders on the attribute values, *i.e.,* for tuples t_1

and t_2, whether the value in the A-attribute of t_1 is more current than the value in the A-attribute of t_2, denoted by $t_2 \prec_A t_1$, in the absence of complete timestamps?

One may want to approach this problem by training a ranking model that sorts objects according to their degrees of relevance or importance (Liu, 2010) using either machine learning (ML) (Burges et al., 2005) or reinforcement learning (Hu et al., 2018). These approaches have also been used in search engine (Chapelle and Chang, 2011) and machine translation (Zhang et al., 2016; Duh, 2009) to rank objects or entities. By means of ranking models one can learn temporal orders and decide whether $t_1 \prec_A t_2$ for *all* tuples t_1, t_2 and attribute A. However, it is hard to justify whether the ranking conforms to the temporal orders in the real world. For data-driven decision making, we need to ensure that the learned orders are reliable. Moreover, these approaches cannot explain the ranking of objects that follow a complex attribute relations.

Another approach is to employ logic rules, *e.g.,* CCs (Ding et al., 2020; Duan et al., 2020; Fan et al., 2014b, 2012; Li et al., 2018; Liang et al., 2019; Wang et al., 2018). These rules help us deduce temporal orders. For instance, the shoe sizes of the same person typically monotonically increases (before 20 years old), and the address of a person may be associated with the marital status (once the marital status changes, the address also changes).

Using CCs, one can deduct up-to-date attribute values by looking at other attributes. However, it is challenging to find enough rules to deduce relative orders on *each and every* pair of values. Besides, it is difficult to generalize rules to handle lexically different but semantically similar values, since tuples might be extracted from different source (*e.g.,* marital status: married vs. wedded). Even though employing semantic representations like word embeddings to compare words within a rule, this approach alone may still fall short in identifying the most current attribute values. This limitation arises from the absence of consideration for other relevant attributes.

Neither ML nor logic rules work well when used separately. A natural question is whether it is possible to combine ML models and logic rules in a uniform framework, such that we can learn temporal orders, and use the rules to validate the ranking and improve the learning? How well can this framework improve the accuracy of the ML models and logic rules when being used alone? Thus, we next categorize the related work for learning and deducing entities as follows.

2.1.1 Rule Based Methods

Currency constraints were first studied in (Fan et al., 2012) by employing partial currency orders and later extended in (Fan et al., 2013b, 2014b) for conflict resolution, by considering both CCs and conditional functional dependencies (Fan et al., 2008). A class of currency repairing rules was proposed in (Li and Li, 2016), which combines logic rules and statistics. A two-step approach was developed in (Ding et al., 2017) to determine the currency of dynamic data, by means of a topological graph. A class of uncertain currency rules was supported by UncertainRule (Li et al., 2018) that considers both temporal orders and data certainty. Improve3C (Ding et al., 2018) and Imp3C (Ding et al., 2020) are 4-step data cleaning frameworks, including data consistency, completeness and currency, which use a metric to repair noisy data using (Fan et al., 2008) and currency orders. There has also been work (Duan et al., 2020) on

parallel incremental updating algorithms by employing traditional currency rules.

Our work in Chapter 3 differs from the prior work as follows. (a) To the best of our knowledge, we make the first effort to combine deep learning and logic rules for determining currency, for the two to enhance each other, while the prior work at most extends rules with statistic. In particular, we employ the classic chase algorithm (Fan et al., 2020) for handling logic rules, making sure it ensures the Church-Rosser property. This process involves identifying tuple pairs (or valuations) associated with a given temporal order and then inferring new ranked pairs based on this information. (b) We propose a deep learning model for inferring temporal orders and use logic deduction to derive more ranked pairs.

2.1.2 ML Models

There have also been efforts on inferring temporal orders by ML models (Goyal and Durrett, 2019; Bramsen et al., 2006; Chambers and Jurafsky, 2008; Tourille et al., 2017; Ning et al., 2017, 2018). A related topic is to incorporate temporal information into knowledge graphs, *e.g.,* for temporal link predictions (Divakaran and Mohan, 2020; Sadeghian et al., 2021) and reasoning (Trivedi et al., 2017; Zhang et al., 2020a; Dikeoulias et al., 2022). These methods often embed temporal information in ML models, for inference varying over time. Learning temporal orders has been modeled as a learning-to-rank problem (pointwise, pairwise and listwise), where global orders are learned for currency attributes. Temporal problems have also been studied in, *e.g.,* search and recommendation (Xu et al., 2020; Fan et al., 2022), information

retrival (Liu, 2009) and natural language processing (Nogueira and Cho, 2019; Li et al., 2020b). In particular, ranking in natural languange processing (*e.g.,* in question-answering tasks) can be handled by pre-trained language models with cross-entropy loss or by a ranking function with a binary classifier as the comparison operator.

2.2 Linking Entities across Relations and Graphs

Entity resolution, also known as record linkage or deduplication, is the process of identifying and merging duplicate or related records in a dataset to create a unified and accurate view of entities (e.g., individuals, companies) represented in the data. This involves determining which records in the dataset refer to the same real-world entity, even if they have slight variations or discrepancies. For example, in a customer database, there may be multiple entries for the same person with slightly different information (e.g., misspellings, variations in addresses). Entity resolution helps identify and merge these entries to create a single, accurate representation of each customer.

Furthermore, entity resolution is a complex and longstanding challenge to the data management community" (Golshan et al., 2017), as it involves dealing with large volumes of data, noisy and incomplete information, and various types of errors and ambiguities. ER can be done on relational data, graph data or both. We conduct ER across relational data and graphs in this thesis. Hence we categorize the related work as follows. First, we review previous research on ER in the context of relations and graphs. Then, we discuss related work on graph simulation, which is relevant to our proposed method of parametric simulation.

2.2.1 Entity Resolution

ER has primarily been studied for relational data that is specified by a schema (see (Christophides et al., 2019) for a survey). ER systems can be categorized into (1) *rules*, *e.g.,* keys (Fan et al., 2015) and graph differential dependencies (Kwashie et al., 2019); (2) *ML models*, *e.g.,* (Saeedi et al., 2018) adopts unsupervised clustering and matches vertices in the same cluster; (Trivedi et al., 2018) employs deep neural networks; (Dong et al., 2017) and (Zhao et al., 2020) make use of vertex embedding based on heterogeneous skip-gram and co-occurrence; (Ngomo and Auer, 2011) exploits triangle inequality for blocking, and conducts ER based on the metric; (Isele et al., 2010) aggregates value similarities via link conditions; (Jeh and Widom, 2002; Yu et al.,

2019; Kusumoto et al., 2014; Wang et al., 2020) compute SimRank scores to conduct vertex matching.

There also exists work on heterogeneous ER (Papadakis et al., 2012, 2018; Li et al., 2020a; Zhang et al., 2020b; Sun et al., 2011; Fu et al., 2020). But their notions of heterogeneity are quite different from ours. In (Li et al., 2020a), it means the (relational) schema heterogeneity. (Sun et al., 2011) and (Fu et al., 2020) target heterogeneous network with multiple typed objects and links. In (Papadakis et al., 2012), it comes from loose schema binding. JedAI (Papadakis et al., 2018) 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 (Ngomo and Auer, 2011). While (Zhang et al., 2020b) 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 works well across relations $\mathcal{D}$ and graphs G. Relational ER methods rely on the known structure of schema, 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. ER methods for graphs target entities with similar topological structures, while $\mathcal{D}$ and G often have radically different structures even for the same entity. Moreover, these methods and those heterogeneous ones explore only local properties, but 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.

Our work in Chapter 4 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 embed ML models in topological matching, by proposing parametric simulation with score functions learned with ML models, to cope with the heterogeneity of $\mathcal{D}$ and G. It is beyond conventional ML methods, and cannot be expressed as rules since parametric simulation is inductively defined with ML models and aggregate scores, beyond the expressive power of first-order logic. (c) It "globally" assesses the semantic closeness by recursively inspecting properties (descendants) linked to entities via paths, as opposed to checking labels, attributes and other local neighbors of vertices.

2.2.2 Graph Simulation

Graph simulation is a fundamental problem in the field of graph theory and has found applications in various domains. Several approaches and techniques have been proposed to address this problem.

Graph Simulation Algorithms: Graph simulation algorithms aim to determine if there exists a mapping between nodes of two graphs that preserves certain relationships, such as the reachability or connectivity between nodes. This notion has been extended to map edges to paths, *e.g.,* bounded (Fan et al., 2010) and strong simulation (Ma et al., 2014). Other notions for graph matching, *e.g.,* subgraph isomorphism and homomorphism (see (Gallagher, 2006) for a survey), are too strong to match entities with different topological structures; worse yet, they incur intractability (Garey and Johnson, 1979)

Graph Isomorphism: One of the classic problems related to graph simulation is graph isomorphism, where the goal is to determine if two graphs are isomorphic, meaning they have the same structure. Various algorithms, such as the Ullmann algorithm and the VF2 algorithm, have been developed to solve this problem efficiently (Ullmann, 1976).

Subgraph Matching: Subgraph matching is a related problem where the goal is to find all occurrences of a given pattern graph within a larger data graph. This problem has applications in database querying and network analysis. Techniques like the subgraph isomorphism algorithm and subgraph matching with augmented data structures have been proposed for efficient subgraph matching (Cordella et al., 2004).

Graph Neural Networks: Recent advancements in deep learning have led to the development of graph neural networks, which can perform tasks like node classification, link prediction, and graph classification. They leverage graph simulation principles to propagate information across nodes in a graph and capture complex dependencies (Kipf and Welling, 2016; Bronstein et al., 2017).

Parametric simulation radically differs from graph isomorphism, simulation (bounded, strong) and graph neural networks 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.

Chapter 3

Learning and Deducing Entities

In this chapter, we focus on the problem of data currency, which refers to the challenge of determining the most current or up-to-date values of attributes associated with a given entity, based on a set of tuples pertaining to the same entity. To tackle this problem, we rely on the concept of temporal orders, which involves deducing the order of events or changes to the attributes of a given set of tuples over time. In many cases, these tuples may not have complete timestamps, which can make it difficult to accurately identify the temporal order of attribute values.

3.1 Introduction

Real-life data keeps changing. As reported by Royal Mail, on average, "9590 households move, 1496 people marry, 810 people divorce, 2011 people retire and 1500 people die" each day in the UK (Royal Mail, 2018). It is estimated that inaccurate customer data costs organizations 6% of their annual revenues (Royal Mail, 2018). Outdated data incurs damage not only to Royal Mail. When the data at a search engine is out of date, a restaurant search may return a business that had closed three years ago. When the data about the condition of infrastructure assets is obsolete, it may delay the maintenance of equipment and cause outage. Moreover, data-driven decisions based on outdated data can be worse than making decisions with no data (Little, 2020). Indeed, "as a healthcare, retail, or financial services business you cannot afford to make decisions based on yesterday's data" (Exasol, 2020). Unfortunately, "82% of companies are making decisions based on stale information" (Businesswire, 2022).

These highlight the need for determining the currency of data, *i.e.,* how up-to-date the information is. This is, however, highly nontrivial. Consider a set of tuples that

pertain to the same entity. Their attribute values may become obsolete and inaccurate over the time. Worse yet, only *partial* reliable timestamps might be available, where a timestamp is *reliable* if it is *precise*, *correct* and moreover, it indicates that at the time, the values are *correct* and *up-to-date*. Apart from mechanical reasons (malicious attacks or hardware failures), the logical reasons below are the major temporal issues in data quality (Bose et al., 2013), and account for the absence of reliable timestamps.

(1) Missing timestamps. Timestamps may simply not be recorded, *e.g.,* in an e-health database (Kurniati et al., 2019), only 16 out of 26 relations are timestamped. Even when a relation has timestamps, it may not be complete, *e.g.,* 25.36% missing in (Martin et al., 2019), up to 82.28% in (Kurniati et al., 2019).

(2) Imprecise timestamps. Timestamps may be too coarse, leading to unreliable ordering. An e-form may be submitted multiple times to an office automation system by employees during a day. If the forms are recorded in datestamps, it is not clear which form (all on the same day) is the latest. Similar problems are often encountered in hospital data, where only dates are recorded (Bose et al., 2013). Another example concerns inconsistent granularity of timestamps (*e.g.,* minutes vs. days (Bose et al., 2013; Kurniati et al., 2019)). If two values have timestamps "12-8-2021" and "12-8-2021 20:41", it is not clear which one is more up-to-date. As reported in (Martin et al., 2019), 90.41% of appointment records have imprecise timestamps.

(3) Incorrect timestamps. Many factors can lead to incorrect timestamps. Taking medical data (Bose et al., 2013) as an example, an X-ray machine has many asynchronous modules, each of which has a local clock and a local buffer. There can be a discrepancy between when a value is actually updated and when it is recorded since the value is first queued in the buffer before it is recorded. Moreover, 36.96% of appointments have the overlapping issue (*i.e.,* the next appointment in a particular room appears to have started before the current one has ended) (Martin et al., 2019), indicating incorrect timestamps. As remarked earlier by Royal Mail (Royal Mail, 2018), customer data changes frequently. To prevent the data from being outdated, the sales department may contact the customers and get regular updates on some *critical* information (*e.g.,* email and phone). Due to the impatience of customers and high cost (*e.g.,* man power) of contact, the rest (*e.g.,* jobs, marital status) may not be frequently updated and may gradually become obsolete, no longer having a reliable timestamp. Moreover, data values are often copied or imported from other sources (Dong et al., 2009, 2010), and even in the same database, values may come from multiple relations (*e.g.,* by join), where

no uniform scheme of timestamps is granted, *e.g.,* some values are more frequently timestamped than the others (Kurniati et al., 2019). This calls for an attribute-level time-stamping scheme (*i.e.,* each value has a timestamp). It subsumes the tuple-level scheme (*i.e.,* all values in the same tuple have the same timestamp) as a special case.

No matter how desirable, the percentage of reliable timestamps is far lower than expected. How can we determine the temporal orders on the attribute values, *i.e.,* for tuples t_1 and t_2, whether the value in the A-attribute of t_1 is more current than the value in the A-attribute of t_2, denoted by $t_2 \prec_A t_1$, in the absence of complete timestamps?

Example 3.1: Consider customer records t_1-t_6 shown in Figure 3.1, which have been identified to refer to the same person Mary. Each tuple t_i has attributes FN (first name), LN (last name), sex, address, marital status, job, kids and SZ (shoe size). Some attribute values of these tuples have become stale since Mary's data changes over the years, *e.g.,* her job, address and last name have changed 4 times, five times and twice, respectively. Only some attribute values might be associated with reliable timestamps, *e.g.,* the timestamp of $t_5[\mathsf{job}]$ and $t_6[\mathsf{job}]$ are 2016 and 2019 (not shown), respectively, indicating that at that time, the values are up-to-date. In the absence of complete timestamps, it is hard to know whether $t_2 \prec_{\mathsf{LN}} t_6$, *i.e.,* whether the value of $t_2[\mathsf{LN}]$ is more up-to-date than $t_6[\mathsf{LN}]$? Moreover, what Mary's current job title is, *e.g.,* whether $t_i \prec_{\mathsf{job}} t_6$ for $i \in [1,4]$? □

	FN	LN	sex	address	status	job	kids	SZ
t_1, e_1:	Mary	Goldsmith	F	19 xin st	single	n/a	-	5
t_2, e_1:	Mary	Taylor	F	6 gold plaza	married	journalist	1	6
t_3, e_1:	Mary	Taylor	F	19 mall st	espoused	assoc editor	2	7
t_4, e_1:	Mary	Taylor	F	7 ave	divorced	chief editor	2	7
t_5, e_1:	Mary	Taylor	F	7 avenue	detached	chief editor	2	7
t_6, e_1:	Mary	Goldsmith	F	6 const. ave	married	producer	3	7

Figure 3.1: Customer records dataset

Contributions & organization. We categorize this chapter as follows. *(1) Temporal orders* (Section 3.2). We define the notion of temporal orders $t_1 \prec_A t_2$ and $t_1 \preceq_A t_2$ on attributes, and formulate the problem for determining temporal orders. We also review the CCs of (Fan et al., 2012), for deducing temporal orders by our

framework. We show that CCs can specify interesting temporal properties, *e.g.,* the monotonicity, comonotonicity and transitivity.

(2) GATE (Section 3.3). We introduce the framework, GATE. It iteratively invokes a creator to rank the temporal orders on attribute values, followed by the critic that validates the ranking of the creator and deduces more ranked pairs via discovered CCs. The critic also produces augmented training data for the creator to improve its ranking in the next round. This process proceeds for the creator and critic to mutually enhance each other, until the temporal order cannot be further improved.

(3) Creator (Section 3.4). We propose a deep learning model underlying the creator of GATE, to learn temporal orders on attribute values. It departs from previous ranking models in that it learns the temporal orders based on contexts (*i.e.,* attribute correlations) and calculates the confidence of the ranking, by employing chronological embeddings and adaptive pairwise ranking strategies.

(4) Critic (Section 3.5). The critic complements GATE with discovered rules (CCs). We show how it justifies the ranked pairs learned by the creator, and (incrementally) deduces latent temporal orders with the chase (Sadri and Ullman, 1980) using CCs. The chase has the Church-Rosser property (cf. (Abiteboul et al., 1995)), *i.e.,* it guarantees to converge at the same result no matter in what orders the CCs are applied. Moreover, the critic augments the training data for the creator to improve its model in the next round.

(5) Experimental study (Section 3.6). Using real-life and synthetic data, we find that (a) the F-measure of GATE on dataset Career (Leone, 2022) is 0.866, versus 0.35 and 0.36 by rule-based UncertainRule (Li et al., 2018) and Improve3C (Ding et al., 2018) (resp. 0.54 and 0.53 by ML-based $\mathsf{RANK}_{\mathsf{Bert}}$ (Nogueira and Cho, 2019) and $\mathsf{Ditto}_{\mathsf{Rank}}$ (Li et al., 2020b)). (b) On average, GATE is 43.8% and 7.8% more accurate than the critic and the creator, respectively, verifying the effectiveness of combining deep learning and logic rules. (c) GATE is feasible in practice; it only takes 7 rounds to terminate on a real-life dataset of 1,983,698 tuples, with a single machine.

3.2 Determining Temporal Orders

In this section, we first provide the notations for this chapter and then formulate temporal orders and the problem for determining temporal orders (Section 3.2.1). We then review CCs of (Fan et al., 2012) and show that such rules are able to express monotonicity, comonotonicity and transitivity (Section 3.2.2).

Symbols	Notations
$R = (A_1, \ldots, A_n)$	relation schema
(D_e, T_e)	entity instance of schema R pertaining to e
temporal order $t_1 \preceq_A t_2$	t_2 is at least as current as t_1 in attribute A
$\mathsf{conf}(t_1 \preceq_A t_2)$	the confidence of $t_1 \preceq_A t_2$
$(D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$	temporal instance of R
$\varphi = X \rightarrow p_0$	currency constraint
h	valuation of φ in a temporal instance
Γ	the ground truth
D_{aug}	the augmented training data
GATE	Get the lATEst

Table 3.1: Notations

3.2.1 The Problem

Temporal orders and temporal instances. An *entity instance* is (D_e, T_e), where (a) D_e is a normal instance of schema R such that all tuples t_1 and t_2 in D_e refer to the same real-life entity e and hence, $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$; and (b) T_e is a partial function that associates a timestamp $T_e(t[A])$ with the A-attribute of a tuple t in D_e. We refer to (D_e, T_e) as an *entity instance pertaining to e*.

Here the timestamp indicates that at the time $T_e(t[A])$, the A-attribute value of tuple t is correct and up-to-date; it does not necessarily refer to the time when $t[A]$ was created or last updated. If $T_e(t[A])$ is undefined, a reliable timestamp is not available for $t[A]$.

Intuitively, an entity instance extends a normal instance with available timestamps. Its tuples may be extracted from a variety of data sources, and are identified to refer to the same entity e via entity resolution. In the same tuple t, $t[A]$ and $t[B]$ may bear different timestamps (or even no timestamp) for different A and B. Note that we do not assume a timestamp for the entire tuple, since we often find only parts of a tuple to be correct and up-to-date.

Temporal orders. A *temporal order* on attribute A of D_e is a partial order $\preceq_A$ such that for all tuples t_1 and $t_2 \in D_e$, $t_2 \preceq_A t_1$ if the value in $t_1[A]$ is at least as current as $t_2[A]$. We also use a strict partial order $t_2 \prec_A t_1$ if $t_1[A]$ is more current than $t_2[A]$. To simplify the discussion we focus on $\preceq_A$ in the sequel; $\prec_A$ is handled analogously.

In particular, if $T_e(t_1[A])$ and $T_e(t_2[A])$ are both defined and if $T_e(t_2[A]) \leq T_e(t_1[A])$, *i.e.,* when timestamp $T_e(t_1[A])$ is no earlier than $T_e(t_2[A])$, then $t_2 \preceq_A t_1$, *i.e.,* $t_1[A]$ is confirmed at a later timestamp and is thus considered at least as current as $t_2[A]$.

Temporal order $\preceq_A$ is represented as a set of tuple pairs such that $(t_2[\mathsf{tid}], t_1[\mathsf{tid}]) \in \preceq_A$ iff $t_2 \preceq_A t_1$. We write $(t_2[\mathsf{tid}], t_1[\mathsf{tid}])$ as (t_2, t_1) if it is clear in the context. Note that the same value may bear different timeliness in different tuples, *e.g.,* Mary's marital status changed from married (t_2) to divorced (t_4) to married (t_6). While $t_2[\mathsf{status}] = t_6[\mathsf{status}]$, $t_2 \prec_{\mathsf{status}} t_6$. Here $t_2 \prec_{\mathsf{status}} t_6$ ranks the timeliness of the status-attributes of tuples t_2 and t_6, not values (married vs. married) detached from the tuples.

We say that a temporal order $\preceq_A^1$ *extends* $\preceq_A^2$, written as $\preceq_A^2 \subseteq \preceq_A^1$, if for all tuples t_1, t_2 in D_e, if $t_2 \preceq_A^2 t_1$ is defined, then so is $t_2 \preceq_A^1 t_1$. That is, $\preceq_A^1$ includes all tuple pairs in $\preceq_A^2$ and possibly more.

Temporal instances. A *temporal instance* D_t *of* R is given as $(D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$, where each $\preceq_{A_i}$ is a temporal order on A_i ($i \in [1,n]$), $D = \bigcup_{i\in[k]} D_{e_i}$, $T = \bigcup_{i\in[k]} T_{e_i}$, and for all $i \in [1,k]$, (D_{e_i}, T_{e_i}) is an entity instance of R. Here tuples t_1 and t_2 in D are *compatible* under $\preceq_A$ if they pertain to the same entity, *i.e.,* $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$.

Intuitively, D_t is a collection of entity instances, such that each (D_{e_i}, T_{e_i}) pertains to the same entity e_i. We do not rank the currency of tuples if they refer to different entities. A temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ is said to *extend* another temporal instance $D_t' = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T)$ if for all $i \in [1,n]$, $\preceq_{A_i}$ extends $\preceq'_{A_i}$.

Problem statement. We study the problem for *determining the temporal orders* of a temporal instance, stated as follows.

- *Input*: A temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$.
- *Output*: An extended temporal instance $D_t' = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T)$ such that for all $i \in [1,n]$, (a) $\preceq'_{A_i}$ extends $\preceq_{A_i}$ and (b) $\preceq'_{A_i}$ is a total order on all compatible t_1 and t_2 with $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$.

Intuitively, our goal is to extend $\preceq_{A_i}$ such that for all tuples t_1 and t_2 in D, if $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$, we can decide which of $t_1[A_i]$ and $t_2[A_i]$ is more up-to-date. As a consequence, we can deduce the latest value for all attributes. Note that we define a total order on *each* attribute, not a global order on tuples. The total orders on different attributes can be different. This said, we can use the temporal orders learned on one attribute to help the deduction on other attributes, via correlation expressed as currency constraints.

3.2.2 Currency Constraints

We next review the class of currency constraints proposed in (Fan et al., 2012). The predicates over a relation schema R are defined as:

$$p \ ::= \ t[A] \oplus c \ \mid \ t_1[A] \oplus t_2[A] \ \mid \ t_1 \preceq_A t_2,$$

where t, t_1, t_2 are tuple variables denoting tuples of R, A is an attribute of R, c is a constant, $\oplus$ is an operator from $\{=, \neq, >, \geq, <, \leq\}$; $t[A] \oplus c$ and $t_1[A] \oplus t_2[A]$ are defined on attribute values, while $t_1 \preceq_A t_2$ compares the timeliness of $t_1[A]$ and $t_2[A]$. In particular, $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$ says that t_1 and t_2 refer to the same entity.

A *currency constraint* (CC) over schema R is defined as follows:

$$\varphi = X \rightarrow p_0,$$

where X is a conjunction of predicates over R with tuple variables $t_1, \ldots, t_m$, and p_0 has the form $t_u \preceq_{A_i} t_v$ for $u, v \in [1, m]$. We refer to X as the *precondition* of φ, and p_0 as the *consequence* of φ.

As defined in (Fan et al., 2012), a CC can be equivalently expressed as a universal first-order logic sentence of the following form:

$$\varphi = \forall t_1, \ldots, t_m \big(\bigwedge_{j \in [1,m]} (t_1[\mathsf{EID}] = t_j[\mathsf{EID}]) \wedge X \rightarrow t_u \preceq_A t_v \big).$$

Semantics. Currency constraints are defined over temporal instances $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ of R. A *valuation* of tuple variables of a CC φ in $\mathcal{D}_t$, or simply *a valuation of* φ, is a mapping h that instantiates variables $t_1, \ldots, t_m$ with tuples in D that refer to the same real-world entity, as required by $t_1[\mathsf{EID}] = t_j[\mathsf{EID}]$ for $j \in [1, m]$.

A valuation h *satisfies* a predicate p over R, written as $h \models p$, if the following is satisfied: (1) if p is $t[A] \oplus c$ or $t_1[A] \oplus t_2[A]$, then it is interpreted as in tuple relational calculus following the standard semantics of first-order logic (Abiteboul et al., 1995); and (2) if p is $t_1 \preceq_A t_2$, then $t_2[A]$ is at least as current as $t_1[A]$ *i.e.,* (t_1, t_2) is in $\preceq_A$.

For a conjunction X of predicates, we write $h \models X$ if $h \models p$ for *all* p in X. A temporal instance D_t *satisfies* CC φ, denoted by $D_t \models \varphi$, if for *all* valuations h of $X \rightarrow p_0$ in D_t, if $h \models X$ then $h \models p_0$.

Properties. Currency constraints are able to specify interesting temporal properties. Below we exemplify some properties.

Monotonicity. A temporal order $\preceq_A$ over relation schema R is *monotonic* if for any tuples t_1 and t_2 of R that refer to the same entity, if $t_1[A] \leq t_2[A]$ then $t_1 \preceq_A t_2$. For

instance, consider the SZ (shoe size) attribute of the customer relation of Example 3.1. Then $\preceq_{\mathsf{SZ}}$ is monotonic. It can be expressed as the following CC:

$$\varphi_1 = t_1[\mathsf{SZ}] \leq t_2[\mathsf{SZ}] \rightarrow t_1 \preceq_{\mathsf{SZ}} t_2.$$

As another example, marital status only changes from single to married, not the other way around (Canada, 2022). This is expressed as CC:

$$\varphi_2 = t_1[\mathsf{status}] = \text{“single”} \wedge t_2[\mathsf{status}] = \text{“married”} \rightarrow t_1 \preceq_{\mathsf{status}} t_2.$$

With slight abuse of terminologies by considering timestamps as an “attribute” associated with attribute A, we have:

$$\varphi_3 = T_e(t_1[A]) \leq T_e(t_2[A]) \rightarrow t_1 \preceq_A t_2,$$

since $t_2[A]$ is confirmed at a later timestamp.

Comonotonicity. For attributes A and B of R, we say $\preceq_A$ and $\preceq_B$ are *comonotonic* in a temporal instance D_t of R if for all tuples t_1 and t_2 in D_t that refer to the same entity, $t_1 \preceq_A t_2$ if and only if $t_1 \preceq_B t_2$.

Intuitively, $\preceq_A$ and $\preceq_B$ are comonotonic if the two are correlated such that when one is updated, the other will also change. For instance, $\preceq_{\mathsf{status}}$ and $\preceq_{\mathsf{address}}$ are often comonotonic: when the marital status of a person changes from single to married, this person may move to a larger house. This can be expressed as a CC:

$$\varphi_4 = t_1 \preceq_{\mathsf{status}} t_2 \rightarrow t_1 \preceq_{\mathsf{address}} t_2.$$

Along the same lines, we could also have a CC:

$$\varphi_5 = t_1 \preceq_A t_2 \rightarrow t_1 \preceq_{\mathsf{ts}} t_2.$$

Transitivity. Transitivity can also be expressed for any attribute A:

$$\varphi_5 = t_1 \preceq_A t_2 \wedge t_2 \preceq_A t_3 \rightarrow t_1 \preceq_A t_3.$$

Correlating different attributes. One can correlate multiple different attributes to capture implicit temporal ordering. For example, marital status may change from “married” to “divorced” and further from “divorced” to “married” again. To deduce an order on $t[\mathsf{status}]$, we can use the number of kids as an additional condition:

$$\begin{aligned}\varphi_6 = {} & t_1[\mathsf{status}] = \text{“married”} \wedge t_2[\mathsf{status}] = \text{“divorced”} \wedge \\ & t_1[\mathsf{kids}] < t_2[\mathsf{kids}] \rightarrow t_1 \preceq_{\mathsf{status}} t_2.\end{aligned}$$

Deduction. Making use of CCs, we can deduce certain temporal orders, *e.g.,* from φ_1-φ_6 and Figure 3.1, we can deduce the following:

- $t_1 \preceq_{\mathsf{status}} t_2$ by φ_2, $t_2 \preceq_{\mathsf{status}} t_4$ by φ_6 and $t_1 \preceq_{\mathsf{status}} t_4$ by φ_5;
- $t_1 \preceq_{\mathsf{address}} t_2$ by φ_4; hence $t_2[\mathsf{address}]$ is more current for Mary;
- $t_1 \preceq_{\mathsf{SZ}} t_2 \preceq_{\mathsf{SZ}} t_3$ by φ_1; hence Mary's current shoe size is 7.

However, we cannot determine whether $t_2 \preceq_{\mathsf{LN}} t_6$ or $t_6 \preceq_{\mathsf{LN}} t_2$ by deduction with the currency constraints φ_1-φ_6.

Discovery of currency constraints. As noted in (Fan et al., 2012), CCs can be considered as a special case of denial constraints (DCs) (Arenas et al., 1999) extended with temporal orders $\preceq_A$. Several methods have been developed for discovering DCs automatically, *e.g.,* FastDC (Chu et al., 2013), Hydra (Bleifuß et al., 2017), DCFinder (Pena et al., 2019) and ADCMiner (Livshits et al., 2020). We can readily extend these algorithms for discovering CCs. Note that the discovery algorithm is executed once for each relation schema R by sampling its temporal instances. The set Σ of discovered CCs is then applied to different temporal instances. In other words, GATE does not have to discover CCs for each input temporal instance.

3.3 GATE: A Creator-Critic System

In this section, we propose a creator-critic framework for determining temporal orders, and develop system GATE to implement it. A unique feature of GATE is its combined use of deep learning and logic deduction. Below we start with the architecture of GATE, and then present its overall workflow, with termination guarantee.

Architecture. The ultimate goal of GATE is to obtain a total order $\preceq_A$ for each attribute A. As shown in Figure 3.2, GATE first discovers a set Σ of CCs on D_t *offline* for performing logic deduction. Then it takes a temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ as input, and learns and deduces more temporally ranked pairs for D_t *online*. For simplicity, we assume *w.l.o.g.* that D consists of a single entity instance D_e, *i.e.,* all tuples in D pertain to the same entity e and thus their attributes can be pairwisely compared; the methods of this paper can be readily extended to D_t with multiple entity instances.

More specifically, the learning and deducing process in GATE *iteratively* executes two phases, namely, creator and critic, as follows.

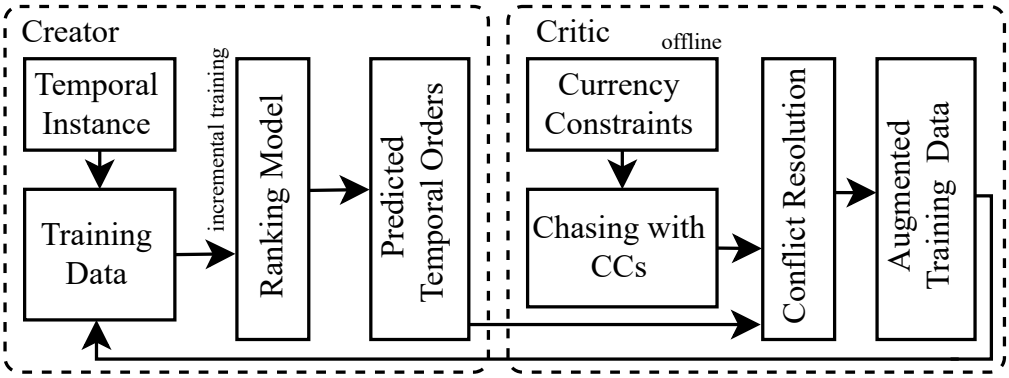


Figure 3.2: Architecture of GATE

(1) Creator (Sections 3.4). In this phase, GATE (incrementally) trains a ranking model $\mathcal{M}_{\mathsf{rank}}$ via deep learning. By taking D_t and the augmented training data D_{aug} (see its definition shortly) from the critic as input, it predicts new orders for extending each $\preceq_A$ of D_t. Our model has three features: (a) For each tuple t in D, the embedding for $t[A]$ is created using both $t[A]$ and other correlated values in t, so that $t[A]$ can be ranked comprehensively. (b) The attribute embeddings (Devlin et al., 2019b) are arranged to preserve chronological orders. (c) We adopt an *attribute-centric adaptive pairwise ranking* strategy in $\mathcal{M}_{\mathsf{rank}}$, so that the ranking result can be justified semantically. Moreover, the temporal instance D_t will also be extended based on D_{aug}.

Given an attribute A, we associate each $(t_1, t_2) \in \preceq_A$ with a *confidence*, denoted by $\mathsf{conf}(t_1 \preceq_A t_2)$, indicating how likely $t_1 \preceq_A t_2$ holds. If $t_2[A]$ has a later timestamp than $t_1[A]$, then $\mathsf{conf}(t_1 \preceq_A t_2)$ is 1. If $t_1 \preceq_A t_2$ is predicted by $\mathcal{M}_{\mathsf{rank}}$, its confidence is from 0 to 1. We only consider (t_1, t_2) predicted by $\mathcal{M}_{\mathsf{rank}}$ with $\mathsf{conf}(t_1 \preceq_A t_2) \geq \delta$, where δ is a predefined threshold, as *candidate pairs* to be extended to $\preceq_A$. We denote the set of all candidate pairs of $\preceq_A$ by $\preceq_A^{\mathcal{M}}$.

The input and output of the creator are as follows:

- *Input*: A temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ and the augmented training data D_{aug} from the critic.
- *Output*: An extended temporal instance $D_t' = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T)$ based on D_{aug} and the predicted orders $(\preceq_{A_1}^{\mathcal{M}}, \ldots, \preceq_{A_n}^{\mathcal{M}})$.

(2) Critic (Sections 3.5). In this phase, the critic of GATE justifies and deduces more temporally ranked pairs, by applying CCs in Σ via the *chase* (Sadri and Ullman, 1980). Denote the result of chasing by $\preceq_A^{\Sigma}$. Depending on the validity of $\preceq_A^{\Sigma}$, we construct an

augmented training data, denoted by D_{aug}, containing the ranked pairs justified (resp. conflicts caught by CCs); D_{aug} will be fed back to the creator, for the next round of model learning, so that the creator can learn from more unseen data and get higher accuracy iteratively. Specifically, D_{aug} is a temporal instance extended with a validity flag $\mathsf{f}_{\mathsf{valid}}$, *i.e.,* $D_{\mathsf{aug}} = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}})$: (a) if $\mathsf{f}_{\mathsf{valid}}$ is true, $\preceq^{\Sigma}_{A}$ is valid and all temporal order deduced by the chase will be added to D_{aug}; and (b) otherwise, the chasing is invalid, *i.e.,* both $(t_1, t_2) \in \preceq_A$ and $(t_2, t_1) \in \preceq_A$ are deduced, and either $t_1 \prec_A t_2$ and $t_2 \prec_A t_1$. In this case, these two *conflicting* orders will be added to D_{aug}, and the creator will be asked to resolve this conflict, by revising its model accordingly.

Formally, the input and output of the critic are as follows:

- *Input*: A temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$, the predicted temporal orders $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n})$ and a set Σ of CCs.
- *Output*: Augmented data $D_{\mathsf{aug}} = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}})$.

The novelty of the critic consists of (a) the deduction using the chase, and (b) an efficient algorithm implementing the chase.

Workflow. As shown in Figure 3.3, GATE takes a temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ and a set Σ of CCs discovered offline as input, and it outputs an extended temporal instance $D'_t = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T)$ with a total order $\preceq'_A$ defined for each attribute A.

GATE first initializes the augmented training data $D_{\mathsf{aug}} = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}} = \mathsf{true})$ (Line 1, details omitted) for the first round of GATE, by deducing its initial temporal orders $\preceq'_A$ via CCs in Σ whose preconditions do not involve timeliness comparison, *e.g.,* we can create temporal orders for those tuples with available timestamps using φ_3. The initialization can be done efficiently by (Fan et al., 2021).

Then GATE iteratively executes the creator and the critic in *rounds*. In the i-th round, (a) the creator extends the temporal instance D_t (Line 4, see Section 3.4) based on the augmented training data D_{aug} returned by the critic in the $(i-1)$-th round, by possibly revising its model to resolve the conflicts. Then the creator *incrementally* trains $\mathcal{M}_{\mathsf{rank}}$ (Line 5, see Section 3.4) and predicts new temporal orders $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n})$ (Line 6), which are candidate pairs (t_1, t_2) with confidences at least δ; and (b) the critic deduces more ranked pairs $(\preceq^{\Sigma}_{A_1}, \ldots, \preceq^{\Sigma}_{A_n})$ by chasing with CCs in Σ (Line 8). Based on the result of chasing, the critic constructs augmented training data D_{aug} via procedure ConstructAugmented (Line 9, see Section 3.5); this D_{aug} is fed back to the creator for the next round.

Input: A temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$, and a set Σ of CCs.
Output: An extended temporal instance $D_t' = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T)$ such that for all $i \in [1,n]$, (a) $\preceq'_{A_i}$ extends $\preceq_{A_i}$ and (b) $\preceq_{A_i}$ is a total order.

1. $D_{\mathsf{aug}} := \mathsf{Initialize}(D_t, \mathsf{f}_{\mathsf{valid}} = \mathsf{true}, \Sigma)$; Initialize the ranking model $\mathcal{M}_{\mathsf{rank}}$;
2. **while** true **do**
3. /* The Creator of GATE */
4. $D_t := \mathsf{Extend}(D_t, D_{\mathsf{aug}})$;
5. Train $\mathcal{M}_{\mathsf{rank}}$ incrementally based on D_t;
6. $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n}) :=$ the predicted temporal orders by $\mathcal{M}_{\mathsf{rank}}$;
7. /* The Critic of GATE */
8. $(\preceq^{\Sigma}_{A_1}, \ldots, \preceq^{\Sigma}_{A_n}) := \mathsf{Chase}(D_t, \Sigma)$;
9. $D_{\mathsf{aug}} := \mathsf{ConstructAugmented}(D, \preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n}, \preceq^{\Sigma}_{A_1}, \ldots, \preceq^{\Sigma}_{A_n}, T)$;
10. **if** D_t no longer changes **then**
11. **break**;
12. $D_t = \mathsf{Extend}(D_t, \mathcal{M}_{\mathsf{rank}})$;
13. **return** D_t;

Figure 3.3: Workflow of GATE

Finally, when D_t no longer changes (Line 10-11), the iteration ends. If D_t is still not a temporal instance with total orders defined on all attributes, we extend it using procedure Extend (Line 12), such that for each pair (t_1, t_2), one of (t_1, t_2) and (t_2, t_1) learned with a higher confidence is in $\preceq_A$, until each $\preceq_A$ becomes a total order.

Termination. We prove that eventually, GATE will terminate (Line 10), *i.e.,* with more iterations, D_t is gradually extended with more orders and finally, becomes stable and does not change.

Theorem 1: GATE *is guaranteed to terminate.* □

Proof of Theorem 1. We prove Theorem 1 by showing that each temporal order $\preceq_A$ will be stable after certain rounds, since (a) once a temporal order $t_1 \preceq_A t_2$ is stable, *i.e.,* if (t_1, t_2) is added to $\preceq_A$ of D_t via procedure Extend (Line 5), it will not be removed from $\preceq_A$ in the subsequent rounds; and (b) the number of stable pairs in $\preceq_A$ is strictly increasing, which is upper-bounded by $O(|D|(|D|-1))$. CCs is a special case of entity enhancing rules (REEs) (Fan et al., 2020), extended with temporal orders $\preceq_A$. We prove Church-Rosser for CCs using a similar argument for REEs in

(Fan et al., 2020), by showing (a) the length of any chasing sequence is bounded and (b) all chasing sequences converge at the same results. □

Example 3.2: Continuing with Example 3.1, assume that $\Sigma = \{\varphi_1\text{-}\varphi_6\}$ and D_t has empty temporal orders. We first initialize the training data D_{aug} by applying CCs in Σ that do not compare timeliness in their preconditions, *e.g.,* we can deduce $t_5 \preceq_{\mathsf{job}} t_6$ by φ_3. Suppose that after training $\mathcal{M}_{\mathsf{rank}}$ based on the initial D_{aug}, no tuple pair predicted by $\mathcal{M}_{\mathsf{rank}}$ is at least δ-confident in the first round. The creator outputs empty $\preceq_A^{\mathcal{M}}$ for each A due to the lack of information. Then by applying the CCs in Σ, the critic can deduce new temporal orders $\preceq_A^{\Sigma}$, *e.g.,* $t_1 \preceq_{\mathsf{address}} t_2$ by φ_4 and $t_1 \preceq_{\mathsf{status}} t_4$ by φ_5. Both $\preceq_A^{\mathcal{M}}$ and $\preceq_A^{\Sigma}$ will be used to construct the augmented training data D_{aug}, based on which the creator extends D_t and incrementally trains $\mathcal{M}_{\mathsf{rank}}$ in the second round. This time the creator might be able to predict confident temporal orders, *e.g.,* $t_1 \preceq_{\mathsf{LN}} t_2$, based on the augmented training data D_{aug}. The iteration continues until D_t does not change anymore. Each temporal order $\preceq_A$ in D_t will be extended to a total order by $\mathcal{M}_{\mathsf{rank}}$ if it is still not total. □

Remark. We adopt a confidence threshold δ to ensure the reliability of ML predictions, so that only reliable orders are considered. When confident orders cannot be decided for the lack of initial information, we may opt to invite user inspection to ensure the correctness of a few initial ranked pairs, from which more reliable orders can be iteratively deduced/learned. The parameter δ plays an important role in model $\mathcal{M}_{\mathsf{rank}}$, *e.g.,* when testing $\mathcal{M}_{\mathsf{rank}}$ on procedural billing data (with 82.28% *real* missing timestamps (Kurniati et al., 2019)), we find that 22.6% pairs are identified as confident when $\delta = 0.55$. The percentage decreases to 16.7% when $\delta = 0.6$. We will test the impact of δ in Section 3.6. The range of the parameter δ is between 0 and 1.

3.4 Creator

In this section, we develop the creator of GATE. Given a temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ and the augmented training data D_{aug} (from the critic), the creator (a) trains a ranking model $\mathcal{M}_{\mathsf{rank}}$ to predict new orders and (b) extends each $\preceq_A$ of D_t based on D_{aug}.

Review on ranking models. *Learning to Rank* (Liu, 2009) aims to learn a ranking model so that objects can be ranked based on their degrees of relevance, preference, or importance (in our setting, timeliness).

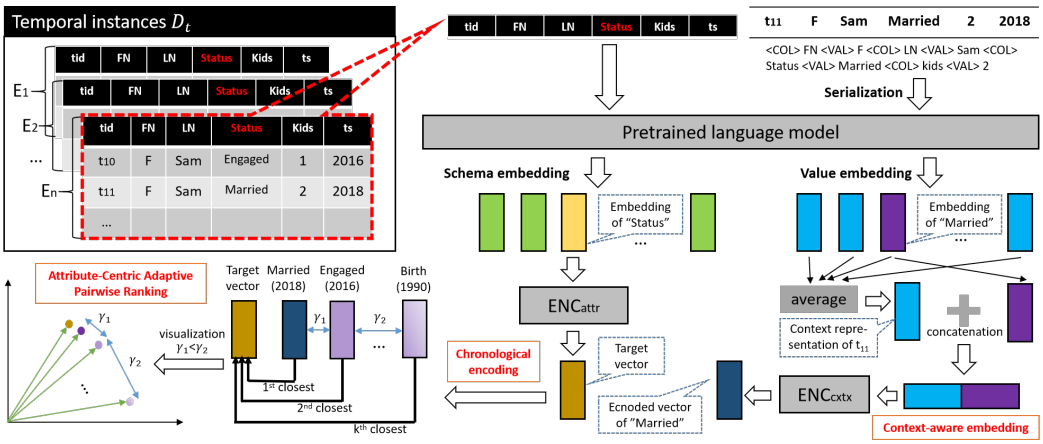


Figure 3.4: The Network Architecture of Creator

We adopt pairwise ranking since (a) its semantics is consistent with temporal orders, which is a set of *tuple pairs* on which partial orders are defined (Section 3.2); and (b) by transitivity of temporal orders, pairwise ranking helps us get a total order for all attributes.

Challenges. One naturally wants to adopt an existing model. However, it is hard to directly apply one, since the unique features of temporal orders are not well considered, resulting in poor performance.

(1) Attribute correlation. Due to the correlated nature of temporal order, we often need to reference other attributes to determine the orders of a given attribute. Moreover, since a value may change back and forth (*e.g.,* the marital status changes from "married" to "divorced", and back from "divorced" to "married"), it is hard to determine the up-to-date value based on a single attribute only.

(2) Limitation of embedding models. To determine the timeliness, care should be taken for lexically different but semantically similar values (*e.g.,* "baby" vs. "birth" and "dead" vs. "expired" for attribute status). Although existing embedding models (*e.g.,* ELMo (Peters et al., 2018) or Bert (Devlin et al., 2019b)) are widely adopted, they cannot be directly used here since they are not trained to organize data chronologically.

(3) Adaptive margin. Existing ranking strategies do not consider real-life characteristics of timeliness, *e.g.,* the timespan for a person's status to move from "birth" to "engaged" is typically longer than from "engaged" to "married" (Figure 3.4). Instead of ranking status with a *fixed* margin as most existing strategies did, we need a new methodology to embed values using *adaptive* margins, to conform to their real-life behaviors and justify the semantic of ranking.

Model overview. We propose a ranking model to tackle the above challenges, whose novelty includes (a) a context-aware scheme that embeds each value along with other correlated values, (b) an encoding mechanism to re-organize the embeddings in a chronological manner, and (c) an attribute-centric adaptive ranking strategy.

As shown in Figure 3.4, our ranking model $\mathcal{M}_{\mathsf{rank}}$ takes the current D_t as input, and outputs new ranked pairs, where each (t_1, t_2) is associated with a *confidence*, indicating how likely $t_1 \preceq_A t_2$ holds.

Our ranking model consists of three stages as follows: context-aware embedding, chronological encoding and order prediction.

(1) $\mathcal{M}_{\mathsf{rank}}$ first builds a context-aware embedding for each attribute value using pre-trained language models (ELMo (Peters et al., 2018) or Bert (Devlin et al., 2019b)), in a way that information of correlated values is also embedded.

(2) Based on the embeddings, $\mathcal{M}_{\mathsf{rank}}$ encodes a target vector ϕ_A for attribute A, via *non-linear transformation* (see below). Similarly, a value vector $\phi_{t[A]}$ is encoded for each A-attribute value of tuple t, so that (a) if $t[A]$ is more current, $\phi_{t[A]}$ is closer to ϕ_A and (b) the gap between $\phi_{t[A]}$ and ϕ_A is trained adaptively, to reflect real semantics.

(3) Finally, given $\phi_{t_1[A]}$ and $\phi_{t_2[A]}$ of t_1 and t_2, $\mathcal{M}_{\mathsf{rank}}$ predicts the order, *i.e.,* whether $t_1 \preceq_A t_2$ holds with high enough confidence.

We next briefly elaborate the context-aware embedding and the chronological encoding scheme with the adaptive margin.

Context-aware embedding. To reference correlated values, we treat tuples as sequences and adopt the idea of serialization (Li et al., 2020b) (so that tuples can be meaningfully ingested by models) to embed values.

Following (Li et al., 2020b), given a tuple t in D_t, we serialize its values:

$$\mathsf{serialize}(t) = \langle \mathsf{COL} \rangle A_1 \langle \mathsf{VAL} \rangle t[A_1] \ldots \langle \mathsf{COL} \rangle A_n \langle \mathsf{VAL} \rangle t[A_n],$$

where $\langle \mathsf{COL} \rangle$ and $\langle \mathsf{VAL} \rangle$ are special tokens, denoting the start of attribute and value, respectively (see Figure 3.4). This serialization is fed as input to a pre-trained language model $\mathsf{emb}(\cdot)$ to compute a d-dimensional embedding for each A-attribute value, denoted by $\mathsf{emb}(t[A]) \in \mathbf{R}^d$. Besides, we average out the embedding vectors for all $t[A]$ to get a context representation of tuple t, *i.e.,* $\mathsf{emb}(t) = \frac{1}{n}\sum_{i=1}^{n} \mathsf{emb}(t[A_i])$. Finally for each A-attribute value of t, we get the context-aware embedding of $t[A]$, denoted by $E_{t[A]} \in \mathbf{R}^{2d}$:

$$E_{t[A]} = [\mathsf{emb}(t[A]); \mathsf{emb}(t)],$$

where $[;]$ denotes vector concatenation. In this way, $E_{t[A]}$ embeds not only the A-attribute value, but also the contextual information from other attribute values, to allow comprehensive ranking. Similarly, a schema embedding for each attribute A is computed: $E_A = \mathsf{emb}(A)$, by feeding the attribute name, *e.g.,* status, to the model.

Chronological encoding with adaptive margins. While pre-trained embeddings are widely adopted to capture semantics, they are not trained to organize temporal orders. Thus we propose chronological encoding to re-organize the embeddings to preserve timeliness. The idea is to use schema embedding as the target and make the embedding of a more current value closer to the target; moreover, instead of ranking in fixed margins, embeddings are ordered adaptively.

Specifically, given the embedding of the A-attribute of t, *i.e.,* $E_{t[A]}$, we encode it using a context encoder $\mathsf{ENC}_{\mathsf{cxtx}}(\cdot)$ as follows:

$$\phi_{t[A]} = \mathsf{ENC}_{\mathsf{cxtx}}(E_{t[A]}) = \sigma(\mathsf{W}_2 * \sigma(\mathsf{W}_1 * E_{t[A]})),$$

where W_1 and W_2 are learnable parameters of the encoder, and σ is the non-linear sigmoid activation function given by $\sigma(x) = \frac{1}{1+e^{-x}}$.

Similarly, the target vector for attribute A is encoded as $\phi_A = \mathsf{ENC}_{\mathsf{attr}}(E_A)$, where $\mathsf{ENC}_{\mathsf{attr}}(\cdot)$ denotes the schema encoder.

To train the encoders with ordered embeddings and adaptive margins, we adopt an attribute-centric adaptive margin-based loss. Given $\preceq_A$ in D_t, the loss on A is formulated as follows:

$$\mathsf{loss}(A) = \sum_{(t_1,t_2)\in\preceq_A} \left\{ \mathsf{max}\{-\mathsf{tanh}(\langle \phi_{t_2[A]}, \phi_A \rangle) + (\gamma_{t_1,t_2}) + \mathsf{tanh}(\langle \phi_{t_1[A]}, \phi_A \rangle), 0\} \right\},$$

where $\langle \cdot, \cdot \rangle$ is the inner product and γ_{t_1,t_2} is the adaptive margin between the two tuples; we set γ_{t_1,t_2} to be $1 - \cos(v_{t_1[A]}, v_{t_2[A]})$ in practice, where $v_{t_1[A]}$ and $v_{t_2[A]}$ are the Word2Vec (Church, 2017) embeddings which characterize the co-occurrence of $t_1[A]$ and $t_2[A]$.

Intuitively, by minimizing the loss, for each training instance $t_1 \preceq_A t_2$, (a) we make the encoded vector of the more current value $t_2[A]$ closer to target ϕ_A (the first term) and (b) we ensure an adaptive margin γ_{t_1,t_2} between the two tuples (the second term). In other words, the attribute values in the encoded space are not only arranged chronologically by their distances to ϕ_A, from which temporal orders can be easily derived, their margins are also adaptively determined, to reflect the semantic of timeliness ranking.

Model training and instance extension. In each round, the creator receives augmented training data $D_{\mathsf{aug}} = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}})$, based on which it (a) incrementally trains $\mathcal{M}_{\mathsf{rank}}$ via back-propagation and (b) extends the temporal instance D_t with D_{aug}.

In the first round, D_{aug} is initialized to be the temporal orders constructed by applying those CCs in Σ without timeliness comparison in their preconditions (Fan et al., 2021). In the following rounds, D_{aug} is constructed by the critic, based on the result of chasing with CCs.

Specifically, depending on flag $\mathsf{f}_{\mathsf{valid}}$ in D_{aug}, we have two cases:

(1) If $\mathsf{f}_{\mathsf{valid}}$ is true, the result of chasing is valid and $\mathcal{M}_{\mathsf{rank}}$ is incrementally trained on the orders in D_{aug} (see below). Moreover, the temporal instance $D_t = (D, \preceq_{A_1}, \ldots, \preceq_{A_n}, T)$ is extended with the ranked pairs in D_{aug}, *i.e.,* for each (t_1, t_2) in $\preceq'_A$ of D_{aug}, (t_1, t_2) is added to $\preceq_A$. In this case, we say that (t_1, t_2) becomes *stable*. Once a temporal order becomes stable, it will not be removed from D_t.

(2) If $\mathsf{f}_{\mathsf{valid}}$ is false, the result of chasing is invalid, *i.e.,* there is an attribute A such that conflicting orders (t_1, t_2) and (t_2, t_1) are both in $\preceq'_A$ of D_{aug} *i.e.,* $\{(t_1, t_2), (t_2, t_1)\} \subseteq \preceq'_A$, and either $t_1 \prec_A t_2$ or $t_2 \prec_A t_1$. In this case, we decide that either (t_1, t_2) or (t_2, t_1) is added to $\preceq_A$ using $\mathcal{M}_{\mathsf{rank}}$, with a higher confidence (and possibly user inspection). Assume *w.l.o.g.* that (t_1, t_2) is added to $\preceq_A$ (*i.e.,* it becomes stable). Then, the creator fine-tunes $\mathcal{M}_{\mathsf{rank}}$ so that t_1 and t_2 are better separated in the encoded space.

Incremental training. Incremental training of $\mathcal{M}_{\mathsf{rank}}$ might lead to the catastrophic forgetting issue (Kirkpatrick et al., 2016), *i.e.,* the model might forget some temporal orders learned in prior rounds. To overcome this, we adopt a simple strategy to retain prior knowledge: In each round, $\mathcal{M}_{\mathsf{rank}}$ first makes inference to previously learned orders, and then extracts those that make wrong predictions. Then, we add these orders to the augmented data D_{aug} and then train the model on D_{aug}. In this way, the model is able to learn from previous training instances and alleviate the impact of the catastrophic forgetting issue.

Monotonicity. One can verify that the number of stable temporal orders in D_t is (strictly) monotonically increasing when more rounds GATE are performed. The termination of GATE (Theorem 1) partly depends on this monotonicity.

Confidence. Given $\preceq_A$ and a tuple pair (t_1, t_2), $\mathcal{M}_{\mathsf{rank}}$ predicts $(t_1, t_2) \in \preceq_A$ if $\mathsf{tanh}(\langle \phi_{t_2[A]}, \phi_A \rangle) > \mathsf{tanh}(\langle \phi_{t_1[A]}, \phi_A \rangle)$; its confidence indicates how likely $t_1 \preceq_A t_2$

holds. We compute it to be

$$\mathsf{conf}(t_1 \preceq_A t_2) = \sigma(\tanh(\langle \phi_{t_2[A]}, \phi_A \rangle) - \tanh(\langle \phi_{t_1[A]}, \phi_A \rangle))$$

and set $\mathsf{conf}(t_2 \preceq_A t_1)$ to 0. Thus, given a confidence threshold $\delta > 0$, we will not predict both $t_1 \preceq_A t_2$ and $t_2 \preceq_A t_1$ as confident orders.

Intuitively, we use $\tanh(\langle \phi_{t[A]}, \phi_A \rangle)$ to measure the "distance" between $\phi_{t[A]}$ and ϕ_A, where the closer one is more current. The distance gap between $\phi_{t_1[A]}$ and $\phi_{t_2[A]}$ quantifies the confidence: the larger the gap, the larger the confidence, which ranges from 0 to 1.

Example 3.3: Consider the example in Figure 3.4, where we focus on attribute status. After creating the context-aware embedding based on a pre-trained model, it chronologically encodes a target vector ϕ_{status} and value vectors for all values, so that they are arranged by their distances to ϕ_{status}. To illustrate, we also label the *unknown* timestamp of each value vector in the figure (*e.g.,* $T_e(\text{Married}) = 2018$). Since ϕ_{Married} is the closest to ϕ_{status}, it is predicted to be the latest status value and new ranked pairs are constructed accordingly for $\preceq_{\mathsf{status}}$, as augmented training data in the next round. □

Remark. Our creator learns temporal orders by utilizing context-aware embedding, chronological encoding and attribute-centric adaptive ranking. However, it does not explicitly take into account of some temporal properties, such as the transitivity. This motivates us to use critic to deduce and justify the temporal orders based on the semantics of the data, as will be presented in the next section.

3.5 Critic

In this section, we develop the critic under GATE for justifying and deducing temporal orders. Taking a temporal instance D_t, the temporal orders $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n})$ predicted by the creator and a set Σ of mined CCs as input, the critic (a) deduces more ranked pairs $(\preceq^{\Sigma}_{A_1}, \ldots, \preceq^{\Sigma}_{A_n})$ by applying CCs via the *chase*, and (b) constructs the augmented training data D_{aug} and feeds D_{aug} back to the creator.

3.5.1 Chasing with CCs

We extend the classic chase in (Fan et al., 2020) for CCs under GATE. As opposed to the classical chase, we apply a CC only if its precondition is satisfied by a collection

"ground truth". In the following, we first specify fixes and ground truth. We then present the chase for CCs, with the Church-Rosser property.

Fixes. We extend temporal orders in D_t by applying CCs in Σ to deduce *fixes*, which are modeled as sets of ranked pairs, denoted by $\overline{U} = (\overline{U}_{A_1}, \ldots, \overline{U}_{A_n})$, where each ranked pair (t_1, t_2) in $\overline{U}_A$ is referred to as a *fix* and it means that $t_1 \preceq_A t_2$ or $t_1 \prec_A t_2$ is deduced. We only apply a CC if its precondition is satisfied by a collection Γ of "ground truth". Intuitively, the fixes are logical consequences of Σ and Γ, *i.e.,* as long as the CCs in Σ and Γ are correct, so are the fixes.

Validity. We say $\overline{U}$ is *valid* if it has no conflicting fixes, *i.e.,* there exist no attribute A and tuples t_1, t_2 such that $(t_1, t_2) \in \overline{U}_A$ and $(t_2, t_1) \in \overline{U}_A$ at the same time, and either $t_1 \prec_A t_2$ or $t_2 \prec_A t_1$.

Ground truth. To justify the correctness of fixes, we maintain and employ a collection $\Gamma = (\Gamma_{A_1}, \ldots, \Gamma_{A_n})$ of validated data, *enclosed* in $\overline{U}$. In our setting, block Γ is initialized by applying CCs in Σ whose preconditions do not involve timeliness comparison (*e.g.,* initial temporal orders with partial timestamps via φ_3), and is iteratively expanded with the temporal orders learned by the creator with confidence above threshold δ or deduced by the chase in the critic.

The chase. Given a temporal instance D_t, the chase deduces fixes by chasing D_t with CCs in Σ and ground truth in Γ. It uses sets $\preceq^{\Sigma} = (\preceq^{\Sigma}_{A_1}, \ldots, \preceq^{\Sigma}_{A_n})$ to keep track of the affected fixes in $\overline{U}$. Specifically, the i-th chase step of D_t by Σ at $(\overline{U}_i, \preceq^{\Sigma}_i)$ is:

$$(\overline{U}_i, \preceq^{\Sigma}_i) \Rightarrow_{(\varphi,h)} (\overline{U}_{i+1}, \preceq^{\Sigma}_{i+1}),$$

where $\varphi : X \rightarrow p_0$ is a CC in Σ, h is a valuation of φ in $\mathcal{D}_t$, and the application of (φ, h) should satisfy the following conditions:

(1) All predicates $p \in X$ are *validated*, *i.e.,* if p is $t[A] \oplus c$ or $t_1[A] \oplus t_2[A]$, then $h \models p$; and if p is $t_1 \preceq_A t_2$, then (t_1, t_2) is in $\overline{U}_A$.

(2) The consequence $p_0 : t_1 \preceq_A t_2$ extends $\overline{U}_i$ to $\overline{U}_{i+1}$, such that (t_1, t_2) is added to $\overline{U}_A$ of $\overline{U}_i$; similarly, p_0 extends $\preceq^{\Sigma}_i$ to $\preceq^{\Sigma}_{i+1}$.

Chasing. Starting from a set $\overline{U}_0$ of fixes, initialized to be Γ, and an empty $\preceq^{\Sigma}_0$, a chasing sequence ξ of D_t by (Σ, Γ) is

$$(\overline{U}_0, \preceq^{\Sigma}_0), \ldots, (\overline{U}_k, \preceq^{\Sigma}_k),$$

where $(\overline{U}_i, \preceq^{\Sigma}_i) \Rightarrow_{(\varphi,h)} (\overline{U}_{i+1}, \preceq^{\Sigma}_{i+1})$ is a valid chase step, *i.e.,* a valuation h of φ extends $(\overline{U}_i, \preceq^{\Sigma}_i)$ to $(\overline{U}_{i+1}, \preceq^{\Sigma}_{i+1})$ where $\overline{U}_{i+1}$ is valid.

The chasing sequence is *terminal* if there exist no CC φ in Σ and valuation h of φ such that (φ, h) leads to another valid chase step.

A chase sequence ξ terminates in one of the following cases:

(1) No more CCs in Σ can be applied. If so, we say that ξ is valid, with $(\overline{U}_k, \preceq_k^\Sigma)$ as its result.

(2) Either $\overline{U}_0$ is invalid or there exist $\varphi, h, \overline{U}_{k+1}$ and $\preceq_{k+1}^\Sigma$ such that $(\overline{U}_k, \preceq_k^\Sigma) \Rightarrow_{(\varphi,h)} (\overline{U}_{k+1}, \preceq_{k+1}^\Sigma)$ but $\overline{U}_{k+1}$ is invalid. Such ξ is invalid, and the result of the chase is $\perp$ (undefined).

Intuitively, the chase helps us deduce more ranked pairs when it terminates with enriched $(\overline{U}_k, \preceq_k^\Sigma)$; moreover, it justifies and explains the learned order if no invalid chase step is taken. When its result is $\perp$, it detects invalid ranked pairs of the learner.

Example 3.4: Consider D_t in Figure 3.1. Assume that $\Sigma = \{\varphi_1\text{-}\varphi_6\}$ and $\preceq^\Sigma = (\preceq_{A_1}^\Sigma, \ldots, \preceq_{A_n}^\Sigma)$, where each $\preceq_{A_i}^\Sigma$ is empty. We initialize $\overline{U}_0$ and Γ by applying CCs without timeliness comparison, as we did in Example 3.2, *e.g.,* since $t_1[\mathsf{status}]$ (resp. $t_2[\mathsf{status}]$) is "single" (resp. "married") in Figure 3.1, $t_1 \preceq_{\mathsf{status}} t_2$ is initialized in Γ by applying φ_2.

We have the following chase steps of D_t by (Σ, Γ):

(1) By applying (φ_4, h_4), where φ_4 is $t_1 \preceq_{\mathsf{status}} t_2 \rightarrow t_1 \preceq_{\mathsf{address}} t_2$ and h_4 maps the variables of φ_4 to tuples t_1 and t_2 in D_t, we deduce $t_1 \preceq_{\mathsf{address}} t_2$ by the chase step $(\overline{U}_0, \preceq_0^\Sigma) \Rightarrow_{(\varphi_4,h_4)} (\overline{U}_1, \preceq_1^\Sigma)$, *i.e.,* $\overline{U}_1$ extends $\overline{U}_0$ by adding (t_1, t_2) to $\overline{U}_{\mathsf{address}}$; similar to $\preceq_1^\Sigma$.

(2) The chase proceeds to deduce $t_1 \preceq_{\mathsf{status}} t_4$ by applying φ_5.

This chasing sequence is valid since each chase step in the sequence is valid and no more CCs in Σ can be applied anymore. □

Church-Rosser property. Following (Abiteboul et al., 1995), we say that chasing with CCs is *Church-Rosser* if for any temporal instance D_t, any set Σ of CCs, any collection Γ of ground truth, all chasing sequences of D_t by (Σ, Γ) are terminal and converge at the same result. Below we show that chasing with CCs is Church-Rosser .

3.5.2 Discovery of CCs

As noted in (Fan et al., 2012), CCs can be considered as a special case of denial constraints (DCs) introduced in (Arenas et al., 1999), extended with temporal orders

$\preceq_A$. Several algorithms have been developed for discovering DCs, *e.g.,* FastDC (Chu et al., 2013), Hydra (Bleifuß et al., 2017), DCFinder (Pena et al., 2019) and ADCMiner (Livshits et al., 2020).

We extend DCFinder (Pena et al., 2019) for discovering CCs as follows. (1) We support timeliness comparisons, *i.e.,* $t_1 \preceq_A t_2$, by adding them to the evidence set used in (Pena et al., 2019) or transforming categorical values to numerical ones using a mapping function $f_{\mathsf{map}}(\cdot)$, such that timeliness is preserved, *e.g.,* given $t_1 \preceq_A t_2$, we ensure $f_{\mathsf{map}}(t_1[A]) \leq f_{\mathsf{map}}(t_2[A])$. (2) We restrict our evidence set on tuples with the same EIDs, instead of using a global evidence set. This accelerates CCs discovery, since CCs are only defined on tuple variables that refer to the same entity. (3) We revise DCFinder by always selecting $t_1[\mathsf{EID}] = t_2[\mathsf{EID}]$ as the first predicate. Note that CCs could also be added manually, *e.g.,* to ensure the transitivity of the data.

3.5.3 Monotonicity

We next show that the number of stable temporal orders in D_t is (strictly) monotonically increasing when more rounds GATE are performed (Lemma 2). The termination of GATE (Theorem 1) partly depends on this lemma. Lemma 2 directly follows from the way we expand stable temporal orders.

Lemma 2: *For temporal instances D_t^{j-1} and D_t^j in the j-th round of* GATE *before and after the extension, respectively,* i.e., $D_t^j = \mathsf{Extend}(D_t^{j-1}, D_{\mathsf{aug}})$*, the following holds:*

$$(a)\ \forall i \in [1,n], \preceq_{A_i}^{j-1} \subseteq \preceq_{A_i}^{j} \text{ and } (b)\ \exists i^* \in [1,n], \preceq_{A_{i^*}}^{j-1} \subset \preceq_{A_{i^*}}^{j}$$

where $\preceq_{A_i}^{j-1}$ and $\preceq_{A_i}^{j}$ are the orders in D_t^{j-1} and D_t^j, respectively. □

Proof. By the way we expand stable temporal orders, there are two cases: (1) If $\mathsf{f}_{\mathsf{valid}}$ is true, the result of chasing is valid. Then at least one non-empty $\preceq'_A$ of D_{aug} will be used to extend $\preceq_A$, whose size strictly increases. (2) If $\mathsf{f}_{\mathsf{valid}}$ is false, the result of chasing is invalid, *i.e.,* there is an attribute A such that $t_1[A] \neq t_2[A]$ but conflicting orders (t_1,t_2) and (t_2,t_1) are both in $\preceq'_A$ of D_{aug}. Either (t_1,t_2) or (t_2,t_1) is added to $\preceq_A$, resulting an increased size of $\preceq_A$. Once a temporal order $t_1 \preceq_A t_2$ is added to $\preceq_A$, it will not be removed. □

3.5.4 Structures and strategies

To support lazy evocation of valuations, we employ the following:

(1) A set RHS of triples, where each triple (t_1, t_2, A) in RHS indicates that the order between $t_1[A]$ and $t_2[A]$ is not settled, *i.e.,* neither $t_1 \preceq_A t_2$ nor $t_2 \preceq_A t_1$ is stable in D_t yet. Note that we only apply a CC if its consequence has a corresponding triple in RHS. By ensuring this, the length of a chasing sequence is bounded by $O(|\mathsf{RHS}|)$.

(2) A set $\mathcal{H}$ of *partial valuations*. Specifically, a valuation h of CC $\varphi : X \to p_0$ is said to be *partial* if some predicates in X are validated, while others are not. If all predicates in X are validated, h becomes *complete* and we can deduce temporal orders by applying (φ, h). The set $\mathcal{H}$ is maintained to avoid repeated predicate validation.

(3) An index I for the partial valuations in $\mathcal{H}$, *i.e.,* for each temporal order o, $I[o]$ maintains the partial valuations h of $\varphi : X \to p_0$ in $\mathcal{H}$ such that there exists a predicate p in X and $o = h(p)$. By maintaining I, every time a temporal order o is deduced, we can efficiently locate the valuations affected by o in $I[o]$, without scanning the entire Σ. Besides, an inverted index is also built for each h so that once h is removed from $\mathcal{H}$, I can be updated efficiently.

(4) A set M of triples, where each triple (t_1, t_2, φ) indicates that the ranked pair (t_1, t_2) has been used to evoke the valuations for φ before and thus those valuations will not be evoked again.

Moreover, we adopt the following strategies.

(5) Lazy evocation, where valuations of CCs in Σ are constructed if they are evoked by some newly deduced orders, instead of being generated all at the beginning of the chase. Specifically, when a new temporal order o is deduced, we check each $\varphi : X \to p_0$ in Σ and evoke a new partial valuation h of φ if o corresponds to a predicate in X (*i.e., o* is validated in h) and h has not be evoked before (checked by M). Such h can only be evoked if the temporal order it deduces, *i.e.,* $h(p_0)$, is not deduced by other valuations before (checked by RHS).

Algorithm. Putting these together, we present Chase in Figure 3.5 (a complete version of Figure 3.6, with data structures incorporated). It returns new orders $\preceq^{\Sigma}$ if the chase is valid, and $\perp$ otherwise.

Chase starts with the initialization (Line 1). (a) Ground truth Γ is initialized with all stable ranked pair in D_t via procedure Initialize (omitted). (b) $\overline{U}$ and $\preceq^{\Sigma}$ are initialized as stated in Section 3.5.1 to be Γ and $\emptyset$, respectively. (c) The set Δ of newly vali-

Input: A temporal instance D_t, the set Σ of CCs, the set $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n})$ predicted by the creator, the global structures $\mathsf{GS} = (\mathsf{RHS}, \mathcal{H}, I, M)$.

Output: The result of chasing, $\preceq^{\Sigma}$.

1. $\Gamma := \mathsf{Initialize}(D_t)$; $\overline{U} := \Gamma$; $\preceq^{\Sigma} := \emptyset$; $\Delta = \mathsf{NewStable}(\Gamma)$;
2. **while** Δ is not empty **do**
3. $\quad \Delta^{\mathsf{new}} = \emptyset$;
4. $\quad$ **for each** $o \in \Delta$ **do**
5. $\quad\quad \mathcal{H} := \mathcal{H} \cup \mathsf{CCEvoke}(D_t, \Sigma, o, \mathsf{GS})$; Update I and M;
6. $\quad\quad$ **for each** $h \in I[o]$ where h deduces $t_1 \preceq_A t_2$ **do**
7. $\quad\quad\quad$ **if** $(t_1, t_2, A) \notin \mathsf{RHS}$ **then**
8. $\quad\quad\quad\quad$ Remove h from $\mathcal{H}$ and I; **continue** ;
9. $\quad\quad\quad$ Mark o as validated in h;
10. $\quad\quad\quad$ **if** h is complete **then** /* $t_1 \preceq_A t_2$ is a newly deduced order */
11. $\quad\quad\quad\quad$ Remove h from $\mathcal{H}$ and I;
12. $\quad\quad\quad\quad$ $\overline{U}_A := \overline{U}_A \cup (t_1, t_2)$; $\preceq^{\Sigma}_A := \preceq^{\Sigma}_A \cup (t_1, t_2)$;
13. $\quad\quad\quad\quad$ **if** $(t_2, t_1) \in \overline{U}_A$ or $(t_2, t_1) \in \preceq^{\mathcal{M}}_A$ **then** /* conflict */
14. $\quad\quad\quad\quad\quad$ $\preceq^{\Sigma} := \perp$; **return** $\preceq^{\Sigma}$;
15. $\quad\quad\quad\quad$ $\Delta^{\mathsf{new}} := \Delta^{\mathsf{new}} \cup \{t_1 \preceq_A t_2\}$;
16. $\quad$ $\Delta := \Delta^{\mathsf{new}}$;
17. **return** $\preceq^{\Sigma}$;

Figure 3.5: Procedure Chase (with data structures)

dated orders is initialized to be the newly stable orders in Γ via procedure NewStable (omitted), and they are the temporal orders "triggering" the chase.

Then for each order o in Δ, Chase does the following (Line 5-15): (a) Evoke the valuations h based on o via the lazy evocation strategy stated above, by calling CCEvoke (omitted), and add h to $\mathcal{H}$ (Line 5). (b) For each valuation h in $I[o]$, mark o as validated in h (Line 9). If h becomes complete (Line 10), the consequence $t_1 \preceq_A t_2$ of h is deduced, and the ranked pair (t_1, t_2) is added to $\overline{U}_A$ and $\preceq^{\Sigma}_A$ (Line 12). (c) Check conflicts (Line 13-14): if (t_1, t_2) conflicts with (t_2, t_1) that is already in $\overline{U}_A$ or $\preceq^{\mathcal{M}}_A$, the chase terminates with $\preceq^{\Sigma} = \perp$. In this case, the partial valuations and the temporal orders that have been examined and deduced are kept temporally in the global structures so that they can be re-used in the next round of GATE when the

conflicts are resolved (not shown). (d) Maintain the global structures in the three cases below: (i) if new valuation h is evoked by o (Line 5), I is updated accordingly and M is also updated such that h will not be evoked again; (ii) if h becomes useless, *i.e.,* the ranked pair it deduces is no longer in RHS (Line 7-8), h is removed from $\mathcal{H}$ and I; and (iii) if h becomes complete, we deduce new orders, namely $t_1 \preceq_A t_2$, by applying h; then h is removed from $\mathcal{H}$ and I (Line 11). (e) Add the newly deduced order to the set Δ^{new} (Line 15) for iteratively processing, by assigning Δ^{new} to Δ (Line 16).

Finally, the result of chasing, $\preceq^{\Sigma}$, is returned (Line 17).

Complexity. The loop of Chase executes at most $O(|R||D|^2)$ times, since there are at most $|R||D|^2$ temporal orders to be deduced (*i.e.,* the length of any chasing sequence is $O(|R||D|^2)$). For each temporal order o deduced, we evoke CCs based on o and update the data structures in $O(c_{\mathsf{val}}|\Sigma|)$ time, where c_{val} denotes the unit cost of constructing the valuations for fixed o and φ, and there are at most $|\Sigma|$ many CCs. Thus, Chase takes at most $O(c_{\mathsf{val}}|\Sigma||R||D|^2)$ time.

3.5.5 Deduction with the Chase

No matter how desirable, the chase could be expensive if we enumerate valuations of CCs in an exhaustive manner. Below we provide an efficient algorithm to implement the chase.

Challenges. A brute-force implementation of the chase is by enumerating valuations h of each CC φ in Σ. If (φ, h) can be applied, a chase step is performed, until the chasing sequence terminates. This method is, however, costly since valuation enumeration is inherently exponential. To tackle this challenge, we develop an efficient algorithm to implement the chase; the key idea is to only evoke valuations *pertaining* to the affected fixes in the chase *lazily* (see below).

We assume *w.l.o.g.* that for each $\varphi : X \to p_0$ in Σ, X has a predicate p in the form of $t_1 \preceq_A t_2$ (*e.g.,* φ_4). For those CCs that do not compare timeliness in their precondition (*e.g.,* φ_1), we apply them in a pre-processing step, to generate initial temporal orders in D_t.

Lazy evocation. To allow lazy evocation, the valuations of CCs in Σ are generated only when they are evoked by some newly deduced orders, instead of constructing all at the beginning of the chase.

Specifically, when a new temporal order o is deduced, we check each $\varphi : X \to p_0$ in Σ and evoke a new valuation h of φ if (a) o corresponds to a predicate in X (*i.e., o*

is validated in h); in this case, we say that h is a valuation *pertaining* to o since h is "activated" by o, (b) h has not been evoked before and (c) the order that h deduces, *i.e.,* $h(p_0)$, is not deduced by other valuations before. We maintain designated data structures for checking conditions (a), (b) and (c) efficiently.

Algorithm. Putting these together, we present Chase in Figure 3.6. It returns new orders $\preceq^{\Sigma}$ if the chase is valid, and $\perp$ otherwise.

Chase starts with the initialization (Line 1). (a) Ground truth Γ is initialized with all stable ranked pair in D_t via procedure Initialize (omitted). (b) It initializes $\overline{U}$ and $\preceq^{\Sigma}$ as stated in Section 3.5.1 to be Γ and $\emptyset$, respectively. (c) The set Δ of newly validated orders is initialized to be the newly stable orders in Γ via procedure NewStable (omitted); and they are the temporal orders "triggering" the chase. (d) The set $\mathcal{H}$ of evoked valuations is initialized to be empty.

Then for each order o in Δ, Chase does the following (Line 5-15): (a) Evoke the valuations pertaining to o via the lazy evocation strategy stated above, by calling CCEvoke (omitted), and add them to $\mathcal{H}$ (Line 5). (b) For each valuation h pertaining to o that deduces unknown ranked pair (checked in Line 7-8), mark o as validated in h (Line 9). If all predicates in the precondition of h are validated (Line 10), (h,φ) can be applied and the consequence $t_1 \preceq_A t_2$ of h is deduced (Line 11). The ranked pair (t_1,t_2) is added to $\overline{U}_A$ and $\preceq_A^{\Sigma}$ (Line 12). (c) Check conflicts (Line 13-14): if (t_1,t_2) conflicts with (t_2,t_1) that is already in $\overline{U}_A$ or $\preceq_A^{\mathcal{M}}$, the chase terminates with $\preceq^{\Sigma}=\perp$. In this case, the valuations in $\mathcal{H}$ are kept temporally so that they can be re-used in the next round of GATE when the conflicts are resolved (not shown). (d) Add the newly deduced order to the set Δ^{new} (Line 15) for iterative processing, by assigning Δ^{new} to Δ (Line 16).

Finally, the result of chasing, $\preceq^{\Sigma}$, is returned (Line 17).

Example 3.5: Recall that φ_4 is $t_1 \preceq_{\mathsf{status}} t_2 \rightarrow t_1 \preceq_{\mathsf{address}} t_2$ and φ_5 is $t_1 \preceq_{\mathsf{status}} t_2 \wedge t_2 \preceq_{\mathsf{status}} t_3 \rightarrow t_1 \preceq_{\mathsf{status}} t_3$. Let $\Sigma = \{\varphi_4, \varphi_5\}$ and $\Delta = \{t_1 \preceq_{\mathsf{status}} t_2\}$. We process $t_1 \preceq_{\mathsf{status}} t_2$ in Δ as follows. It first evokes valuations h_4 and h_5 which map the variables of φ_4 and φ_5 to tuples t_1 and t_2 in D_t, respectively, with $t_1 \preceq_{\mathsf{status}} t_2$ validated. Since the predicate $t_2 \preceq_{\mathsf{status}} t_3$ in h_5 is not validated, h_5 is kept in $\mathcal{H}$ for later processing. In contrast, all predicates in the precondition of h_4 are validated and φ_4 deduces $t_1 \preceq_{\mathsf{address}} t_2$. Suppose that there is no conflicting order in $\overline{U}_{\mathsf{address}}$ and $\preceq^{\mathcal{M}}_{\mathsf{address}}$. Then $t_1 \preceq_{\mathsf{address}} t_2$ forms a new set Δ and the process continues, until Δ is empty. □

Input: A temporal instance D_t, the set Σ of CCs, the predicted $(\preceq^{\mathcal{M}}_{A_1}, \ldots, \preceq^{\mathcal{M}}_{A_n})$
Output: The result of chasing, $\preceq^{\Sigma}$.

1. $\Gamma :=$ Initialize(D_t); $\overline{U} := \Gamma$; $\preceq^{\Sigma} := \emptyset$; $\Delta =$ NewStable(Γ); $\mathcal{H} := \emptyset$;
2. **while** Δ is not empty **do**
3. $\quad \Delta^{\mathsf{new}} = \emptyset$;
4. $\quad$ **for each** $o \in \Delta$ **do**
5. $\quad\quad \mathcal{H} := \mathcal{H} \cup$ CCEvoke(D_t, Σ, o);
6. $\quad\quad$ **for each** h of $\varphi : X \to t_1 \preceq_A t_2$ s.t. h pertains to o **do**
7. $\quad\quad\quad$ **if** the order between $t_1[A]$ and $t_2[A]$ is already settled **then**
8. $\quad\quad\quad\quad \mathcal{H} := \mathcal{H} \setminus \{h\}$; **continue** ;
9. $\quad\quad\quad$ Mark o as validated in h;
10. $\quad\quad\quad$ **if** all predicates in the precondition of h are validated **then**
11. $\quad\quad\quad\quad \mathcal{H} := \mathcal{H} \setminus \{h\}$; /* $t_1 \preceq_A t_2$ is a newly deduced order */
12. $\quad\quad\quad\quad \overline{U}_A := \overline{U}_A \cup (t_1, t_2)$; $\preceq^{\Sigma}_A := \preceq^{\Sigma}_A \cup (t_1, t_2)$;
13. $\quad\quad\quad\quad$ **if** $(t_2, t_1) \in \overline{U}_A$ or $(t_2, t_1) \in \preceq^{\mathcal{M}}_A$ **then** /* conflict */
14. $\quad\quad\quad\quad\quad \preceq^{\Sigma} := \bot$; **return** $\preceq^{\Sigma}$;
15. $\quad\quad\quad\quad \Delta^{\mathsf{new}} := \Delta^{\mathsf{new}} \cup \{t_1 \preceq_A t_2\}$;
16. $\quad \Delta := \Delta^{\mathsf{new}}$;
17. **return** $\preceq^{\Sigma}$;

Figure 3.6: Procedure Chase

Complexity. The loop of Chase executes at most $O(|R||D|^2)$ times, since there are at most $|R||D|^2$ temporal orders to be deduced. For each temporal order o deduced, we evoke CCs based on o and update data structures in $O(c_{\mathsf{val}}|\Sigma|)$ time, where c_{val} denotes the unit cost of constructing valuations for fixed o and φ, and there are at most $|\Sigma|$ CCs. Thus Chase takes at most $O(c_{\mathsf{val}}|\Sigma||R||D|^2)$ time.

Augmented training data construction. Recall that the result of chasing, denoted by $\preceq^{\Sigma}$, is valid or invalid. Based on $\preceq^{\Sigma}$, we construct the augmented training data D_{aug} as follows.
(1) If $\preceq^{\Sigma}$ is valid, both the temporal orders deduced by the chase and predicted by the creator are used to create $D_{\mathsf{aug}} = (D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}} = \mathsf{true})$ where $\preceq'_{A_i} = \preceq^{\mathcal{M}}_{A_i} \cup \preceq^{\Sigma}_{A_i}$ $(i \in [1, n])$.

(2) If $\preceq^{\Sigma}$ is $\bot$, then there exist conflicting ranked pairs, *i.e.,* both (t_1, t_2) and (t_2, t_1)

are in $\overline{U}_A$ or $\preceq_A^{\mathcal{M}}$ with $t_1 \prec_A t_2$ or $t_2 \prec_A t_1$. In this case, we construct D_{aug} to be $(D, \preceq'_{A_1}, \ldots, \preceq'_{A_n}, T, \mathsf{f}_{\mathsf{valid}} = \mathsf{false})$ where $\preceq_{A_i} = \{(t_1,t_2),(t_2,t_1)\}$ if $A_i = A$ and $\preceq_{A_i} = \emptyset$ otherwise. As shown in Section 3.4, the creator fine-tunes its model by using the deduced ranked pairs or the detected conflicts in D_{aug}.

3.6 Experimental Study

Using real-life and synthetic data, we evaluated (1) the effectiveness and (2) the efficiency of GATE for determining temporal orders. We also (3) conducted a case study to showcase the usefulness of GATE.

Experimental settings. We start with the experimental setting.

Datasets. We used three real-life datasets and one synthetic dataset. (1) Career (Leone, 2022), a benchmark about the careers of football players from FIFA-15 to FIFA-22; it contains 108.5K tuples from 27.2K entities with 20 attributes. We determine the timeliness of potential, position, reputation and league name. (2) NBA (Ding et al., 2018), a dataset that encompasses the careers of basketball players; it contains 10.6K tuples with 12 attributes. We determine the currency of team, pts (points) and weight. (3) COM (Govement, 2022), an open-source dataset about self-employed entrepreneurs in Shenzhen. After removing duplicated tuples and digital attributes (*e.g.,* the organization code), the dataset has 1,983,698 tuples and 8 attributes. We derive the timeliness of entrepreneur names. (4) Person, a synthetic dataset with 12.3K tuples from 1K entities. Just like in Figure 3.1, we adopted 7 attributes and determine the currency of LN, Status, Kids. Here Person is generated by enforcing CCs (*e.g.,* φ_1-φ_6), to simulate real-world scenarios.

All the datasets have ground truth, *i.e.,* all tuples carry timestamps and are grouped by entities. The timestamps of COM are in seconds, while the others are in years; it is reasonable since, *e.g.,* it is uncommon for NBA players to frequently change teams in one year. We randomly selected 5% data with initial timestamps and masked the remaining. This default ratio ts% of initial timestamps is set intentionally small (so that the problem is more challenging); we will test the impact of ts% by varying ts% (Exp-1).

Currency constraints. We extended DCFinder (Pena et al., 2019) to discover CCs as discussed in Section 3.2. Note that CC discovery is conducted once on each dataset offline. Besides, we manually checked and adjusted the CCs discovered to ensure their

correctness. We found 42, 32, 40 and 36 CCs for Career, NBA, COM and Person, respectively.

ML model. To learn the ranking model $\mathcal{M}_{\mathsf{rank}}$, we used Bert (Devlin et al., 2019b) (distilbert) with 768 dimension to initialize the embeddings. We adopted 2 hidden layers in our encoders with sizes 200 and 100, respectively. The margin γ was adaptively computed and the model was trained with 30 epochs using Adam optimizer (Kingma and Ba, 2015). The learning rate is 1e-4. We used 5% data with timestamps as training data.

Baselines. GATE was implemented in Python and we compared it with the following baselines: (1) Creator, a variant of GATE with the creator only, *i.e.,* it predicts temporal orders using $\mathcal{M}_{\mathsf{rank}}$; (2) Critic, a variant of GATE with the critic only, *i.e.,* it deduces temporal orders by chasing with CCs; (3) $\mathsf{Creator}_{\mathsf{itr}}$, a variant of Creator that iteratively updates its training data with predicted but unjustified temporal orders. (4) $\mathsf{Creator}_{\mathsf{NC}}$, $\mathsf{Creator}_{\mathsf{NE}}$, $\mathsf{Creator}_{\mathsf{NA}}$, another three variants of Creator that implement $\mathcal{M}_{\mathsf{rank}}$ without contextual information, without chronological encoding, and using regular cross entropy loss instead of adaptive margin-based loss, respectively; (5) $\mathsf{GATE}_{\mathsf{NC}}$, a variant of GATE that adopts the brute-force method for the chase, by enumerating all valuations exhaustively.

We also tested (6) UncertainRule (Li et al., 2018), which uses uncertain currency rules to evaluate data currency; (7) Improve3C (Ding et al., 2018), a data quality framework that combines completeness, consistency and currency (Fan et al., 2014b); we only compare its accuracy for currency; (8) $\mathsf{RANK}_{\mathsf{Bert}}$ (Nogueira and Cho, 2019), a state-of-the-art ML ranking model based on Bert; and (9) $\mathsf{Ditto}_{\mathsf{Rank}}$, a ranking model that first trains a ditto model (Li et al., 2020b) to conduct binary classification on attribute values (with contextual information) and then sorts all attribute values using ditto as the comparison operator.

Among the baselines, (a) Critic, UncertainRule and Improve3C are rule-based, where rules for the latter two are converted from same CCs mined by DCFinder, (b) Creator, $\mathsf{Creator}_{\mathsf{itr}}$, $\mathsf{Creator}_{\mathsf{NA}}$, $\mathsf{Creator}_{\mathsf{NC}}$, $\mathsf{Creator}_{\mathsf{NE}}$, $\mathsf{RANK}_{\mathsf{Bert}}$ and $\mathsf{Ditto}_{\mathsf{Rank}}$ are ML-based, and (c) $\mathsf{GATE}_{\mathsf{NC}}$ is a hybrid method, which produces same results as GATE. Thus, we compared $\mathsf{GATE}_{\mathsf{NC}}$ mostly for efficiency.

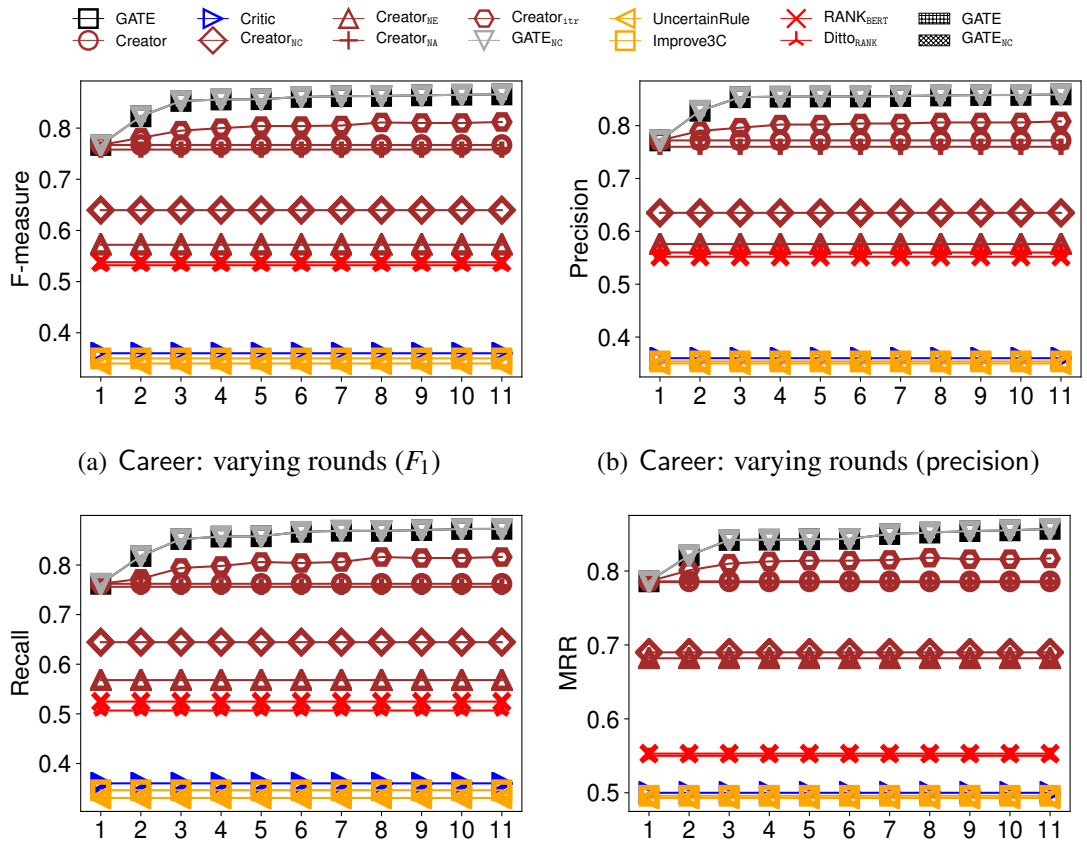
We did the experiments on a single machine powered by 256GB RAM and 32 processors with Intel(R) Xeon(R) Gold 5320 CPU @2.20GHz. We ran each experiment 3 times and report the average.

Experimental results. We next report our findings.

Exp-1: Effectiveness. Since we adopted the pairwise ranking setting to deduce ranked pairs, we evaluated the accuracy of GATE following (Fan et al., 2013b; Ding et al., 2018; Li and Sun, 2018): (1) precision, the ratio of temporal orders determined correctly to all ranked pairs predicated true, (2) recall, the ratio of temporal orders predicted correctly to all true orders, and (3) F-measure $= 2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$. To evaluate the ranking of GATE, for each entity instance pertaining to entity e, we compute $\mathsf{MRR}(e) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{\mathsf{rank}_i}$, the mean reciprocal rank over a set of n ranking results, where n is the number of currency attributes and rank_i denotes the rank of the latest value of the i-th attribute for entity e, and $\mathsf{MAP@K}(e) = \frac{1}{n}\sum_{i=1}^{n} \mathsf{AP@K}(i)$, the mean average precision at K that assesses whether the top-K values predicted are relevant and whether the latest K values are at the top, where $\mathsf{AP@K}(i)$ is the average precision at K of the i-th attribute for entity e. Let D_t be a collection of k entity instances. We report (4) $\mathsf{MRR} = \frac{1}{k}\sum_{i=1}^{k} \mathsf{MRR}(e_i)$ and (5) $\mathsf{MAP@K} = \frac{1}{k}\sum_{i=1}^{k} \mathsf{MAP@K}(e_i)$, with K = 3 by default , which is the first 3 elements of i-th attribute for entity e.

Rounds. We report the performance of GATE from the first round till its termination; at the end of the fixed round, we use current $\mathcal{M}_{\mathsf{rank}}$ in the creator. As shown in Figures 3.7(a)-3.9(a), GATE takes 11, 12, 7 and 9 rounds to terminate on Career, NBA, COM and Person, respectively, *i.e.,* GATE converges quickly. Besides, we find the following.

(1) Although the performance of GATE might fluctuate (*e.g.,* Figure 3.8(d)), which is common in ML models (Devlin et al., 2019b), all metrics increase with more rounds in most cases, *e.g.,* F-measure, MAP and MRR increase from 0.767 to 0.866, 0.786 to 0.857, and 0.752 to 0.809, respectively, after 11 rounds on Career, verifying that GATE is able to deduce the latest values and produce good currency ranking. This is because the creator iteratively accumulates training data from the critic such that the model is better trained with more rounds; meanwhile, with better results predicted by the creator, the critic deduces more orders as augmented training data for the creator in subsequent rounds. Moreover, in Figures 3.7(b) and 3.7(c), precision and recall are 0.859 and 0.873, respectively, indicating that GATE achieves a good balance between the two and is fairly accurate. Note that $\mathsf{Creator}_{\mathsf{itr}}$ suffers from the accuracy fluctuation since its model is affected by noisy (unjustified) temporal orders accumulated over rounds. The performance of other methods does not depend on rounds, as shown in flat lines. Since GATE behaves similarly under all metrics, below we focus on F-measure.



(a) Career: varying rounds (F_1)

(b) Career: varying rounds (precision)

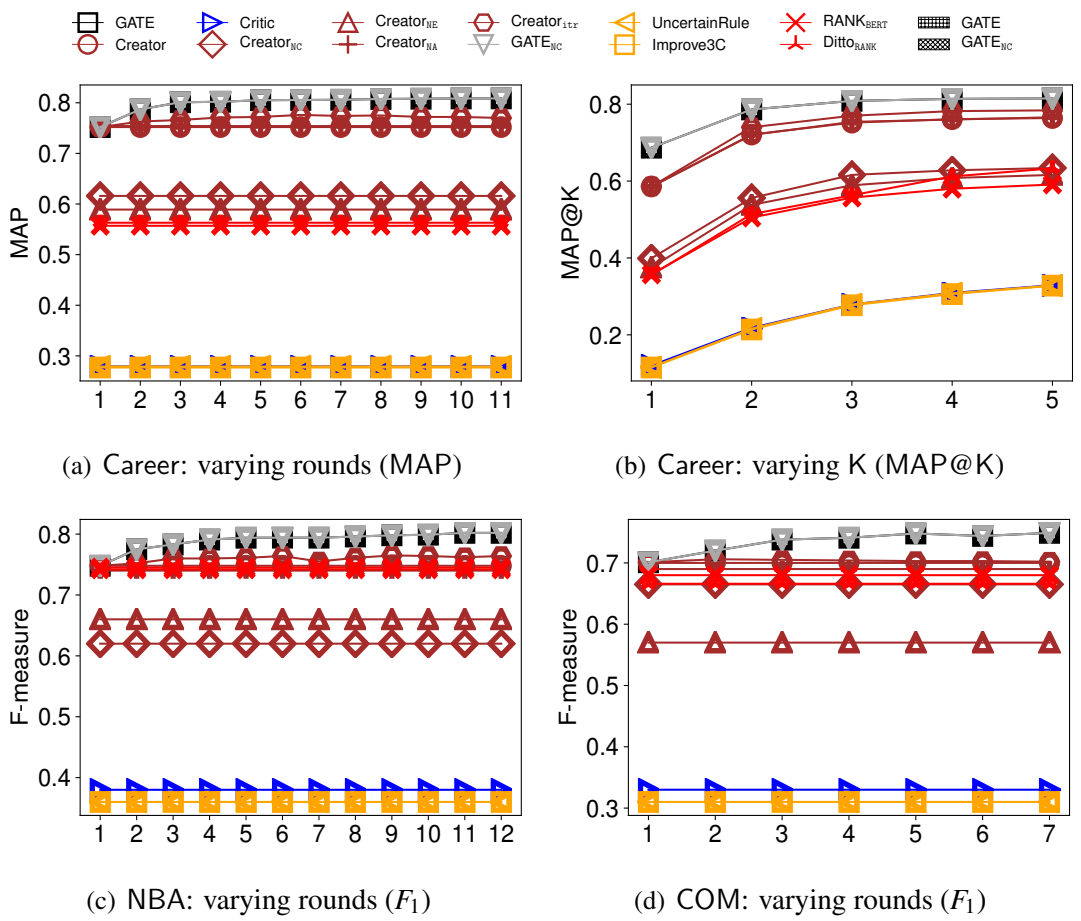
(c) Career: varying rounds (recall)

(d) Career: varying rounds (MRR)

Figure 3.7: F-measure, Precision, Recall and MRR vs varying rounds for GATE, its variants and baseline models

(2) On average GATE outperforms Creator and Critic by 7.8% and 43.8% in F-measure, respectively, up to 11.0% and 50.6%, improving both. The creator and critic benefit each other: (a) Creator produces "hidden" temporal orders for Critic to preform deduction, and (b) Critic deduces and justifies the orders, which are in turn provided as augmented training data to Creator; on average, the critic generates 5733 new training data (tuple pairs) per round on COM, improving F-measure of GATE from 0.701 to 0.748 after 5 rounds.

(3) Creator is more accurate than all its variants, *e.g.,* the average F-measure of Creator is 0.722, as opposed to 0.641, 0.613 and 0.714 by $\text{Creator}_{\text{NC}}$, $\text{Creator}_{\text{NE}}$ and $\text{Creator}_{\text{NA}}$, respectively, on Career. Intuitively, (a) without utilizing the contextual information, $\text{Creator}_{\text{NC}}$ cannot reference correlated attributes; (b) $\text{Creator}_{\text{NE}}$ has low accuracy with existing embedding models, and (c) compared to the regular cross en-



(a) Career: varying rounds (MAP)

(b) Career: varying K (MAP@K)

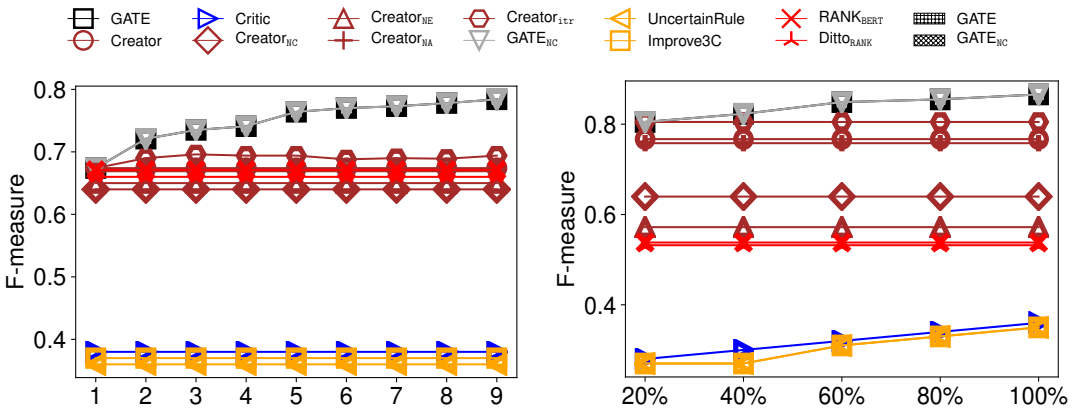
(c) NBA: varying rounds (F_1)

(d) COM: varying rounds (F_1)

Figure 3.8: MAP and F-measure performance evaluation of GATE, its variants and baseline models

tropy loss used in $\text{Creator}_{\text{NA}}$, the adaptive pairwise ranking loss helps by considering the semantics in ranking. Moreover, GATE is 10.1% more accurate than $\text{Creator}_{\text{itr}}$ on average. This verifies the need for justifying the temporal orders learned by Creator.

(4) The accuracy of GATE is higher than UncertainRule, Improve3C, $\text{RANK}_{\text{Bert}}$ and $\text{Ditto}_{\text{Rank}}$, *e.g.,* the average F-measure of GATE is 0.802 as opposed to 0.344, 0.349, 0.659 and 0.651 for the four, respectively. This shows the benefits of combining deep learning and logic rules: (a) compared with rule-based methods, GATE can learn from unseen data and has better generalizability; and (b) compared with ML-based methods, GATE is able to justify the reliability of deduction and produces more training data for the model. We also report the accuracy of GATE from the first round till its termination on Career in Figure 3.10(c). We find: (a) the Accuracy of GATE increases with more rounds, *e.g.,* increases from 0.766 to 0.863 after 11 rounds, which verifies



(a) Person: varying rounds (F_1) (b) Career: varying $|\Sigma|$ (F_1)

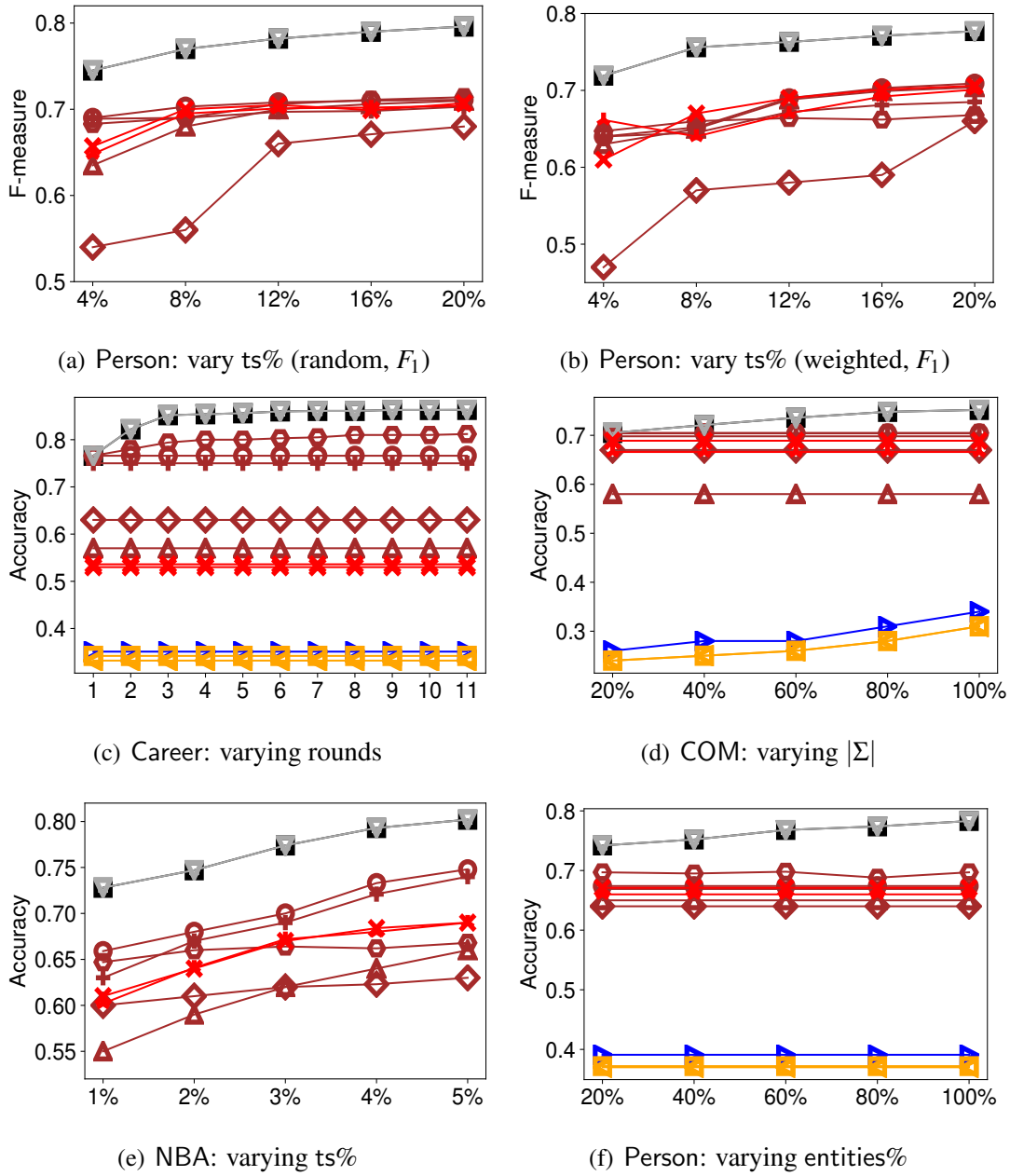
Figure 3.9: F-measure performance evaluation of GATE, its variants and baseline models

that GATE is capable to deduce the latest currency values; (b) the Accuracy of GATE is consistently higher than Creator, Critic and their variants, *e.g.,* it outperforms Creator and Critic by 9.7% and 51.2%, respectively; and (c) GATE is more accurate, in terms of Accuracy, than rule-based methods UncertainRule and Improve3C and ML-based models $\text{Ditto}_{\text{Rank}}$ and $\text{RANK}_{\text{Bert}}$, *e.g.,* the Accuracy of GATE is 22.7% higher than the best baseline. This verifies the benefit of combining deep learning and logical rules.

Varying K. We varied parameter K of MAP@K from 1 to 5 in Figure 3.8(b). GATE consistently achieves the highest MAP@K, *e.g.,* 5% higher than the best baseline on average, up to 10.1%. This verifies that GATE ranks attribute values better than the baselines.

Varying $|\Sigma|$. Varying $|\Sigma|$, we evaluated the impact of the number of CCs in Figure 3.9(b). The accuracy of GATE, UncertainRule and Improve3C improves given more rules. For GATE, its F-measure changes from 0.805 to 0.866 when $|\Sigma|$ varies from 20% to 100%. Indeed, the critic deduces more orders with more CCs for the creator to fine-tune its model, to get a higher accuracy in an earlier stage. We also varied $|\Sigma|$ from 20% to 100% on COM in Figure 3.10(d). As expected, the accuracy of GATE, Critic, UncertainRule and Improve3C increase when $|\Sigma|$ increases, *e.g.,*Accu of GATE increases from 0.705 to 0.752 when $|\Sigma|$ is from 20% to 100%. This show that more temporal orders can be correctly deducted given more rules.

Varying initial ts%. We varied the ratio ts% of initial timestamps (randomly selected) from 4% to 20%. More initial timestamps help since (a) the creator has more training



(a) Person: vary ts% (random, F_1)

(b) Person: vary ts% (weighted, F_1)

(c) Career: varying rounds

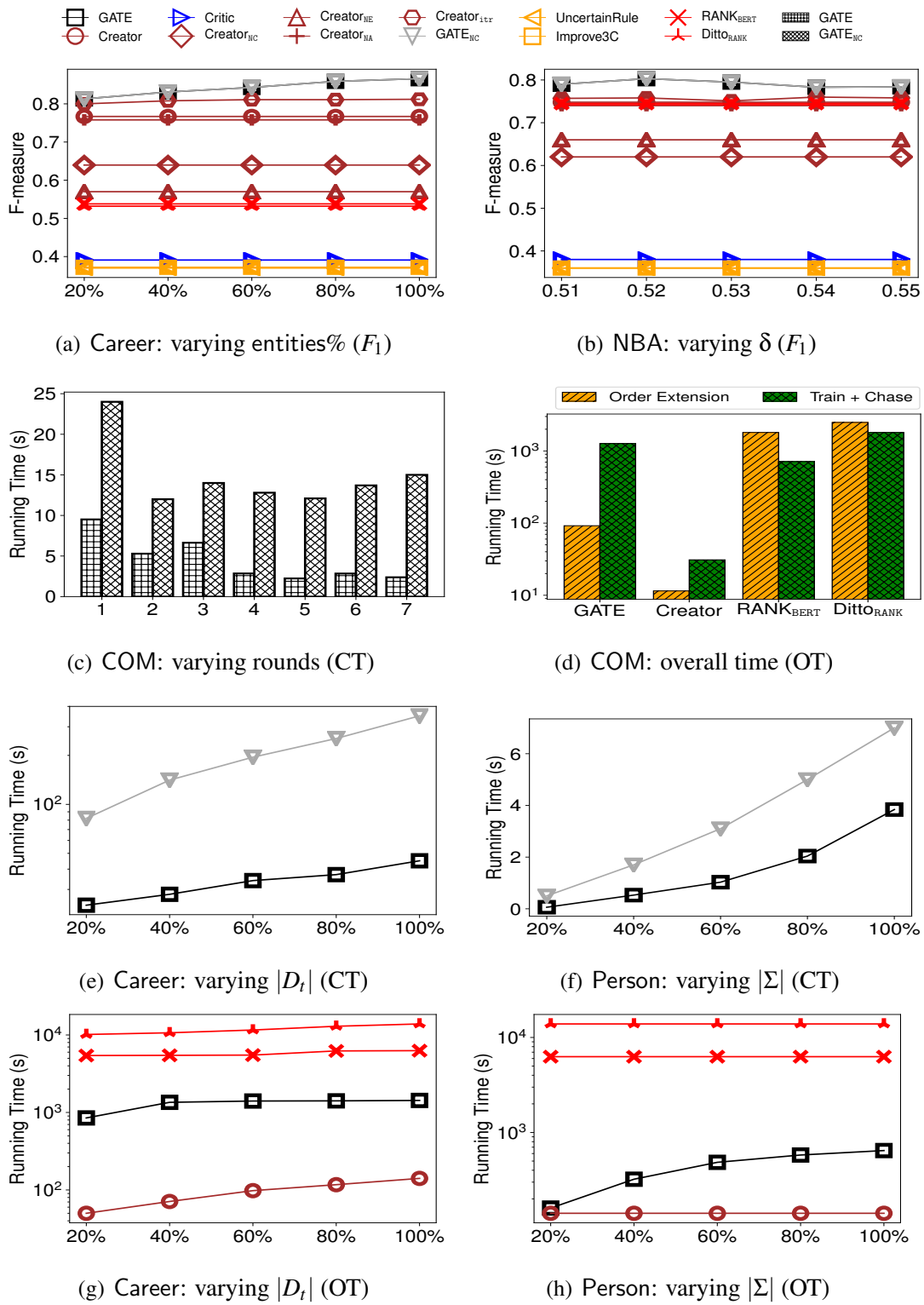
(d) COM: varying $|\Sigma|$

(e) NBA: varying ts%

(f) Person: varying entities%

Figure 3.10: F-measure and Accuracy performance evaluation on different datasets

data and can have better initial performance; and (b) the critic gets temporal orders as ground truth to perform deduction at the beginning of the chase. As shown in Figure 3.10(a), the F-measure of GATE increases (from 0.75 to 0.796 on Person) when ts% varies from 4% to 20%, as expected. We also varied the ratio ts% of initial timestamps from 1% to 5% on NBA in Figure 3.10(e). As shown there, all methods tends to be more accuracy with larger ts%, since more temporal orders can be deduced based on tuples with initial timestamps and ML models are inclined to get more accurate when given more training data.



(a) Career: varying entities% (F_1)

(b) NBA: varying δ (F_1)

(c) COM: varying rounds (CT)

(d) COM: overall time (OT)

(e) Career: varying $|D_t|$ (CT)

(f) Person: varying $|\Sigma|$ (CT)

(g) Career: varying $|D_t|$ (OT)

(h) Person: varying $|\Sigma|$ (OT)

Figure 3.10: F-measure and Running Time performance evaluation of GATE, its variants and baseline models

The discrepancy in the initial values of the F-measure between Figure 3.8(a) and Figures 3.9(a) and 3.9(b) arises from distinct starting points. Specifically, Figure 3.9(a) portrays the result across all rounds, leading to a higher starting point. In contrast, Figure 3.8(a) exclusively illustrates the outcome of the initial round. However, it also improves as there are more rounds.

As remarked earlier, some attribute values may have more reliable timestamps than the others (Kurniati et al., 2019). To study the impact of timestamp distributions, we assigned a weight to each attribute, where a larger weight indicates that the values of this attribute is more likely to be selected with initial timestamps, *e.g.,* on Person, values on Status are more likely to have timestamps than Kids. Consistent with Figure 3.10(a), Figure 3.10(b) shows that the accuracy under weighted sampling is also improved with larger ts%. Note that the accuracy under weighted sampling (Figure 3.10(b)) is slightly lower than random sampling (Figure 3.10(a)), *e.g.,* when ts%=20% on Person, its F-measure is 2.5% lower than random sampling. This is because weighted sampling is impacted by distribution discrepancy between training and testing data, which is a common out-of-distribution issue for ML.

Varying entities%. As reported in Figure 3.11(a), we varied the percentage of entities that are used for the chase from 20% to 100%. As expected, GATE improves its F-measure since with more entities, the critic can deduce more orders via the chase, and the creator can have more augmented data to train the model, achieving higher accuracy. For instance, F-measure of GATE is improved by 5.3% on average. We also evaluated all methods by varying the percentage of entities used for chasing from 20% to 100% in Figure 3.11(a). GATE has higher accuracy with more entities, *e.g.,* its Accuracy changes from 0.742 to 0.783 when entities% is from 20% to 100%, since more training data can be produced by the critic, benefiting the creator.

Varying δ. We next tested the impact of confidence threshold δ. As shown in Figure 3.11(b), (a) although when δ is small, less confident predictions may appear in subsequent deductions, more temporal orders could be used for training; (b) when δ is too large, few ranked pairs learned are retained, and hence less augmented training data is returned. Since the creator receives less data to fine-tune its model, the accuracy may not be improved and it converges slowly. When $\delta = 0.52$, GATE has the highest accuracy on NBA. Thus, we set $\delta = 0.52$ as its default; for other datasets, δ is set similarly.

Exp-2: Efficiency. We tested the efficiency of GATE, $\mathsf{GATE}_{\mathsf{NC}}$, Creator, $\mathsf{RANK}_{\mathsf{Bert}}$ and $\mathsf{Ditto}_{\mathsf{Rank}}$. Denoted by CT (resp. OT) the chase time (resp. the overall time, for training, chasing and order extension). For GATE, OT is accumulated over all rounds; its order extension time is the time for extending stable orders to total orders by $\mathcal{M}_{\mathsf{rank}}$ (Line 12 of Figure 3.3). For other baselines, the order extension time is the inference time of models for generating total orders. We did not report rule-based methods, which are fast, since they do not need to train models; but as shown in Exp-1, they are not accurate.

Chase time (rounds). We report CT of GATE and $\mathsf{GATE}_{\mathsf{NC}}$ in the iterative process. As shown in Figure 3.11(c), GATE is substantially faster than $\mathsf{GATE}_{\mathsf{NC}}$ for all rounds; it is 3.99X faster than $\mathsf{GATE}_{\mathsf{NC}}$ on average, up to 6.30X on COM. The speedup of GATE is due to the lazy evocation strategy we adopted, which accelerates the chase by maintaining designated structures. In contrast, $\mathsf{GATE}_{\mathsf{NC}}$ enumerates valuations and incurs redundant computation. Since GATE and $\mathsf{GATE}_{\mathsf{NC}}$ produces the same results, their total rounds are the same.

Overall time. We next report OT in Figure 3.11(d). Although GATE has multiple rounds, its overall time is comparable to most ML methods, *e.g.,* GATE is 3.15X faster than $\mathsf{Ditto}_{\mathsf{Rank}}$ on average. In particular, the order extension time of GATE is smaller than most baselines except Creator, since GATE has to perform extra checks so that the total order generated is consistent with the known stable orders.

Varying $|D_t|$. We evaluated CT (resp. OT) of GATE and $\mathsf{GATE}_{\mathsf{NC}}$ (resp. ML methods) by varying $|D_t|$ from 20% to 100% in Figure 3.11(e) (resp. 3.11(g)). With larger $|D_t|$, all methods take longer, as expected. Nonetheless, GATE is faster than $\mathsf{GATE}_{\mathsf{NC}}$ when $|D_t|$ gets larger, since GATE maintains structures to avoid recomputation and does deduction pertaining to affected orders, *e.g.,* GATE is 4.80X faster than $\mathsf{GATE}_{\mathsf{NC}}$ when $|D_t|$ is 100%; the result for OT is consistent.

Varying $|\Sigma|$. We varied $|\Sigma|$ from 20% to 100% on Person in Figure 3.11(f) and 3.11(h). As shown there, GATE is 3.86X faster than $\mathsf{GATE}_{\mathsf{NC}}$ on average, up to 8.33X, which again verifies the effectiveness of lazy evocation. OT of GATE increases as $|\Sigma|$ is larger, since more training data (*i.e.,* temporal orders) is deduced by CCs to train the ranking model.

Exp-3: Case study. We use Career to illustrate why GATE works.

(1) Initial prediction. In the first round, GATE trains the creator on data with initial timestamps. Due to the limited (5%) training data, its F-measure is only 0.65 and some ranked pairs are mispredicted. One of confident and correct predictions is $t_1 \preceq_{\mathsf{height}} t_2$, where t_1 and t_2 denote the same player with heights 186cm and 188cm, respectively. While this player moved from team PARMA to SPAL, *i.e.,* $t_1 \preceq_{\mathsf{team}} t_2$, the creator makes a wrong prediction $t_2 \preceq_{\mathsf{team}} t_1$.

(2) Critic *helps* Creator. After the creator stage, the critic uses CCs to correct mispredicted temporal orders. By applying $\varphi_7 : t_a \preceq_{\mathsf{height}} t_b \rightarrow t_a \preceq_{\mathsf{team}} t_b$ to the known $t_1 \preceq_{\mathsf{height}} t_2$, it deduces $t_1 \preceq_{\mathsf{team}} t_2$, correcting the mistake of Creator. Intuitively, φ_7 holds since $\preceq_{\mathsf{height}}$ is monotonic, and $\preceq_{\mathsf{height}}$ and $\preceq_{\mathsf{team}}$ correlate for young players.

Moreover, Critic provides augmented training data to Creator. For instance, if $t_0 \preceq_{\mathsf{league\_name}} t_1$ and $t_1 \preceq_{\mathsf{potential}} t_2$ are in the ground truth, the critic could apply CC $\varphi_8 : t_a \preceq_{\mathsf{league\_name}} t_b \wedge t_b \preceq_{\mathsf{potential}} t_c \wedge t_a[\mathsf{height}] \leq t_c[\mathsf{height}] \rightarrow t_a \preceq_{\mathsf{position}} t_c$, and deduces a new pair $t_0 \preceq_{\mathsf{position}} t_2$ that is unknown before. Here φ_8 is learned from the data; intuitively, if a player moves to a new league (as indicated by monotonic $\preceq_{\mathsf{height}}$) and if his potential changes, his position is likely adjusted, *e.g.,* from LM to CAM. After the first round, the critic creates 100,277 new ranked pairs as augmented training data, and the creator improves its model with the new data.

(3) Creator *helps* Critic. Critic cannot correctly deduce total orders from limited initial 5% timestamps. Nonetheless, with augmented training data provided by Critic, Creator can learn more ranked pairs with high confidence. On average it ranks 2843 tuple pairs with high confidence in the first 5 rounds. These ranked pairs are in turn provided to Critic, for Critic to deduce more new ranked pairs.

(4) The creator learns better with more data. ML models are inclined to get more accurate when given more training data. Creator continually receives more augmented training data and incrementally trains its model accordingly. As a consequence, its F-measure increases from 0.65 to 0.74 (resp. 0.81) after the first (resp. last) round.

Summary. We find the following. (1) Combining deep learning and logic rules makes a promising approach to deducing currency. GATE is the most accurate, *e.g.,* 0.866 in F-measure on Career, as opposed to 0.35 and 0.36 by rule-based UncertainRule and Improve3C, and 0.54 and 0.53 by ML-based $\mathsf{RANK}_{\mathsf{Bert}}$ and $\mathsf{Ditto}_{\mathsf{Rank}}$. (2) GATE only takes 7 rounds to terminate on COM, which has 1,983,698 tuples. (3) On average,

GATE is 43.8% and 7.8% more accurate than Critic and Creator, respectively, *i.e.,* the creator and critic indeed benefit each other. (4) GATE beats GATE_{NC} in efficiency (with the same accuracy) by 9.62X on average, up to 19.37X, verifying the usefulness of lazy evocation strategies. (5) GATE has competitive overall time against existing ML methods, *e.g.,* 1359s on COM, as opposed to 2514s by the fastest of them. Although rule-based methods are fast (they do not train models), their accuracy are low. (6) Creator is more accurate than its variants $\text{Creator}_{\text{NC}}$, $\text{Creator}_{\text{NE}}$, $\text{Creator}_{\text{NA}}$ and $\text{Creator}_{\text{itr}}$ by 19.5%, 22.3%, 12.2% and 10.1%, respectively, verifying the need for context-aware embedding, chronological encoding, adaptive margin and order justification, respectively.

Chapter 4

Linking Entities across Relations and Graphs

In this chapter, we focus on entity resolution to link entities across relations $\mathcal{D}$ and graphs G. To achieve this, we first convert entities in relational data $\mathcal{D}$ into a graph G_D, where each attribute value of an entity is mapped to a unique vertex, and the attribute itself is mapped to the corresponding edge between vertices. Then we propose a novel graph matching algorithm, parametric simulation, to link entities between these graphs.

4.1 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?

Example 4.1: 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 vary across enterprises, the orders can be uniformly expressed as relations, *e.g.,* Tables 4.1 and 4.2. As shown in Fig. 4.1, company A maintains a knowledge graph G for items it carries. Consider the following three scenarios.

(1) Given ordered item t_1 of Table 4.1, company A wants to check whether it is the item represented by vertex v_1 in graph G of Fig. 4.1. 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 (Koren et al., 2009).

	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 4.1: 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 4.2: Relation brand

Contributions & organization. We categorize this chapter as follows. To the best of our knowledge, we make a first effort to link entities across relations and graphs based on their semantics.

(1) System (Section 4.2). 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 (W3C, 2012b), and supports three modes. (a) 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) Alternatively, users may ask for all vertices in G that match a given tuple $t \in \mathcal{D}$. (c) One may also request HER to find all matches across $\mathcal{D}$ and G. These modes correspond to cases (1)–(3) of Example 4.1. In particular, VPair conducts real-time analysis as in, *e.g.,* (Whang et al., 2013), and APair is practiced for fine-grained advertising (Yan et al., 2011).

(2) A new notion (Section 4.3). 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 4.4). As parameters, we define the score functions with BERT-based embedding and metric learning models (Reimers and Gurevych, 2019; Devlin et al., 2019a), to quantify the semantic similarity between labels. We select top-k "properties" of a vertex via Long Short Term Memory (LSTM) network (Melis et al., 2017) for a bound k. Bounds are decided by random search (Bergstra and Bengio, 2012), a trade-off between efficiency and accuracy. Moreover, HER interacts with users to improve the parameter functions with feedback, which employs triplet loss function (Schroff et al., 2015) and majority voting to make the fine-tuning robust.

(4) Complexity and algorithm (Section 4.5). We show that parametric simulation takes quadratic time, as opposed to the intractability of graph homomorphism and subgraph isomorphism (cf. (Garey and Johnson, 1979)). 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 4.6). 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 4.7). 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 (Feng et al., 2016; Konda et al., 2016; Mudgal et al., 2018) and rule-based (Papadakis et al., 2018) 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 202 M 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.

□

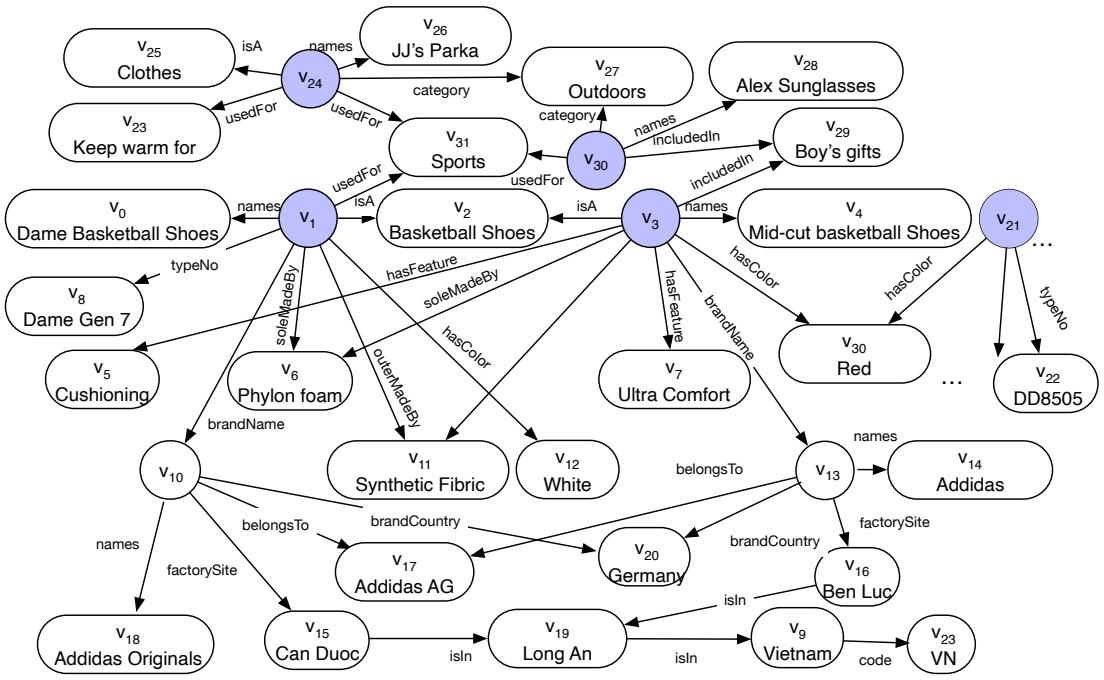


Figure 4.1: An e-commerce knowledge graph

4.2 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.

Intuitively, edge labels of Φ typify predicates, and vertex labels of Θ represent values. As will be seen in Section 4.4, we treat labels of Φ and Θ with different ML models.

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 4.3: Notations

Architecture. As shown in Fig. 4.2, 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 (W3C, 2012b), 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.

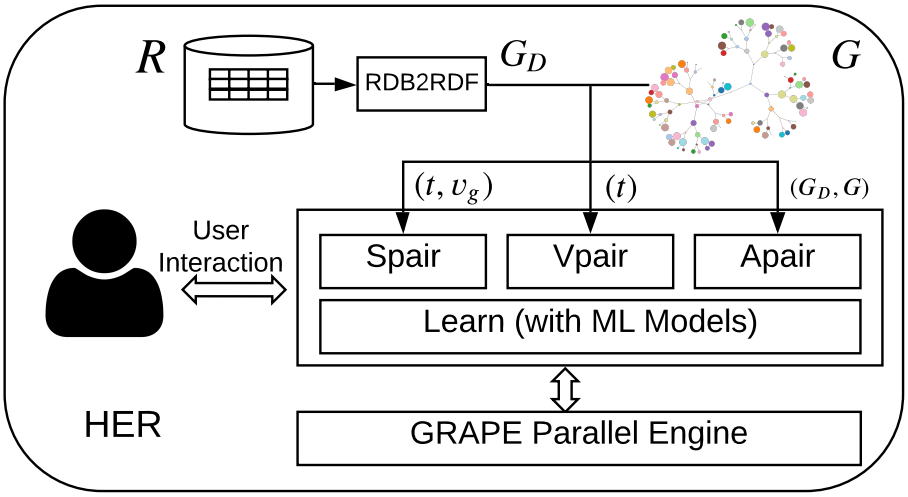


Figure 4.2: HER Architecture

(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, taking the learned parameters. We will define parametric simulation in Section 4.3, learn parameters in Section 4.4, and develop algorithms underlying SPair, VPair and APair in Sections 4.5 and 4.6. The algorithms run on top of GRAPE (Fan et al., 2018b; gra, 2020b), an open-source parallel graph engine.

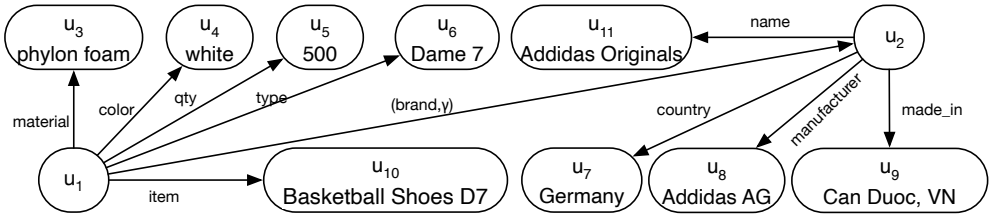


Figure 4.3: Graph representation of t_1 and b_1 in $\mathcal{D}$

RDB2RDF. We next present RDB2RDF. Several methods are in place for converting relations to graphs, *e.g.,* (Michel et al., 2014). For instance, Labelled Property Graph (LPG) (Angles, 2018) is another way to represent a relational data into a graph. However, it is designed for representing complex relationships and hierarchical structures while offering a high degree of flexibility, allowing for the representation of complex relationships. We take RDB2RDF (W3C, 2012a) for simplicity since we do not require hierarchical structures; But HER allows user to plug in other methods for representing relations as graphs.

Following direct mapping rules of RDB2RDF (Berners-Lee, 1998), 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 4.2: Figure 4.3 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. 4.3). □

4.3 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 $len(\rho)$, 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:

$$h_v(u', v') = \mathcal{M}_v(L_1(u'), L_2(v')) \tag{4.1}$$

$$h_\rho(\rho_1, \rho_2) = \frac{\mathcal{M}_\rho(L_1(\rho_1), L_2(\rho_2))}{len(\rho_1) + len(\rho_2)} \tag{4.2}$$

As will be seen in Section 4.4, $\mathcal{M}_v$ 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 $len(\rho_1) + len(\rho_2)$. Both $h_v(u', v')$ and $h_\rho(\rho_1, \rho_2)$ are in [0, 1].

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 4.7). 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 4.4). 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 4.3: Consider vertices u_6 in canonical graph G_D of Fig. 4.3 and v_8 in graph G of Fig. 4.1. 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. □

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 4.4.

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 4.4: Let σ=0.7, δ=1.5 and k=5. Vertices u_1 in Fig. 4.3 and v_1 of Fig. 4.1 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 4.4, and how to pick lineage sets in Section 4.5.

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 $\Pi(u_1, v_1) = \{(u_1, v_1)\} \cup S_{(u_1,v_1)} \cup S_{(u_2,v_{10})}$. □

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 $(h_v, h_\rho, h_s, \sigma, \delta, k)$. That is, parametric simulation retains the uniqueness of graph simulation (Milner, 1989).

4.4 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 (Reimers and Gurevych, 2019), 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 $h_v(u,v) \in [0,1]$. The semantic similarity of 1 indicates high similarity between vectors x_u and x_v, whereas a value of 0 signifies dissimilarity. In cases where the cosine similarity is negative, the semantic similarity is considered to be 0. Otherwise, it can range from 0 to 1.

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 (Devlin et al., 2019a) 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 typically special relation tokens for predicates, *e.g.,* "/akt:has-author" in a publication graph (DBLP, 2020a). 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 (Devlin et al., 2019a). 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 (Yang, 2001); this guides model training and fine-tuning (see more details in Section 4.7).

Example 4.5: Consider the candidate match (u_7, v_{20}) in the lineage set $S_{(u_2,v_{10})}$ of Example 4.4. 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 $h_\rho(L_D(\rho_1), L(\rho_2))=\mathcal{M}_\rho(L_D(\rho_1), L(\rho_2))/(len(\rho_1)+len(\rho_2)) = \mathcal{M}_\rho(\mathsf{country}, \mathsf{brandCountry})/2 = 0.75/2 = 0.375$. 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. □

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 Long Short Term Memory (LSTM) network as $\mathcal{M}_r$. Given one word as a start, LSTM generates a sequence of following words with reasonable semantic meanings (Melis et al., 2017).

(2) Function h_r ranks paths in P as follows. Given a path $\rho=(v_0, v_1, \ldots, v_l)$, we extend resource allocation (Lin et al., 2015) 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 4.6: Taking v_1 in Fig. 4.1 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 (Pennington et al., 2014)) 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 (Bergstra and Bengio, 2012), since grid search 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 noisy, which is a common practice for annotation quality control (Karger et al., 2011). Moreover, we employ triplet loss function (Schroff et al., 2015) 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 $G = (V, E, L)$.

4.5 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. 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 [true, $\mathcal{W}_1$] and [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.

(c) 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.

(d) 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.

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$[u', v']$.φ;
19. **else** match := ParaMatch$(G_D, G, (u', v'), \mathsf{FP})$;
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$(G_D, G, (u^p, v^p), \mathsf{FP})$;
32. **return** false;

Figure 4.4: Algorithm ParaMatch

Algorithm. Putting these together, we present ParaMatch in Fig. 4.4. 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 (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 vertices in 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 $[\text{false}, \emptyset]$ (line 28), and then re-runs ParaMatch to update stale cache entries that directly depend on (u,v) (lines 29-31). Finally, it returns false (line 32).

Example 4.7: Recall Example 4.4. 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 match, and the hashmap is updated: $\mathsf{cache}[u_2, v_{10}] = [\mathsf{true}, \{(u_7, v_{20}), (u_8, v_{17}), (u_9, v_9), (u_{11}, v_{18})\}]$.

(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 $\mathsf{cache}[u_1, v_1] = [\mathsf{true}, \{(u_2, v_{10}), (u_3, v_6), (u_4, v_{12}), (u_6, v_8), (u_{10}, v_0)\}]$, and returns true. □

Analyses. Algorithm ParaMatch is correct and takes quadratic time $O(|V_D|^2 + |V|^2)$ where $|V_D|$ and $|V|$ are the number of vertices in GD and G respectively. Indeed, (I) it takes the algorithm takes $O(|V_D||E_D| + |V||E|)$ time to select top-k descendants for each pair (u, v) (see Section 4.4); 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 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 $\mathsf{cache}[u, v].\mathcal{W}$ and (ii) $|\mathsf{cache}[u, v].\mathcal{W}| \leq k^2$; (c) line 1 of Fig. 4.4 takes $O(|V_D||V|)$ times in total; and (d) during each recursive call, all lines of Fig. 4.4 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.

In contrast, bounded simulation and strong simulation take $O(|V|(|V_D| + |E_D|)(|V| + |E|))$ time (Fan et al., 2010; Ma et al., 2014).

Theorem 3: *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.* □

Proof of Theorem 3. 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 4.3). 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). □

4.5.1 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 3. 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 4.3), 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. □

4.5.2 Example for the Challenges of Parametric Simulation

Given G_D and G in Fig. 4.5, 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).

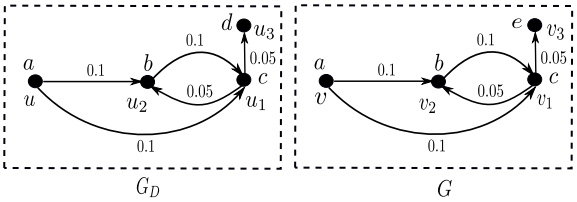


Figure 4.5: An example for rechecking

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 $0.1 = \delta$. 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 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)).

4.5.3 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 4.2), 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 4.5). We refer to $\Gamma(u_t, v_g)$ as the *schema matches* pertaining to (t, v_g).

Algorithm ParaMatch in Section 4.5 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 $\mathsf{cache}(u_t, v_g) = [\mathsf{true}, \mathcal{W}]$, where $\mathcal{W}$ is a set of pairwise matching properties of (u_t, v_g), *i.e.,* it consists of matches (u, v) for $u \in V^k_{u_t}$ and $v \in V^k_{v_g}$, 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 4.4). 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 4.4) 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 4.8: Continuing with Example 4.7, when ParaMatch terminates, schema matches $\Gamma(u_2, v_{10})$ is computed as follows. Since $\mathsf{cache}[u_2, v_{10}]$=$[\mathsf{true}, \mathcal{W}]$ 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 $\Gamma(u_1,v_1)$. □

4.6 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 4.6.1). We then parallelize these algorithms (Section 4.6.5).

4.6.1 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.

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. 4.6, VParaMatch first selects all vertices v_g in G with $h_v(u_t,v_g)\geq\sigma$, and initializes a set $C(u_t)$ with such candidates (u_t,v_g) (lines 2-3). It then sorts the pairs in $C(u_t)$ following the increasing order of degrees of vertices in $C(u_t)$ (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 $C(u_t)$ (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 (Zobel et al., 1998) on critical information (*e.g.,* years of papers in DBLP) as blocking strategies; for example, papers of the same year are in the same block of candidates.

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. $\Pi(u_t) := \emptyset$; $(u_t) := \emptyset$; initialize the hashmap cache;
2. **for each** $v_g \in V$ such that $h_v(u_t, v_g) \geq \sigma$ **do**
3. add $(u_t, v_g) \in (u_t)$;
4. sort matches (u, v) in (u_t) in increasing orders of degrees;
5. **for each** $(u, v) \in (u_t)$ **do**
6. remove (u, v) from (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** $\Pi(u_t)$;

Figure 4.6: Algorithm VParaMatch

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 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 5: 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.* □

Proof of Corollary 5. We only prove the correctness and complexity of VParaMatch; AllParaMatch can be shown similarly.

(1) For the correctness, observe that the set $C(u_t)$ consists of all possible candidates (lines 2-3) and VParaMatch verifies all candidates in $C(u_t)$ 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 $C(u_t)$; 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. □

4.6.2 Examples for Algorithm VParaMatch

Continuing with Example 4.1, 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 4.7, 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 $C(u_1)$ with candidate matches (*i.e.,* (u_1, v_1) and (u_1, v_3)). (2) It then inspects candidates in $C(u_1)$ along the same lines as Example 4.7, 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 (Zobel et al., 1998) 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

---

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. $\Pi := \emptyset$; $C := \emptyset$; 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 $(u_t, v_g) \in C$;
4. sort matches (u, v) in C in increasing orders of the degrees;
5. compute the set Π as VParaMatch;

Figure 4.7: Algorithm AllParaMatch

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.

4.6.3 Algorithm AllParaMatch

As shown in Fig. 4.7, 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 $C$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 4.9: Continuing with Example 4.1, 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 $C$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 C due to the different labels of vertices. It then checks candidates in C along the same lines as Example 4.7.

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. □

4.6.4 Fixpoint computation

Given fragmented graphs G_D and G, PAllMatch computes matches Π in parallel. It adopts the fixpoint model of GRAPE (Fan et al., 2018b; gra, 2020b,a). 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:

$$R_i^0 = \mathsf{PPSim}(F_i, \sigma, \delta, k), \tag{4.3}$$

$$R_i^{j+1} = \mathsf{IncPSim}(R_i^j, F_i, \sigma, \delta, k, M_i). \tag{4.4}$$

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 4.6.

4.6.5 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 (Valiant, 1990). 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 (Bourse et al., 2014). 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 (Bourse et al., 2014), 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 (Fan et al., 2018b; gra, 2020b). It starts with a procedure PPSim at each worker, and then iteratively runs procedure IncPSim to incrementally refine the result, as follows.

(1) PPSim. In the first superstep, each worker P_i starts by setting $\mathsf{cache}[u,v']$ 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.,* $\mathsf{cache}[u,v].\varphi$ 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 $v.\mathsf{status}$, which stores invalid matches (u,v) deduced. Initially, $v.\mathsf{status}$ 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 $v.\mathsf{status}$ 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 $\mathsf{cache}[u,v]$ 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.,* $\Pi = \cup_{i\in[1,n]} R_i^{r^*}$.

Theorem 6: *Given fragmented graphs G_D and G as above,* PAllMatch *correctly computes the match* Π *of* (G_D, G). □

Proof of Theorem 6. 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. 4.4, we can verify that given a candidate (u_0, v_0), ParaMatch returns true if and only if 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 4.6), $\mathcal{T}$ 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 4.6).

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. □

Remarks. (1) PAllMatch also works asynchronously unless out of space. Under the conditions of (Fan et al., 2018a) we can show that PAllMatch correctly computes Π under the adaptive asynchronous parallel model of (Fan et al., 2018a). (2) *IncPSim*

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

4.6.6 Implementation of HER on top of GRAPE

Implementation of parametric simulation on top of GRAPE is as follows:

(1)Algorithms run with n share-nothing workers $P_1,\ldots,P_n$, under the Bulk Synchronous Parallel (BSP) model (Valiant, 1990). 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 (Bourse et al., 2014). 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 $V_i^D \cup O_i^D$. Note that (i) partitioning G_D via edge-cut can reduce the communication. Since G_D is constructed from a relational database, G_D usually consists of a set of star graphs, where a star graph is a graph with a central vertex having edges to all other vertices (see Fig. 4.3 for an example); then when checking whether a candidate (u,v) is valid on such G_D, we only need to check the children of u and v, and do not need communication between fragments, since all such vertices are in the same fragment due to the edge-cut; (ii) As will be seen below, the vertices in O_i^D are used to synchronize computation between different fragments.

(3) Graph G is also partitioned into n fragments $F_1^G,\ldots,F_n^G$ via edge-cut (Bourse et al., 2014). Each fragment F_i^G is defined as $(V_i^G \cup O_i^G, E_i^G, L_i^G)$. We cannot partition G with arbitrary partition algorithms; this is because G_D has been partitioned, if G is also partitioned independently, some candidates matches (u,v) will be lost, *e.g., u* and v have the same labels, but are located in different fragments. To solve this, we can first partition G_D via edge-cut, and then partition G according to the fragmented G_D; more specifically, for each vertex u of G_D in fragment F_i^D we assign all vertices v in G with all its edges to the fragment F_i^D such that (u,v) is a candidate match, *i.e.,* $h_v(u,v) \geq \sigma$; this can be done using inverted index search.

(4) If G are not large, one can also replicate them to reduce communication cost. To simply the discussion, in the following we use fragment F_i to represent both the fragment F_i^D of G_D and the fragment F_i^G of G which are assigned to F_i^D.

4.7 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 4.4: (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 Websites (UK, 2020). (2) DBpediaP, subsets of DBPedia knowledge base about athletes and politicians in relations (dbp, 2020a) and graphs (dbp, 2020b). (3) DBLP, publication data in relations (DBLP, 2020b) and graphs (DBLP, 2020a). (4) Movie, movie data in relations(IMD, 2020b) and graphs(IMD, 2020a). (5) FBWIKI, consisting of (a) part of FreeBase (Bast et al., 2014) knowledge graph and (b) entries of people extracted from the wikidata (wik, 2020).

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).

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
Movie	7.5M	34.2M	2.3M	5.4M	movies
FBWIKI	4.0M	7.4M	60.8M	362.2M	knowledge base

Table 4.4: Real-life datasets for evaluation

ML models. For $\mathcal{M}_v$, we employed Sentence-bert (Reimers and Gurevych, 2019), a pre-trained sentence embedding model for its high accuracy. For $\mathcal{M}_\rho$, we first constructed an edge label corpus (see Section 4.4) for pre-training BERT (Devlin et al., 2019a), which contains 195K labels in 1845 categories from all datasets. The pre-training 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 (pyt, 2020), except 650 hidden units per layer. When collecting training paths for $\mathcal{M}_r$, we took paths of at most 4 edges following (Lin et al., 2015) 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 4.4), we manually annotated 5000 matches as ground truth for all datasets, assisted by a mapping provided by Google (fre, 2020). 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.

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

Table 4.5: Accuracy of HER and baseline models (F-measure)

Baselines. We used five baselines. (1) MAGNN (Fu et al., 2020), a GNN-based model that learns vertex embeddings for similarity, with both vertex attributes and meta-paths. We implemented it using the default configuration in (Fu et al., 2020) in Pytorch to extract embedding, and cascaded a 3-layer neural network as a classifier. (2) Bsim, bounded simulation (Fan et al., 2010), based on vertex labels and topological matching. (3) JedAI (Papadakis et al., 2018), 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". The threshold parameters were set default as (Papadakis et al., 2020), which has been verified generally effective for different datasets. As a state-of-the-art non-ML system, the accuracy of JedAI is comparable to many ML models (Papadakis et al., 2020). (4) MAG (Konda et al., 2016) (Magellan), a state-of-the-art ML-based system for ER on relations. We adopted the configuration in (Mag, 2020), using its random forest model with feature

tables. MAG is mainly implemented in Python, with several core functions optimized via C++. (5) DEEP (Mudgal et al., 2018), a deep-learning-based ER Python package under Magellan, configured as in (Dee, 2020) with the default hybrid model for matching.

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

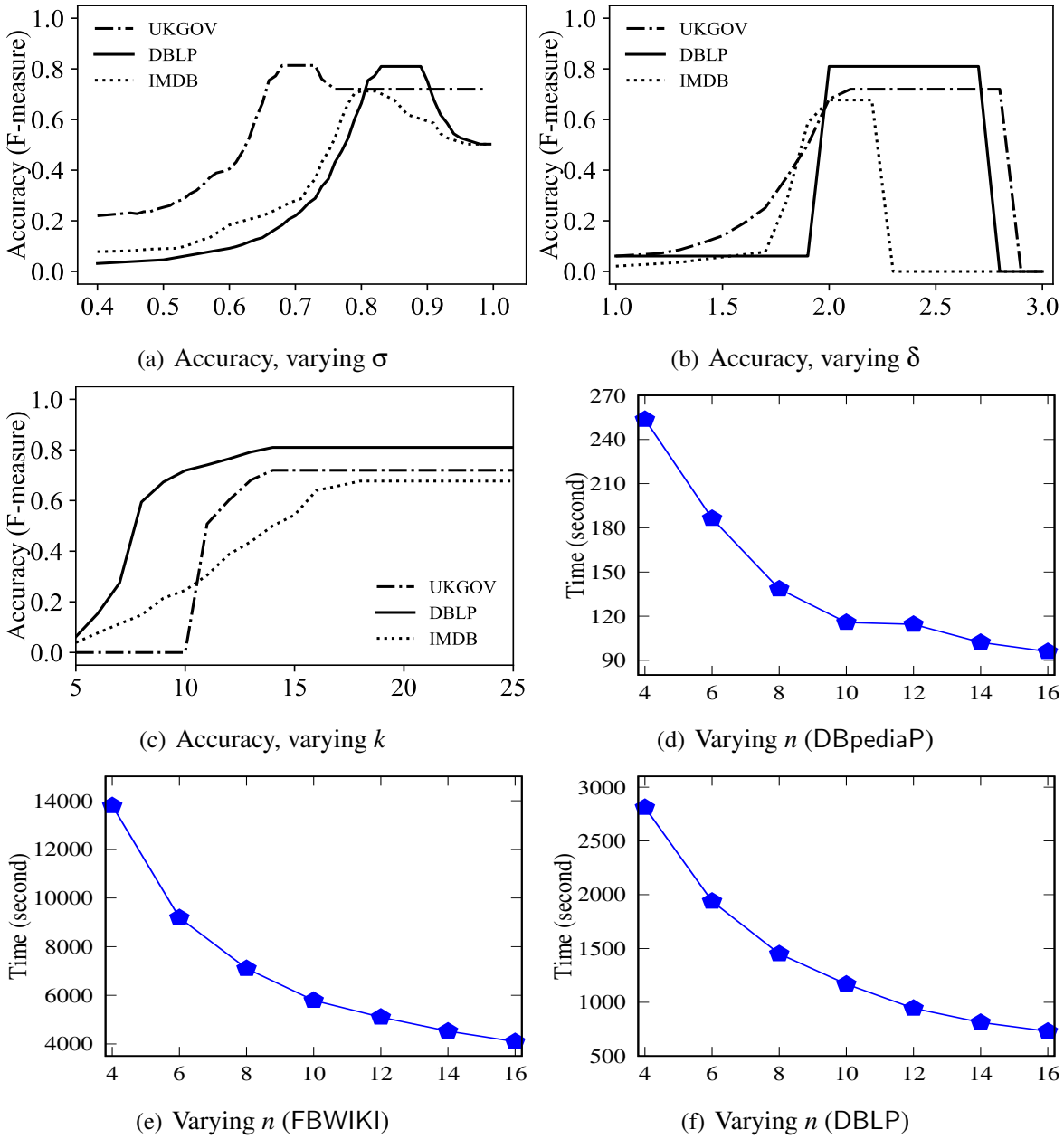
For a fair comparison, we tested the baselines and HER with a single machine using an Intel Xeon 2.5 GHz CPU and 192 GB memory, since only Bsim has a parallel solution among the five baselines. We also tested the parallel scalability of HER and Bsim on GRAPE (Fan et al., 2018b; gra, 2020b), using an 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 accuracy using all datasets in Table 4.4. As shown in Table 4.5, it is 0.94 for HER on average, consistently outperforming all the baselines. Bsim ran out of memory (OM) for all datasets, giving no results.

(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 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



(a) Accuracy, varying σ (b) Accuracy, varying δ

(c) Accuracy, varying k (d) Varying n (DBpediaP)

(e) Varying n (FBWIKI) (f) Varying n (DBLP)

Figure 4.8: HER accuracy,scalability and efficiency with varying parameters

and graphs needs both inductive topological matching and ML models, beyond existing rules.

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. 4.8(a), F-measure first grows steadily when σ increases; it reaches the peak and then drops sharply with larger σ. This is because when using lower σ values, an extensive number of pairs are considered as matches, leading to a decrease in recall.

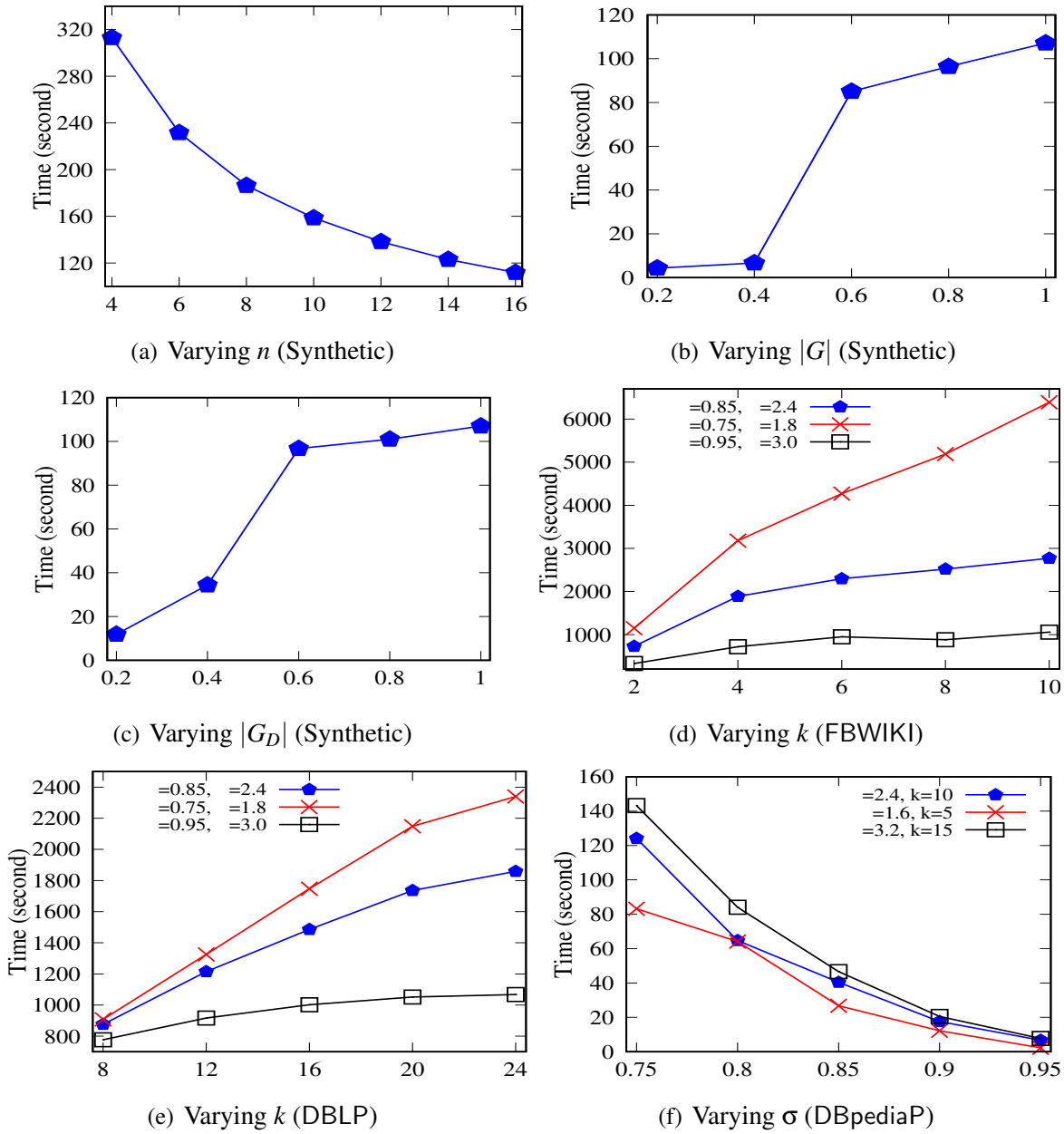
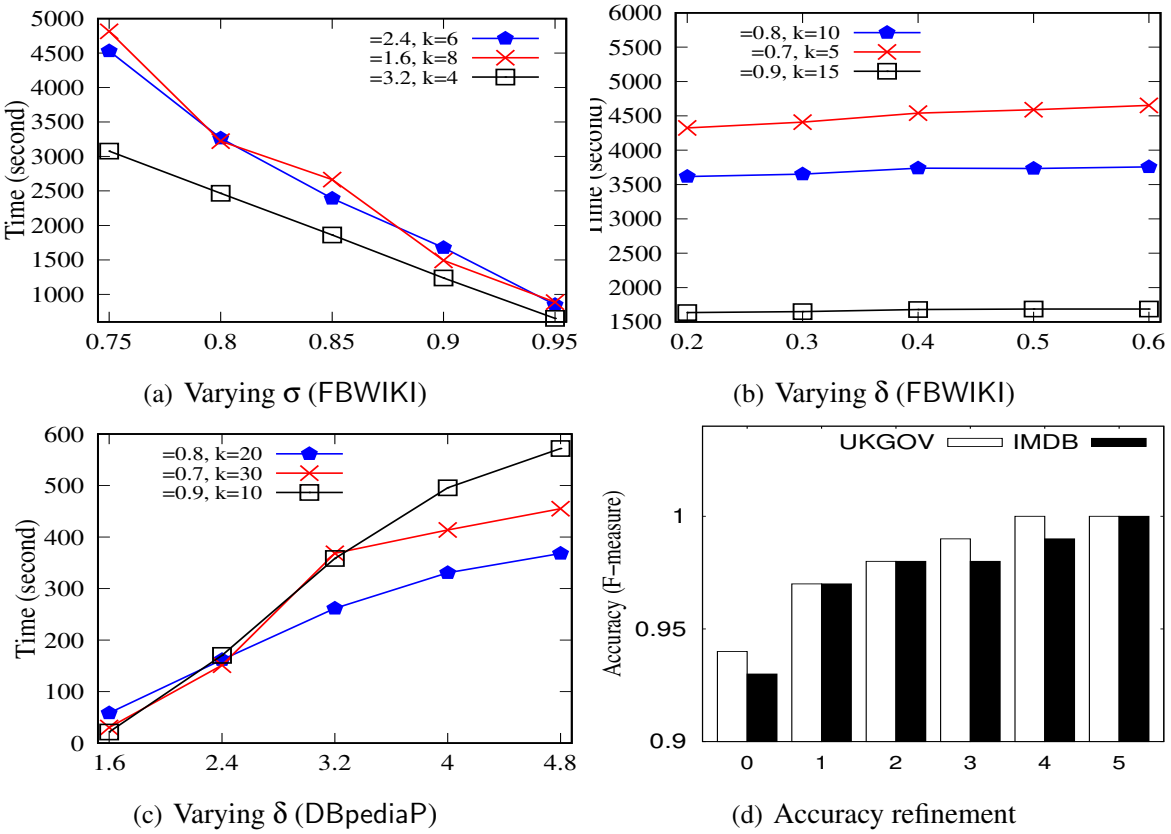


Figure 4.9: HER accuracy,scalability and efficiency with varying parameters

Conversely, setting a higher σ results in only perfect matches being considered, leading to decrease in precision. 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. 4.8(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. 4.8(c), F-measure first increases and then remains stable after k reaches a value



(a) Varying σ (FBWIKI) (b) Varying δ (FBWIKI)

(c) Varying δ (DBpediaP) (d) Accuracy refinement

Figure 4.10: HER accuracy,scalability and efficiency with varying parameters

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.

	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 4.6: Sequential execution time (s)

Exp-2: Efficiency. Over real-life datasets DBpediaP and DBLP, we report in Table 4.6 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.

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 fair comparison, as HER firstly applies inverted index to quickly 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 that 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 4.8(d)–4.9(a), 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. We have the potential for further enhancement in these results if an alternative graph partitioning approach is employed. The primary challenge lies in optimizing the communication process, particularly concerning bridge vertices, which exchange messages at each super step.

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 whole 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 4.9(c) shows that the execution time increases with larger $|G_D|$ and HER takes 107s when $|G|$=202M and $|G_D|$=342M. In this scenario, increasing the size of G_D leads to an increase in runtime. Determining a precise speed factor is inherently challenging, primarily due to the experimental nature of this evaluation, which is why Figure 4.9(c) has been included. It is worth emphasizing that the potential for improvement is largely possible on the availability of an ideal graph partitioning algorithm capable of significantly reducing message exchanges. In an optimal scenario where no messages are sent, achieving perfect parallelism would be attainable.

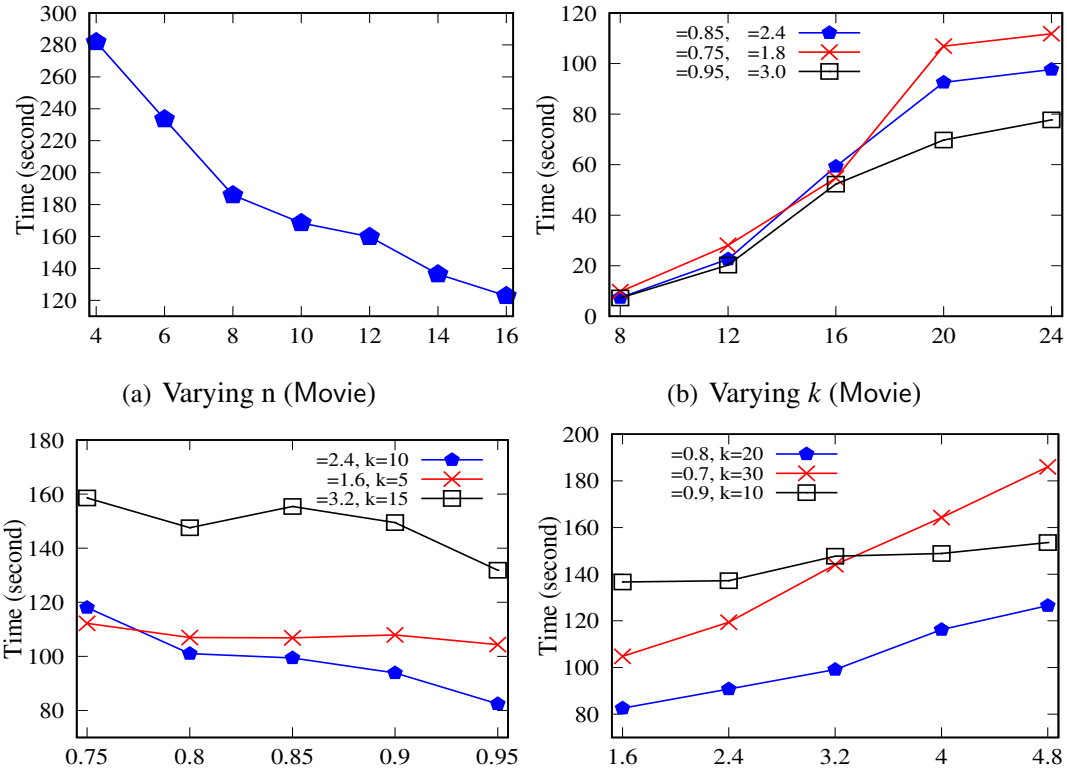
(2) Varying $|G|$. Figure 4.9(b) reports results over the entire G_D by varying $|G|$. The results are consistent with Figure 4.9(c).

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 4.9(d) and 4.9(e) 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 4.9(f) and 4.10(a) 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 4.10(b) and 4.10(c)).

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 varying range of 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. We also report the scalability and efficiency of HER in the APair mode on Movie in Figure 4.11. 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 Movie (see Fig. 4.11(a)); and (2) Figures 4.11(b)-4.11(d) show the efficiency of APair on Movie 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 other datasets in Section 4.7.



(a) Varying n (Movie)

(b) Varying k (Movie)

(c) Varying σ (Movie)

(d) Varying δ (Movie)

Figure 4.11: HER scalability and efficiency on Movie

Exp-4: Refinement. We next tested the impact of user interaction on the accuracy, using UKGOV and Movie. We have chosen these datasets randomly among others. 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 4.4. As shown in Figure 4.10(d), F-measure goes up by 3% and 4% on UKGOV and Movie, respectively, in the first round, and 5 rounds suffice for HER to reach 100% accuracy. The results on other datasets are consistent (not shown).

Summary. We find the following. (1) HER is more accurate than ML and rule-based methods. On average HER beats MAGNN, JedAI, MAG and DEEP by 31%, 42%, 22% and 17% in accuracy, respectively. (2) HER also performs the best in efficiency. When running in the VPair mode on a single machine, it is 90 times faster than these baselines on average. (3) When $|G|$=202M and $|G_D|$=342M, SPair and VPair take 0.68ms and 15.3s on a single machine, respectively, and APair takes 107s using 16 machines. In contrast, all the baselines could not finish in hours. (4) HER scales well with the number n of processors. It is on average 3.2 times faster when n varies from 4 to 16. (5) At most 5 rounds of user interaction suffice to fine-tune the ML models in parametric simulation. While this finding is based on experimental data, it is important to note that this value may vary across different datasets. The results presented here pertain specifically to the datasets utilized in our experimental settings.

Chapter 5

Conclusion

This thesis introduces two different approaches for improving data quality. In Chapter 3, (1) We formulate a new problem for determining the timeliness of attribute values. (2) As a solution to the problem, we propose a creator-critic framework by combining deep learning and logic deduction, for the two to enhance each other. (3) We develop a novel ranking model to learn temporal orders on attribute values. (4) We show how to justify the learned orders, deduce more ranked pairs and provide feedback for the learner, by extending the chase using CCs. A potential limitation of the deep learning approach is that it does not consider higher order attribute correlation other than all attribute serialization. This may be alleviated using methods (*e.g.,* (Wang et al., 2021; Cheng et al., 2016)) that consider high order attribute interaction.

In Chapter 4, 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. The ML models help us to link entities semantically. However, limitation of these models is that their accuracy depend on training data, hyperparameters and the model itself, which (i) creating training data for vertex and edge closeness might be biased (ii) and is hard to control hyperparameters and (iii) the model limitation to catch relatedness of vertex and edge values.

One future topic for data currency is to study how to catch conflicts and missing values given temporal orders. Another topic is to extend CCs (Fan et al., 2012) by embedding ranking models as predicates, to improve the ranking accuracy with logic conditions and interpret ranking in logic.

Another topic for future work for entity linking is to extend HER to other data formats, *e.g.,* JSON. Another topic is to extract, integrate and query data of different sources in data lakes (Nargesian et al., 2019) with HER.

Bibliography

(2020a). DBpedia as tables. http://web.informatik.uni-mannheim.de/DBpediaAsTables/DBpedia-en-2016-04/csvhttps://wiki.dbpedia.org.

(2020b). DBpedia version 2016-10. https://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10https://wiki.dbpedia.org.

(2020). Deepmatcher Introduction. https://nbviewer.jupyter.org/github/anhaidgroup/deepmatcher/blob/master/examples/end_to_end_em.ipynbhttps://github.com/anhaidgroup/deepmatcher.

(2020). Freebase/wikidata mappings. *https://developers.google.com/freebase*.

(2020a). GRAPE. https://github.com/alibaba/libgrape-lite.githttps://github.com/alibaba/libgrape-lite.git.

(2020b). Graphscope. *https://graphscope.io/*https://graphscope.io/.

(2020a). Movie graph dataset. https://www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-latest-dump.ziphttps://www.cs.toronto.edu.

(2020b). Movie relational dataset. https://datasets.imdbws.com/https://datasets.imdbws.com/.

(2020). http://nbviewer.jupyter.org/github/anhaidgroup/py_entitymatching/blob/master/notebooks/guides/end_to_end_em_guides/Basic%20EM%20Workflow%20Restaurants%20-%201.ipynbBasic EM workflow 1 (Restaurants data set).

(2020). https://github.com/pytorch/examples/tree/master/word_language_modelWord-level language modeling RNN, PyTorch example.

(2020). Wikidata. *https://www.wikidata.org/*.

(2022). Gartner report. https://www.forbes.com/sites/forbestechcouncil/2021/10/14/flying-blind-how-bad-data-undermines-business/?sh=7b25643b29e8/.

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Angles, R. (2018). The property graph database model. In *AMW*.

Arenas, M., Bertossi, L., and Chomicki, J. (1999). Consistent query answers in inconsistent databases. In *PODS*.

Bast, H., Bäurle, F., Buchhold, B., and Haußmann, E. (2014). Easy access to the freebase dataset. In *WWW*, page 95–98.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *JMLR*, 13(1):281–305.

Berners-Lee, T. (1998). Relational databases and the semantic Web (in design issues). *World Wide Web Consortium*.

Bleifuß, T., Kruse, S., and Naumann, F. (2017). Efficient denial constraint discovery with Hydra. *PVLDB*, 11(3):311–323.

Bose, R. J., Mans, R. R., and Aalst, V. D. W. W. (2013). Wanna improve process mining results? it's high time we consider data quality issues seriously. In *Computational Intelligence & Data Mining*.

Bourse, F., Lelarge, M., and Vojnovic, M. (2014). Balanced graph edge partition. In *SIGKDD*, pages 1456–1465.

Bramsen, P., Deshpande, P., Lee, Y. K., and Barzilay, R. (2006). Inducing temporal graphs. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. (2005). Learning to rank using gradient descent. In *international conference on Machine learning*, pages 89–96.

Burges, C. J. (2010). From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81.

Businesswire (2022). Over 80 percent of companies rely on stale data for decision-making. *https://www.businesswire.com/news/home/20220511005403/en/Over-80-Percent-of-Companies-Rely-on-Stale-Data-for-Decision-Making*.

Canada, S. (2022). Classification of legal marital status. `https://www23.statcan.gc.ca/imdb/p3VD.pl?Function=getVD&TVD=61748&CVD=61748&CLV=0&MLV=1&D=1`.

Chambers, N. and Jurafsky, D. (2008). Jointly combining implicit constraints improves temporal ordering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 698–706. ACL.

Chapelle, O. and Chang, Y. (2011). Yahoo! learning to rank challenge overview. In *Proceedings of the learning to rank challenge*, pages 1–24. PMLR.

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al. (2016). Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10.

Christophides, V., Efthymiou, V., Palpanas, T., Papadakis, G., and Stefanidis, K. (2019). End-to-end entity resolution for big data: A survey. *ACM Comput. Surv.*, 53(6).

Chu, X., Ilyas, I. F., and Papotti, P. (2013). Discovering denial constraints. *PVLDB*, 6(13):1498–1509.

Church, K. W. (2017). Word2vec. *Natural Language Engineering*, 23(1):155–162.

Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2004). A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372.

DBLP (2020a). DBLP RDF data. `http://dblp.rkbexplorer.com/models/dump.tgz`http://dblp.rkbexplorer.com.

DBLP (2020b). DBLP relational data. https://dblp.org/xml/release/dblp-2015-04-01.xml.gzhttps://dblp.org.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019a). BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019b). BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186.

Dikeoulias, I., Amin, S., and Neumann, G. (2022). Temporal knowledge graph reasoning with low-rank and model-agnostic representations. *CoRR*, abs/2204.04783.

Ding, X., Wang, H., Gao, Y., Li, J., and Gao, H. (2017). Efficient currency determination algorithms for dynamic data. *Tsinghua Science and Technology*, 22(3):227–242.

Ding, X., Wang, H., Su, J., Li, J., and Gao, H. (2018). Improve3c: Data cleaning on consistency and completeness with currency. *arXiv preprint arXiv:1808.00024*.

Ding, X., Wang, H., Su, J., Wang, M., Li, J., and Gao, H. (2020). Leveraging currency for repairing inconsistent and incomplete data. *TKDE*.

Divakaran, A. and Mohan, A. (2020). Temporal link prediction: A survey. *New Gener. Comput.*, 38(1):213–258.

Dong, X., Berti-Equille, L., and Srivastava., D. (2009). Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1):562–573.

Dong, X. L., Berti-Equille, L., Hu, Y., and Srivastava, D. (2010). Global detection of complex copying relationships between sources. In *PVLDB*.

Dong, Y., Chawla, N. V., and Swami, A. (2017). metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*.

Duan, X., Guo, B., Shen, Y., Shen, Y., Dong, X., and Zhang, H. (2020). Research on parallel data currency rule algorithms. In *International Conference on Information Science and System*, pages 24–28.

Duh, K. K. (2009). *Learning to rank with partially-labeled data*. University of Washington.

Exasol (2020). Exasol research finds 58% of organizations make decisions based on outdated data.
https://www.exasol.com/news-exasol-research-finds-organizations-make-decisions-based-on-outdated-data/.

Fan, W., Fan, Z., Tian, C., and Dong, X. L. (2015). Keys for graphs. *PVLDB*, 8(12):1590–1601.

Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2):6:1–6:48.

Fan, W., Geerts, F., Tang, N., and Yu, W. (2013a). Inferring data currency and consistency for conflict resolution. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 470–481. IEEE.

Fan, W., Geerts, F., Tang, N., and Yu, W. (2013b). Inferring data currency and consistency for conflict resolution. In *ICDE*, pages 470–481. IEEE.

Fan, W., Geerts, F., Tang, N., and Yu, W. (2014a). Conflict resolution with data currency and consistency. *Journal of Data and Information Quality (JDIQ)*, 5(1-2):1–37.

Fan, W., Geerts, F., Tang, N., and Yu, W. (2014b). Conflict resolution with data currency and consistency. *Journal Data and Information Quality (JDIQ)*, 5(1-2):6:1–6:37.

Fan, W., Geerts, F., and Wijsen, J. (2012). Determining the currency of data. *TODS*, 37(4):25:1–25:46.

Fan, W., Jin, R., Lu, P., Tian, C., and Xu, R. (2022). Towards event prediction in temporal graphs. *PVLDB*, 15(9):1861–1874.

Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., and Wu, Y. (2010). Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1).

Fan, W., Lu, P., Luo, X., Xu, J., Yin, Q., Yu, W., and Xu, R. (2018a). Adaptive asynchronous parallelization of graph algorithms. In *SIGMOD*.

Fan, W., Lu, P., and Tian, C. (2020). Unifying logic rules and machine learning for entity enhancing. *Sci. China Inf. Sci.*, 63(7).

Fan, W., Tian, C., Wang, Y., and Yin, Q. (2021). Parallel discrepancy detection and incremental detection. *PVLDB*, 14(8):1351–1364.

Fan, W., Yu, W., Xu, J., Zhou, J., Luo, X., Yin, Q., Lu, P., Cao, Y., and Xu, R. (2018b). Parallelizing sequential graph computations. *TODS*, 43(4).

Feng, J., Huang, M., Yang, Y., and Zhu, X. (2016). GAKE: Graph aware knowledge embedding. In *COLING*, pages 641–651.

Freund, Y., Iyer, R., Schapire, R. E., and Singer, Y. (2003). An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969.

Fu, X., Zhang, J., Meng, Z., and King, I. (2020). MAGNN: metapath aggregated graph neural network for heterogeneous graph embedding. In *WC*.

Gallagher, B. (2006). Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, pages 45–53.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

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

Golshan, B., Halevy, A. Y., Mihaila, G. A., and Tan, W. (2017). Data integration: After the teenage years. In *PODS*.

Govement, S. M. (2022). Self-employed entrepreneurs. *https://opendata.sz.gov.cn/data/dataSet/toDataDetails/29200_01300931*.

Goyal, T. and Durrett, G. (2019). Embedding time expressions for deep temporal ordering models. In *Conference of the Association for Computational Linguistics (ACL)*. ACL.

Han, S., Wang, X., Bendersky, M., and Najork, M. (2020). Learning-to-rank with bert in tf-ranking. *arXiv preprint arXiv:2004.08476*.

Hernández, M., Koutrika, G., Krishnamurthy, R., Popa, L., and Wisnesky, R. (2013). Hil: a high-level scripting language for entity integration. In *Proceedings of the 16th international conference on extending database technology*, pages 549–560.

Herzog, T. N., Scheuren, F. J., and Winkler, W. E. (2007). *Data quality and record linkage techniques*, volume 1. Springer.

Hu, Y., Da, Q., Zeng, A., Yu, Y., and Xu, Y. (2018). Reinforcement learning to rank in e-commerce search engine: Formalization, analysis, and application. In *SIGKDD*, pages 368–377.

Isele, R., Jentzsch, A., and Bizer, C. (2010). Silk server - adding missing links while consuming linked data. In *COLD*, volume 665.

Jeh, G. and Widom, J. (2002). Simrank: A measure of structural-context similarity. In *KDD*, pages 538–543.

Karger, D. R., Oh, S., and Shah, D. (2011). Iterative learning for reliable crowdsourcing systems. In *NIPS*, pages 1953–1961.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR (Poster)*.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N. C., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., and Hadsell, R. (2016). Overcoming catastrophic forgetting in neural networks. *CoRR*, abs/1612.00796.

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

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Kurniati, A. P., Rojas, E., Hogg, D., Hall, G., and Johnson, O. A. (2019). The assessment of data quality issues for process mining in healthcare using medical information mart for intensive care iii, a freely available e-health record database. *Health informatics journal*, 25(4):1878–1893.

Kusumoto, M., Maehara, T., and Kawarabayashi, K.-i. (2014). Scalable similarity search for simrank. In *SIGMOD*, pages 325–336.

Kwashie, S., Liu, L., Liu, J., Stumptner, M., Li, J., and Yang, L. (2019). Certus: An effective entity resolution approach with graph differential dependencies (GDDs). *PVLDB*, 12(6):653–666.

Leone, S. (2022). Fifa 22 complete player dataset. `https://www.kaggle.com/stefanoleone992/fifa-21-complete-player-dataset`.

Li, B., Wang, W., Sun, Y., Zhang, L., Ali, M. A., and Wang, Y. (2020a). Grapher: Token-centric entity resolution with graph convolutional neural networks. In *AAAI*.

Li, M. and Li, J. (2016). A minimized-rule based approach for improving data currency. *J. Comb. Optim.*, pages 812–841.

Li, M., Li, J., Cheng, S., and Sun, Y. (2018). Uncertain rule based method for determining data currency. *IEICE TRANSACTIONS on Information and Systems*, 101(10):2447–2457.

Li, M. and Sun, Y. (2018). Currency preserving query: Selecting the newest values from multiple tables. *IEICE TRANSACTIONS on Information and Systems*, 101(12):3059–3072.

Li, Y., Li, J., Suhara, Y., Doan, A., and Tan, W. (2020b). Deep entity matching with pre-trained language models. *PVLDB*, 14(1):50–60.

Liang, Y., Duan, X., Ding, Y., Kou, X., and Huang, J. (2019). Data mining of students' course selection based on currency rules and decision tree. In *International Conference on Big Data and Computing*, pages 247–252.

Lin, Y., Liu, Z., Luan, H., Sun, M., Rao, S., and Liu, S. (2015). Modeling relation paths for representation learning of knowledge bases. In *EMNLP*.

Little, A. (2020). Outdated data: Worse than no data? *https://info.aldensys.com/joint-use/outdated-data-is-worse-than-no-data#: :text=Obsolete%20data%20about%20the%20condition,too%20old%20to%20be%20reliable.*

Liu, T. (2009). Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331.

Liu, T. (2010). Learning to rank for information retrieval. In *SIGIR*.

Livshits, E., Heidari, A., Ilyas, I. F., and Kimelfeld, B. (2020). Approximate denial constraints. *PVLDB*, 13(10):1682–1695.

Ma, S., Cao, Y., Fan, W., Huai, J., and Wo, T. (2014). Strong simulation: Capturing topology in graph pattern matching. *TODS*, 39(1):4:1–4:46.

Martin, N., Martinez-Millana, A., Valdivieso, B., and Fernandez-Llatas, C. (2019). Interactive data cleaning for process mining: A case study of an outpatient clinic's appointment system. In *International Conference on Business Process Management*.

Melis, G., Dyer, C., and Blunsom, P. (2017). On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

Meta (2022). Meta report. `https://investor.fb.com/investor-news/press-release-details/2023/Meta-Reports-Fourth-Quarter-and-Full-Year-2022-Results/default.aspx/`.

Michel, F., Montagnat, J., and Zucker, C. F. (2014). A survey of RDB to RDF translation approaches and tools.

Milner, R. (1989). *Communication and concurrency*. Prentice hall.

Mudgal, S., Li, H., Rekatsinas, T., Doan, A., Park, Y., Krishnan, G., Deep, R., Arcaute, E., and Raghavendra, V. (2018). Deep learning for entity matching: A design space exploration. In *SIGMOD*.

Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., and Arocena, P. C. (2019). Data lake management: Challenges and opportunities. *PVLDB*, 12(12):1986–1989.

Ngomo, A. N. and Auer, S. (2011). LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*.

Ning, Q., Feng, Z., and Roth, D. (2017). A structured learning approach to temporal relation extraction. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1027–1037. ACL.

Ning, Q., Wu, H., Peng, H., and Roth, D. (2018). Improving temporal relation extraction with a globally acquired statistical resource. In *Conference of the North*

American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, pages 841–851. ACL.

Nogueira, R. F. and Cho, K. (2019). Passage re-ranking with BERT. *CoRR*, abs/1901.04085.

Papadakis, G., Ioannou, E., Palpanas, T., Niederee, C., and Nejdl, W. (2012). A blocking framework for entity resolution in highly heterogeneous information spaces. *TKDE*, 25(12):2665–2682.

Papadakis, G., Mandilaras, G., Gagliardelli, L., Simonini, G., Thanos, E., Giannakopoulos, G., Bergamaschi, S., Palpanas, T., and Koubarakis, M. (2020). Three-dimensional entity resolution with JedAI. *Inf. Syst.*

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

Pasumarthi, R. K., Bruch, S., Wang, X., Li, C., Bendersky, M., Najork, M., Pfeifer, J., Golbandi, N., Anil, R., and Wolf, S. (2019). Tf-ranking: Scalable tensorflow library for learning-to-rank. In *SIGKDD*, pages 2970–2978.

Pena, E. H. M., de Almeida, E. C., and Naumann, F. (2019). Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3):266–278.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP*.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *NAACL*.

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

Royal Mail (2018). Dynamic customer data in a digital world: Data services insight report. *https://www.royalmail.com/business/system/files/royal-mail-data-services-insight-report-2018.pdf*.

Sadeghian, A., Armandpour, M., Colas, A., and Wang, D. Z. (2021). Chronor: Rotation based temporal knowledge graph embedding. In *AAAI*, pages 6471–6479. AAAI Press.

Sadri, F. and Ullman, J. D. (1980). The interaction between functional dependencies and template dependencies. In *SIGMOD*.

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

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

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

Tay, Y., Phan, M. C., Tuan, L. A., and Hui, S. C. (2017). Learning to rank question answer pairs with holographic dual lstm architecture. In *SIGIR*, pages 695–704.

TikTok (2022). Tiktok report. `https://newsroom.tiktok.com/en-us/1-billion-people-on-tiktok/`.

Tourille, J., Ferret, O., Névéol, A., and Tannier, X. (2017). Neural architecture for temporal relation extraction: A bi-lstm approach for detecting narrative containers. In *ACL*, pages 224–230. ACL.

Trivedi, R., Dai, H., Wang, Y., and Song, L. (2017). Know-Evolve: Deep temporal reasoning for dynamic knowledge graphs. In *International Conference on Machine Learning (ICML)*, volume 70, pages 3462–3471. PMLR.

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

UK (2020). Government open data. `https://opendata.camden.gov.uk`https://opendata.camden.gov.uk.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42.

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

W3C (2012a). R2RML: RDB to RDF mapping language.

W3C (2012b). Relational databases to RDF (RDB2RDF).

Wang, H., Ding, X., Li, J., and Gao, H. (2018). Rule-based entity resolution on database with hidden temporal information. *TKDE*, 30(11):2199–2212.

Wang, R., Shivanna, R., Cheng, D., Jain, S., Lin, D., Hong, L., and Chi, E. (2021). Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the web conference 2021*, pages 1785–1797.

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

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

Winkler, W. E. (2014). Matching and record linkage. *Wiley interdisciplinary reviews: Computational statistics*, 6(5):313–325.

Xu, J., He, X., and Li, H. (2020). Deep learning for matching in search and recommendation. *Found. Trends Inf. Retr.*, 14(2-3):102–288.

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

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

Yao, J., Dou, Z., Xu, J., and Wen, J.-R. (2021). Rlps: A reinforcement learning–based framework for personalized search. *TOIS*, 39(3):1–29.

Youtube (2022). Youtube report. `https://storage.googleapis.com/transparencyreport/report-downloads/pdf-report-22_2022-1--6-30_en_v1.pdf/`.

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

Zhang, J., Shen, F., Xu, X., and Shen, H. T. (2020a). Temporal reasoning graph for activity recognition. *IEEE Trans. Image Process.*

Zhang, M., Liu, Y., Luan, H., and Sun, M. (2016). Listwise ranking functions for statistical machine translation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(8):1464–1472.

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

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

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