ENFORCING TEMPORAL AND ONTOLOGICAL DEPENDENCIES OVER GRAPHS

ENFORCING TEMPORAL AND ONTOLOGICAL DEPENDENCIES OVER GRAPHS

BY

MORTEZA ALIPOURLANGOURI, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2022) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Enforcing Temporal and Ontological Dependencies over Graphs
AUTHOR:	Morteza Alipourlangouri M.Sc. (Software Engineering), Iran University of Science and Technology, Tehran, Iran
SUPERVISOR:	Dr. Fei Chiang
NUMBER OF PAGES:	xv, 171

Abstract

Graphs provide powerful abstractions, and are widely used in different areas. There has been an increasing demand in using the graph data model to represent data in many applications such as network management, web page analysis, knowledge graphs, social networks. These graphs are usually dynamic and represent the time evolving relationships between entities. Enforcing and maintaining data quality in graphs is a critical task for decision making, operational efficiency and accurate data analysis as recent studies have shown that data scientists spend 60-80% of their time cleaning and organizing data [2]. This effort motivates the need for effective data cleaning tools to reduce the user burden. The study of data quality management focuses along a set of dimensions, including data consistency, data deduplication, information completeness, data currency, and data accuracy. Achieving all these data characteristics is often not possible in practice due to personnel costs, and for performance reasons. In this thesis, we focus on tackling three problems in two data quality dimensions: data consistency and data deduplication.

To address the problem of data consistency over temporal graphs, we present a new class of data dependencies called Temporal Graph Functional Dependency (TGFDs). TGFDs generalize functional dependencies to temporal graphs as a sequence of graph snapshots that are induced by time intervals, and enforce both topological constraints and attribute value dependencies that must be satisfied by these snapshots. We establish the complexity results for the satisfiability and implication problems of TGFDs. We propose a sound and complete axiomatization system for TGFDs. We also present efficient parallel algorithms to detect inconsistencies in temporal graphs as violations of TGFDs. To address the data deduplication problem, we first address the problem of key discovery for graphs. Keys for graphs use topology and value constraints to uniquely identify entities in a graph database and keys are the main tools for data deduplication in graphs. We present two properties that define a key, including *minimality* and *support* and an algorithm to mine keys over graphs via frequent subgraph expansion. However, existing key constraints identify entities by enforcing label equality on node types. These constraints can be too restrictive to characterize structures and node labels that are syntactically different but semantically equivalent. Lastly, we propose a new class of key constraints, Ontological Graph Keys (OGKs) that extend conventional graph keys by ontological subgraph matching between entity labels and an external ontology. We study the entity matching problem with OGKs. We develop efficient algorithms to perform entity matching based on a Chase procedure. The proposed dependencies and algorithms in this thesis improve consistency detection in temporal graphs, automate the discovery of keys in graphs, and enrich the semantic expressiveness of graph keys.

Dedicated to my beloved wife, Zahra,

and to my beautiful twins, Deljhin and Nahal

Love you lots.

Acknowledgements

First, I would like to offer my most sincere gratitude to my supervisor, Prof. Fei Chiang. Throughout my Ph.D. study, Prof. Chiang guided me in developing research ideas, writing research papers, and giving academic presentations. I shall be indebted for everything I have learned from her on both academic levels and personal levels. As I pursue my academic career, Fei will always be my advisor, mentor, and dear friend.

I am deeply grateful to my thesis committee members, Prof. Ryszard Janicki and Prof. Wenbo He, for the discussions throughout my Ph.D. career and insightful comments on this thesis. I would like to thank Prof. Denilson Barbosa, for serving as my external reviewer and offering valuable suggestions. I would like to offer my special thanks to Prof. Yinghui Wu for providing guidance and feedback throughout our collaboration on research projects. I also would like to thank all the other collaborators I have worked with over the years: Prof. Lukasz Golab, Prof. Jaroslaw Szlichta, Prof. Mostafa Milani, Dr. Zheng Zheng, Dr. Yu Huang, Dr. Hanchao Ma, Zhi Qu, Adam Mansfield and many others. I would not have had a successful Ph.D. without them. Finally, this dissertation would not have been possible without my wife, Zahra, for her continuous support and endless love.

Preface

This "sandwich-based" thesis consists of previously published papers over the course of the dissertation. In particular, the work in Chapter 4 was done in collaboration with H. Ma and Y. Wu from Washington State University.

- In Section 4.5, H. Ma and Y. Wu designed the budgeted entity matching algorithm.
- In Section 4.4.2, M. Alipourlangouri contributed in design of the matching algorithm.
- In Section 4.6, M. Alipourlangouri and H.Ma contributed in the implementation and experimental evaluation of the algorithms.
- In the conference paper [83], H. Ma and Y. Wu contributed in the reasoning about OGKs, which has been removed from this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data quality is an important challenge in data management since real-life data is often dirty. Poor-quality data is often the source of inaccurate reports, which cause economic loss to many companies. Based on a report in 2017, IBM suggests the yearly cost of data quality issues in the U.S. during 2016 alone was about \$3.1 trillion [74]. Lack of trust by business managers in data quality is commonly cited among chief impediments to decision-making.

As real-life data is often dirty, there exists a need for data quality management. Data consistency and data deduplication are two important parts of this process [49, 28, 47]. Data consistency aims to ensure the same representation or value of an attribute across multiple occurrences or over time [47]. Data dependencies (*i.e.,* integrity constraints) are used to formalize this representation to preserve data consistency and accuracy. A data dependency, such as Functional Dependency (*i.e.,* FD), is a constraint that defines a relationship between attributes that is expected to hold over the data. Violations of these constraints (*i.e.,* inconsistencies) can occur due to wrong attribute values or value changes over time. Data deduplication, also known as *record linkage* or *record matching*, refers to the process of identifying data records in the same or different databases that refer to the same real-world entity. Data deduplication techniques usually require a comparison between each pair of data records, which is infeasible in a large dataset [73, 44]. However, keys provide information on how to uniquely identify entities in a database. They can be used for deduplication over one database or while integrating data from different sources [37].

Erroneous and duplicate data can lead to false conclusions, inaccurate business decisions making and ineffective marketing. By using integrity constraints such as FDs and keys, we can preserve data quality over a dataset. for example, FDs define relationships among a set of attribute values for a given entity and have been used to detect inconsistencies and repair data [54]

There has been an increasing demand recently for the graph data model in different applications including knowledge graphs [79, 84, 112], social networks [114], peer-to-peer networks [29], and e-commerce [41]. In the graph data model, entities are represented as nodes of a graph and there will be an edge between the nodes if there exists a relationship between their corresponding entities. However, unlike the relational data model, real-life graphs usually do not come with the schema.

Poor data quality continues to be a serious problem in graph data [57, 36, 49, 107].

To preserve data quality over graphs, Graph Functional Dependencies (*i.e.,* GFDs) have been defined by extending FDs from the traditional relational data model [57, 53]. GFDs impose a topological constraint in the form of a graph pattern and then impose a data constraint between the attribute values of the matches of the pattern. GFDs are being used as a main data dependency for *static* graphs. However, real-world graphs evolve, where node attribute values and edges are changing over time. Based on the application semantics, some changes are erroneous and some are valid changes. It is important to differentiate clean and anomalous values in order to preserve data quality on graphs over time. Similar to the relational data model [11], data dependencies (such as GFDs) can be extended to preserve data consistency over evolving (temporal) graphs.

Data deduplication is one of the most common data quality issues facing large organizations [42]. Keys serve a vital role to make a connection between a real-world entity and its representation in a database. Keys are a fundamental integrity constraint used in database systems to define the set of attributes that uniquely identify an entity in a graph [48]. While keys are often defined by a domain analyst according to application and domain requirements, manual specification of keys is expensive and laborious for large-scale datasets. Moreover, as the data naturally evolves over time, maintaining the keys becomes a real challenge. This highlights the need for automated solutions in order to discover keys over large graphs.

Data quality is known to be subjective and highly contextual based on individual preferences or domain specific definitions [14]. Existing constraints for graphs can be too restrictive to characterize structures and node labels that are syntactically different but semantically equivalent. For example, in a knowledge graph, we might have nodes with different types that are semantically similar and refer to the same real world entity *e.g.,* we note that labels *artist* and *band*, and types *hit* and *song* are semantically similar. Existing key constraints that impose label equality cannot capture such cases. The semantics are available as an ontology that provides domain specific concepts and relationships, defining semantic equivalence among node labels. Recent work has motivated the need to do contextual data cleaning rather than declarative approaches by integrating ontologies into data dependencies and cleaning decisions [14, 27, 122]. This highlights the need to integrate data semantics to enrich existing declarative dependencies over graphs, in particularly over keys.

1.1 Temporal Graph Functional Dependencies

Graphs have been widely used to model real-world entities and their relationships, such as knowledge graphs [79, 84, 112], e-commerce [41] and social networks [114]. Data constraints have been extended for graphs to capture inconsistencies and errors in graph data [57, 48, 83]. Given a graph G, a graph data dependency (GFD) is often in the form of $(Q, X \rightarrow Y)$, where Q is a graph pattern that specifies a set of subgraphs in G via graph pattern matching (*e.g.,* subgraph isomorphism), such that each subgraph should satisfy the value constraints enforced by $X \rightarrow Y$ (where X and Y are literals). Notable examples include graph functional dependencies [57], graph keys [48] and graph association rules [56, 113], among others.

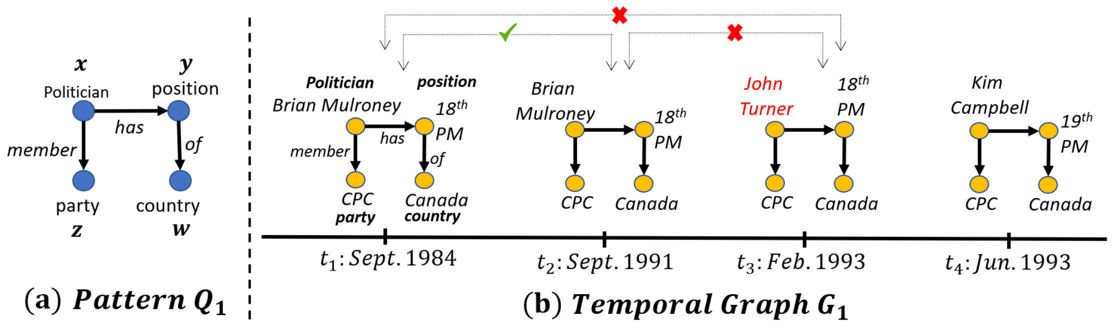


Figure 1.1: Temporal data constraints in real-world evolving graphs.

Nevertheless, existing graph dependencies are often designed for *static* graphs. Real-world graphs are evolving, where (1) both node attribute values and edges are changing over time; and (2) a data constraint may persist only over a fraction of "snapshots" of evolving graphs and for specific time intervals. While some changes naturally occur as the data evolves to model events and application semantics, other changes are erroneous given expected time intervals. Differentiating between clean and anomalous values is critical towards preserving data quality. The need to model such data constraints is prevalent to support decision making in government [4], education, and health care [3, 25]. Consider the following real-world examples.

Example 1: Figure 1.1(a) shows a graph pattern to illustrate data constraints in real graphs. Q_1 specifies the relationship between a politician, their position in the country, and their political party. While existing dependencies such as GFDs model topological constraints (such as Q_1), and dependencies among the attributes, enforcing temporal consistency of schema and attribute constraints are not captured. This can lead to undetected errors when compared to neighbouring matches and values. Politicians often have a maximum time duration for which they hold office, *e.g.,* normally 10 years for a Canadian Prime Minister. Similarly, in the US, to qualify as a state governor, candidates must be a resident of the state, and a citizen for a minimum number of years, *e.g.,* a minimum 5 year state residency, and 5 year citizenship are required in California. Figure 1.1(b) shows matches of Q_1 with *Brian Mulroney* as the 18^{th} *Prime Minister* of *Canada* from *Sept. 1984* (t_1) until *Feb. 1993* (t_3). However, an update happened at t_3 where we have *John Turner* as 18^{th} *Prime Minister* of *Canada*, which is an error. The error is only identified by comparing the match at t_3 with t_1 and t_2 for the 18^{th} *PM.* However, at t_4, *Kim Campbell* is elected as the first female and the 19^{th} Canadian Prime Minister. □

Achieving temporal consistency with respect to (*w.r.t.*) topology and attribute value relationships is pivotal in many areas as one example is highlighted above to ensure stability, adherence to policy, and efficacy. The "minimum" and "maximum" time bounds pose additional requirements where structural and literal conditions specified via graph pattern matching should hold. These constraints cannot be readily captured by prior graph dependencies [57, 53, 87]. GFDs impose schema and attribute constraints over a single snapshot involving a *single* match in a graph G. In contrast, to model temporal consistency, we must compare values between *pairs* of matches occurring at two timestamps in a temporal graph. This requires new temporal data constraints that model temporal consistency of graph patterns and attribute dependencies, and new algorithms that incrementally identify inconsistencies (errors) in the presence of change across the graph snapshots, and efficiently determine which pairs of matches to validate without enumerating all pairwise combinations.

In this thesis, we address the above challenges, and are interested in several questions. (1) How to formalize such temporal data constraints in dynamic graphs? (2) What is the hardness of the fundamental problems (satisfiability, implication, validation and axiomatization) for such temporal graph data constraints? (3) How to efficiently detect violations (errors) with these dependencies in evolving graphs?

1.2 Discovery of Keys for Graphs

Keys are one of the fundamental integrity constraints used in database systems, which define the set of attributes that uniquely identify an entity. Keys serve a vital role in databases to maintain data quality standards by preventing incorrect insertions and updates as the data naturally evolves over time. In addition, keys provide clues for duplicate detection (also referred to as entity resolution), one of the most common data quality issues [42].

While keys are often defined by a domain analyst according to application and domain requirements, manual specification of keys is expensive and laborious for large-scale datasets. Existing techniques have explored mining for keys in relational data (as part of functional dependency discovery) [72], and in XML data [34].

The proliferation of graph databases has lead to the study of integrity constraints over graphs, including functional dependencies [53], and keys [48]. These constraints have shown to have wide applications to deduplication, citation of digital objects, knowledge fusion, and knowledge base expansion [42]. Evolving graphs such as knowledge bases and citation graphs require keys to uniquely identify objects to ensure reliable and accurate deduplication and query answering. In these dynamic settings, where object properties change frequently and new objects are added, manual specification of keys is expensive and labor intensive. Automated solutions are needed to discover keys. Although recent work has proposed techniques to find keys over RDF data [23], these techniques are not applicable for graphs as they do not support: (i) topological constraints; and (ii) recursive keys (a distinct feature in graph keys). To the best of our knowledge, there is no existing work that discovers keys for directed graphs. Consider the following example where keys help to identify entities in a database.

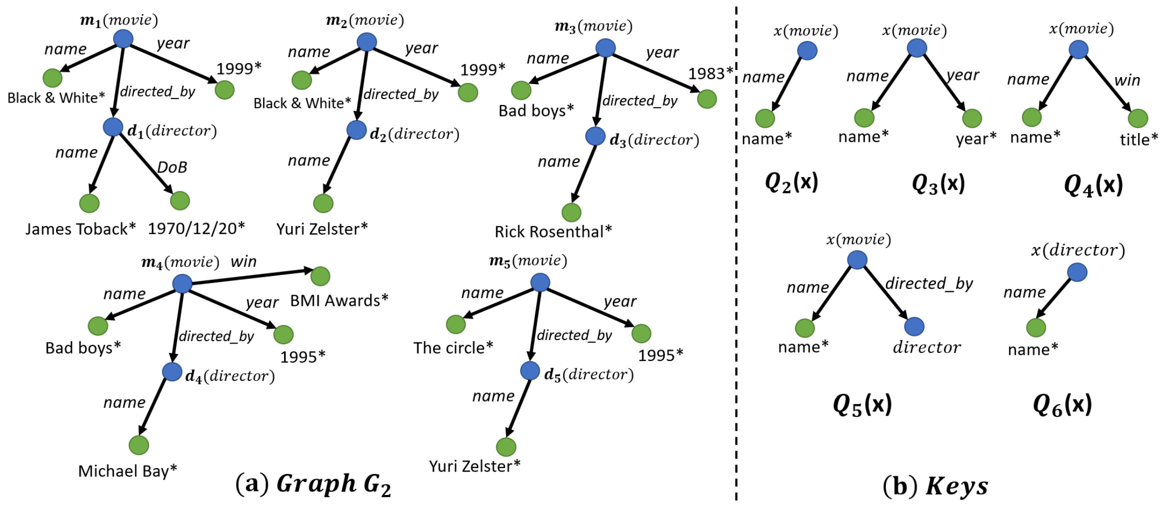


Figure 1.2: (a) Movie instances from the IMDb dataset. (b) Possible keys for *movie* and *director*

Example 2: Figure 1.2(a) shows a sample of five movies from the IMDb movie graph database [5]. In this figure, each movie is represented by a node which is associated with a type (*e.g., movie*). For each movie, there exist a set of properties which are represented by nodes and have a string value for the property and there exists an edge between the movie and the property. The label of the edge shows the property name. For example, for the movie m_1, we have two properties *name* and *year* which are shown by two edges with the same label from m_1 to two nodes with the value of *Black & White* and *1999* resp. Intuitively, a key can be defined by a topological pattern, and a set of edge labels, and node types [48]. We can define a key for a type of entity using a graph pattern, and apply pattern matching algorithms to identify unique entities in a graph. Figure 1.2(b) shows a sample of possible keys to uniquely identify a movie:

$Q_2(x)$: By the movie name (title of the movie).

$Q_3(x)$: By movie name, and year of release.

$Q_4(x)$: By movie name, and awards won.

$Q_5(x)$: By movie name, and a specific director.

$Q_6(x)$: A director can be uniquely identified by his/her name.

This example highlights that many keys are possible, and this often depends on the data and its semantics. According to $Q_2(x)$, all movies with the same name resolve to the same movie. This of course does not hold true over time, as Figure 1.2(a) shows two different movies titled *Bad Boys* in *1983* (m_3) and in *1995* (m_4). Secondly, the domain semantics influence the quality of a key. For example, $Q_4(x)$ indicates a movie is uniquely identified by its name and awards won. However, not all movies win awards, and the second condition will lead to null values for many movies (that have not won awards), thereby leading to poor support and representation across all movies. $\square$

The example demonstrates the need for automatic discovery of keys over large scale graphs as there are many possible keys for each entity type. In this thesis, we address the above challenges, and are interested in several questions. (1) What properties define a meaningful key in a graph? (2) How can we efficiently discover such keys?

1.3 Ontological Keys for Graphs

Existing keys for graphs use syntactic equivalence or similarity of labels to identify entities. However, entities in real-world graphs often contain multiple attributes and different labels. These labels could be syntactically different, but semantically similar given a domain specific relationship. This poses two challenges for entity matching over graphs: (1) nodes that should refer to the same entity may not be captured by key constraints that only enforce label equality [88]; and (2) nodes with equal labels that match key patterns may not necessarily refer to the same entity, due to differing attributes. Furthermore, such graphs are often interpreted *w.r.t.* an ontology that provide domain specific concepts and relationships, defining semantic equivalence among node labels. Consider the following example.

Example 3: Consider a knowledge graph G consisting of triples (subject, predicate, object) where subject and object are nodes, and predicate is an edge connecting subject to object. Figure 1.3(a) illustrates a fraction of IMDb dataset G, with two subgraphs describing two movie entities $\{m_6, m_7\}$, where each node has an associated

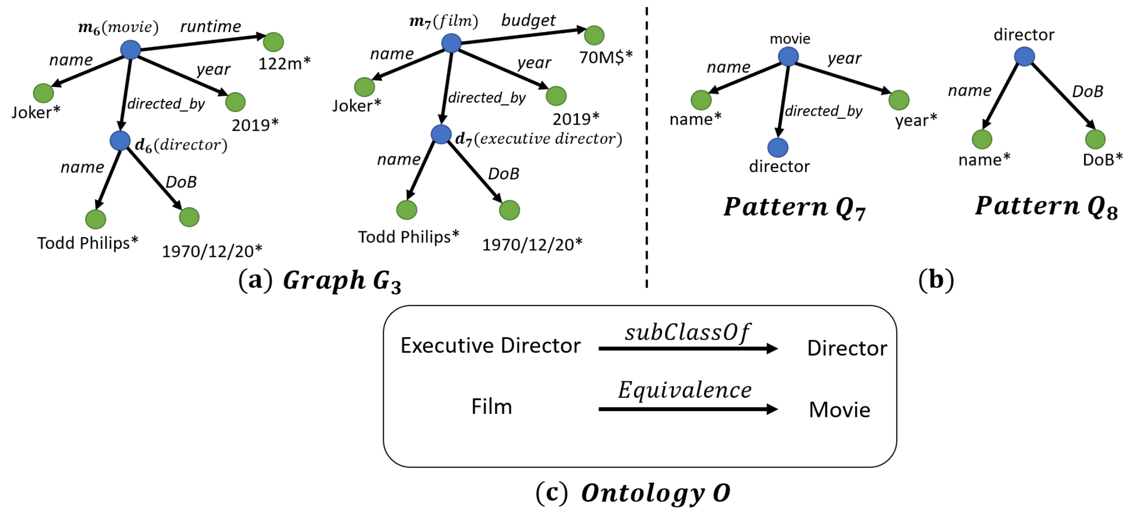


Figure 1.3: Entity matching with ontologies.

type denoted in parentheses. For example, entity m_6 is of type *movie* and m_7 has the type *film*. Consider graph keys Q_7 and Q_8 depicted as graph patterns in Figure 1.3(b). Q_7 states that "*if two movies share the same name, year and director, then they refer to the same movie*". Similarly, a director can be identified by its name and date of birth (DoB), characterized by Q_7. Note the dependence of Q_7 on Q_8, to identify a movie, we need to first identify its director, reflecting the recursive property of graph keys [48]. Applying Q_7 and Q_8 via subgraph isomorphism on G, we obtain only m_6 and d_6 as matches, since (1) m_7 is of type *film* (rather than *movie*); (2) d_7 is of type *executive director* rather than *director*. Hence, Q_7 and Q_8 fail to identify m_6 and m_7 as the same movie as they rely only on label matching.

Given an ontology O, as shown in Figure 1.3(c), we exploit ontological relationships and semantic equivalance to extend graph keys. For example, we recognize an *executive director* is a type of *director* participating in a hyponym (subClassOf) relationship. Similarly, we note that labels *film* and *movie* are semantically similar.

By extending Q_7 and Q_8 with these ontological equivalences, we identify m_6 and m_7 are indeed the same song. □

The above example highlights the need for a new class of dependencies for graphs that go beyond existing subgraph isomorphism to consider entity matching with ontological similarity. We extend existing notions of keys for graphs, which uniquely identify an entity, to exploit the relationships among node labels given in an ontology. The recursive property of graph keys allows us to precisely define related entities for the keys. By incorporating ontologies, we further increase the scope of entities that can be matched to recursive keys to include matches that are ontologically similar. While ontological extensions have been studied for traditional functional dependencies [35, 27], little work has been done to enrich graph keys with ontologies.

In this thesis, we would like to address the above challenges and address the following questions. (1) How to extend graph keys by exploiting ontologies to include data semantics to increase the expressiveness of the keys? (2) How to effectively detect duplicates using ontological matching over graphs?

1.4 Contributions

In this thesis, we address three problems:

1. Problem 1: Existing dependencies do not cover consistency over time intervals.

Given a temporal graph, how can we enforce topological and attribute value constraints over a duration of time? How to efficiently detect violations of these constraints?

2. Problem 2: Given a graph, we study the problem of discovering keys for graphs. We define new properties for graph keys in the context of the discovery problem and propose an efficient algorithm to mine keys based on these properties.

3. Problem 3: To help data deduplication over graphs using contextual-based data cleaning, we study the problem of extending graph keys by an ontology to increase the expressiveness of graph keys. Moreover, we study the problem of ontology-based entity matching in attributed graphs to detect duplicate entities.

We make the following contributions:

1. In Chapter 2, we introduce a new class of dependencies, called *Temporal Graph Functional Dependencies*(TGFDs) that specify topological and attribute requirements over temporal graphs induced by time intervals. We discuss the relationship between TGFDs and existing graph dependencies. We study the satisfiablity, implication, and validation problems for TGFDs, and show their complexity is no harder than their non-temporal counterparts. We introduce two TGFD error detection algorithms: (a) an incremental algorithm (IncTED) that re-uses the matches and errors from earlier graph snapshots; and (b) a parallel algorithm (ParallelTED) that distributes the matching and error detection task

among a set of nodes to minimize the runtime. We conduct an extensive evaluation over real data collections showing the efficiency of our algorithms. This work is under review for ICDE 2023 [18].

2. In Chapter 3, we propose an efficient algorithm to mine keys for graphs. We define meaningful properties for graph keys and formalize the discovery problem of graph keys. We develop an algorithm called GKMiner to mine graph keys efficiently based on the defined properties. We show that our method is scalable and it is feasible to mine meaningful keys in graphs of millions nodes and edges. The preliminary paper of this work is published at a VLDB 2018 workshop called Advances in Mining Large-Scale Time Dependent Graphs (TD-LSG) [16] and the short paper is to appear at DaWaK 2022.

3. In Chapter 4, we extend keys for graphs with ontological pattern matching. We propose *Ontological Graph Keys* (OGKs), a new class of key constraints that exploit ontologies to enhance keys for graphs. We formally introduce the entity matching problem using OGKs by revising the Chase process of conventional data dependencies for OGKs. We experimentally verify the efficiency and effectiveness of our OGK entity matching algorithms using two real-world graphs. We compare against two existing baselines, demonstrate the improved efficiency of our techniques, and our ability to identify semantically equivalent entities that are ignored by existing solutions. This work was published on VLDB 2019 [83].

1.5 Thesis Outline

This thesis is a sandwich thesis consisting of the conference and journal publications [18, 83, 16]. In Chapter 2, we present our work on temporal graph functional dependencies. In Chapter 3, we present our technique to automatically discover graph keys over large graphs. In Chapter 4, we introduce our model to extend graph keys by ontologies to define OGKs. Finally, in Chapter 5, we conclude the thesis with final remarks and directions for future works.

Chapter 2

Temporal Graph Functional Dependencies

2.1 Introduction

Data constraints have been extended for graphs to capture inconsistencies and errors in graph data [57, 48, 83]. Given a graph G, a graph data dependency is often of the form of $(Q, X \rightarrow Y)$, where Q is a graph pattern that specifies a set of subgraphs in G via graph pattern matching (*e.g.,* subgraph isomorphism) such that each subgraph should satisfy the value constraints enforced by $X \rightarrow Y$ (where X and Y are literals from the graph pattern). Notable examples include graph functional dependencies (GFDs) [57], graph keys (GKeys) [48] and graph association rules (GTARs) [56, 113].

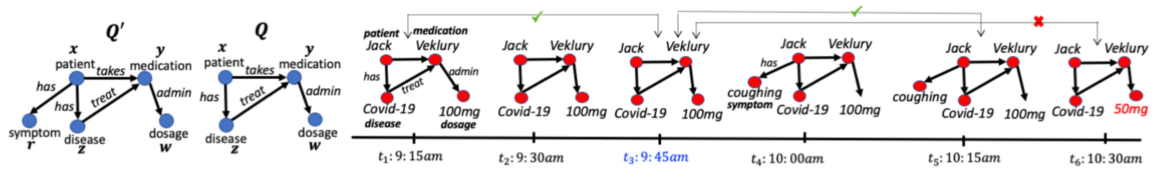


Figure 2.1: Temporal data constraint to validate drug dosage at t_3.

Existing graph dependencies are often designed to capture inconsistencies in static graphs. Nevertheless, data errors also occur in *evolving* real-world graphs. In such scenarios, node attribute values and edges in graphs experience constant changes over time, and data consistency may persist only over a fraction of graphs specified by certain time periods. The need for modeling data consistency in temporal graphs that are time-dependent is evident in time sensitive applications such as policy-making [4] and anomaly detection in health care [3, 25]. Data constraints for static graphs are often insufficient to satisfy such needs, as illustrated next.

Example 4: Figure 2.1 illustrates a temporal graph that monitors medicare activities [25]. A snapshot denotes patient activity describing their diagnosis, medication, and dosage taken at a timestamp. Federal drug regulating agencies, such as the Food and Drug Administration, receive numerous medication error reports due to missed doses, incorrect preparation and administration of drug formulations. To counter this, safe practice recommendations state that administered dosages are verified against patient characteristics, and require that past and subsequent doses are correctly timed [7].

Consider a graph pattern Q that specifies patients who are treated with a specific medication *Veklury*, and a dosage that is administered within a time interval. If the intravenous (IV) infusion is between 30 to 120 minutes, then the dosage must be 100mg [3]. Figure 2.1 shows that continual validation of the dosage at the current time $t_c = t_3$ compares to past drug administrations (at least 30 but no more than 120 minutes away) that must be 100mg. An incorrect dosage of 50mg is captured at t_6. This time-sensitive rule can be expressed by posing a value constraint on the dosage to the patients who match Q, *conditioned* by a time infusion period. To capture violations requires comparing a fraction of "snapshots" induced by a time period. Existing data constraints cannot express such consistency criteria for temporal graphs. □

The above example calls for the need to model data consistency *w.r.t.* topological constraints and attribute values that are contextualized over a time interval duration. In such cases, for each "current time" t_c, there is a need to compare matches at t_c against historical and future matches that are at least p but no more than q time units away. These "minimum" and "maximum" time bounds pose additional requirements where structural and literal conditions specified via graph pattern matching should hold. Such schema and temporal constraints are prevalent in real data.

These constraints cannot be readily captured by prior graph dependencies and rules [57, 53, 87]. For example, GFDs cannot express the temporal constraints that perform necessary pairwise comparisons of single snapshots induced by minimum and maximum time ranges. Even if time periods were captured via graph attributes, modeling such semantics would involve an excessive number of GFDs, making them infeasible to be verified in practice. GTARs detects delayed co-occurrences of events via graph pattern matching, and do not model value dependencies [87]. We address the above challenges, and study several questions. (1) How to formalize such temporal data constraints in dynamic graphs? (2) What is the hardness of the fundamental problems (satisfiability, implication, validation) for these temporal data constraints? (3) How to efficiently detect violations with these dependencies in evolving graphs?

Contributions. This chapter introduces Temporal Graph Functional Dependencies (TGFDs), a class of data dependencies to model the time-dependent consistency of temporal graph data, and studies its fundamental problems as well as TGFD-based inconsistency detection.

(1) We introduce a formal model of TGFDs. TGFDs model time-dependent data consistency of temporal graphs by enforcing value dependencies that are conditioned by topological and temporal constraints in terms of temporal graph pattern matching. TGFDs apply conditionally to temporal graphs, and hold on graph snapshots that are induced by time intervals with lower and upper bounds. A special case of TGFDs with size-bounded graph patterns and their benefit in capturing data errors has been justified by our pilot study [89]. Providing a formal model for general TGFDs is our first contribution.

(2) We study the satisfiablity, implication, and validation problems for TGFDs. We introduce an implication algorithm for TGFDs, and the notion of *temporal closure* that considers the time intervals of inferred values. We also develop a sound and

complete TGFD axiom system for the implication of TGFDs (Section 2.3).

(3) We introduce two TGFD-based inconsistency detection algorithms for temporal graphs: (i) an incremental detection algorithm (IncTED) that re-uses the matches and inconsistencies captured from earlier graph snapshots to compute updated matches and errors given graph changes; and (ii) a parallel algorithm (ParallelTED) that performs fine-grained estimations of the workload by computing the fraction of data induced by the literal conditions and the time interval duration in a TGFD. We show that these algorithms have time costs that are independent of the size of temporal graphs, and are feasible for large graphs (Section 2.4).

(4) We conduct an extensive evaluation over real data collections. We verify the efficiency of ParallelTED over a wide range of parameters achieving 120% and 29% speedup over sequential and GFD-based baselines, respectively, demonstrating TGFD error detection is feasible over real graphs. We show the effectiveness of ParallelTED over error detection using GFDs and GTARs with up to 55% and 74% gain in $\mathsf{F_1}$-score, respectively. Lastly, we conduct a case study with real examples of TGFDs, and the detected inconsistencies to demonstrate the utility of TGFDs in practice (Section 2.5).

2.2 Temporal Graph Functional Dependencies

We start with a notion of temporal graph pattern matching, and then introduce TGFD syntax and semantics.

2.2.1 Temporal Graph Pattern Matching

Temporal Graphs. We consider a temporal graph $\mathcal{G}_T = \{G_1, \ldots, G_T\}$ as a sequence of graph snapshots. Each snapshot G_t ($t \in [1, T]$) is a graph (V, E_t, L_t, F_{A_t}) with a fixed node set V and edge set $E_t \subseteq V \times V$. Each node $v \in V$ (resp. edge $e \in E_t$) has a label $L_t(v)$ (resp. $L_t(e)$) at time t. For each node v, $F_{A_t}(v)$ is a tuple $(A_{1,t} = a_1, \ldots, A_{n,t} = a_n)$ specifying a value of each attribute of v at time t.

Temporal graph pattern matching. A graph pattern is a directed, connected graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$ with a set of nodes V_Q and edges $E_Q \subseteq V_Q \times V_Q$. For each node $u \in V_Q$ and each edge $e \in E_Q$, the function L_Q assigns a label $L_Q(u)$ and $L_Q(e)$ to u and e, respectively. $\bar{x}$ is a set of variables, and μ is a function that maps each $u \in V_Q$ to a distinct variable in $\bar{x}$. We shall refer to $\mu(u)$ as u for simplicity.

Matches. Given a temporal graph $\mathcal{G}_T$ and pattern Q, a *match* $h_t(\bar{x})$ between a snapshot G_t of $\mathcal{G}_T$ and pattern Q is a subgraph $G'_t = (V'_t, E'_t, L'_t, F'_{A_t})$ induced by V'_t of G_t that is isomorphic to Q. That is, there exists a bijective function (a matching)

Table 2.1: Notations and symbols.

Symbol	Description
$\mathcal{G}_T$, G_t, G'_t,	temporal graph, snapshot, and subgraph
$Q[\bar{x}]$	graph pattern
σ, Σ	a single TGFD, and a set of TGFDs
h_t	a match of $Q[\bar{x}]$ in G_t at time t
$\mathcal{E}(\mathcal{G}_T\ ,\sigma)$, $\mathcal{E}(\mathcal{G}_T\ ,\Sigma)$	violations of σ, Σ in $\mathcal{G}_T$
$\mathcal{M}(Q, G_t)$	match set of Q in G_t
I_{h_i}, I_{h_j}	time domain of h_i, h_j
$\rho(i)$	permissable range of h_i
Π_X, π_{XY}	matches with shared values in X, XY
$\Gamma(\Pi_X)$	timestamps of matches in Π_X
$\mu_X(h_i)$	returns π_{XY} where h_i belongs

h_t from V_Q to V'_t such that: (1) for each node $u \in V_Q$, $L_Q(u) = L'_t(h_t(u))$; and (2) for each edge $e = (u, u') \in Q$, there is an edge $e' = (h_t(u), h_t(u'))$ in G'_t such that $L_Q(e) = L'_t(e')$. If $L_Q(u)$ is '_', then it matches with any label for any timestamp t.

As a matching h_t uniquely determines a match (subgraph) for $\bar{x}$ at any time t, we refer to a match as h_t for simplicity.

2.2.2 Temporal Graph Functional Dependencies

Syntax. A *Temporal Graph Functional Dependency* (TGFD) σ is a triple $(Q[\bar{x}], \Delta, X \to Y)$, where:

- $Q[\bar{x}]$ is a general graph pattern (*e.g.,* may include cycles, be a tree, DAG);

- $\Delta = (p, q)$ is a time interval, specified by a lower bound p, and an upper bound q (p, q are integers and $q \geq p \geq 0$);

- a value dependency $X \rightarrow Y$, where X and Y are two (possibly empty) sets of literals defined on $\bar{x}$.

We consider literals of the form $u.A = c$ (a constant literal) or $u.A = u'.A'$ (a variable literal), where $u \in \bar{x}$, A and A' are attributes, and c is a constant.

A TGFD σ enforces three constraints:

1. topological and label constraint of pattern Q,
2. a value dependency specified by $X \rightarrow Y$, and
3. a time interval constraint Δ, which specifies, for a time t, two time windows $[t-q, t-p]$ and $[t+p, t+q]$. The time windows induce the set of snapshots over which the constraint should hold (see "Semantics").

For simplicity, we consider *w.l.o.g.* TGFDs in a normal form, *i.e.,* with a dependency $X \rightarrow Y$ where Y contains a single literal. We justify the normal form in Section 2.3.3.

Example 5: We define a TGFD $\sigma = (Q[\bar{x}],\ \Delta = (30, 120),\ [x.name, z.name = Covid19, y.name = Veklury] \rightarrow [w.\mathsf{val} = 100mg])$. If a patient has *Covid-19* and is treated with *Veklury* over IV infused between 30 to 120 mins, then the dosage is *100mg*. $\square$

Semantics. Given a temporal graph $\mathcal{G}_T$, a TGFD enforces value dependencies between any pair of snapshots that are (1) matches via graph pattern matching, and (2) having timestamps within time ranges specified by Δ, for any timestamp in $[1, T]$.

Consider a TGFD $\sigma = (Q[\bar{x}], \Delta, X \to Y)$, and a pair of matches (h_i, h_j) of Q in G_i and G_j, respectively ($i, j \in [1, T]$). We say (h_i, h_j) *matches* σ if

- (h_i, h_j) satisfies X (denoted as $(h_i, h_j) \models X$), that is, for each constant literal l $= (u.A = c)$ (resp. variable literal $u.A = u'.A'$) in X, $h_i(u) = h_j(u) = c$ (resp. $h_i(u) = h_j(u')$); and
- $|j - i| \in \Delta$.

The term $|j - i| \in \Delta$ conveniently express temporal semantics including "future" and "past" from their conventional counterparts in temporal integrity constraints [116] to temporal graphs. For any specific timestamp $i \in [1, T]$, (a) if $i \leq j$, then $|j - i| \in \Delta$ specifies any timestamp j from a "future" time window $[i + p, i + q]$; (b) if $i > j$, then $|j - i| \in \Delta$ specifies timestamps j from a "past" time window $[i - q, i - p]$. Given a time interval Δ, a TGFD σ enforces data consistency over all pairs $(h_i, h_j) \in [1, T]$ that matches σ in terms of Δ.

For any pair (h_i, h_j) that does not match σ, (h_i, h_j) "trivially" satisfies σ. We say a temporal graph $\mathcal{G}_T$ *nontrivially satisfies* a TGFD σ, denoted as $\mathcal{G}_T \models \sigma$, if (a) there exists at least a pair of matches (h_i, h_j) that also matches σ, and (b) $(h_i, h_j) \models Y$. $\mathcal{G}_T$ satisfies a set of TGFDs Σ if for every $\sigma \in \Sigma$, $\mathcal{G}_T \models \sigma$.

Example 6: Figure 2.1 shows matches (h_1, h_3) and (h_3, h_5) satisfy σ (denoted with a green check mark), but $(h_3, h_6) \not\models \sigma$, having the wrong dosage of *50mg* at t_6 (denoted with a red x) given the required infusion time of 30 to 120 minutes. $\square$

Remarks. As justified in our pilot study [89] on a special case of the general TGFDs with bounded pattern size over a real knowledge graph DBpedia, we found over 140 temporal data constraints in real knowledge base DBpedia. Of these constraints, 28% can capture at least one inconsistency; and each constraint can capture on average 7 distinct erroneous attribute values that span over 5 timestamps.

Relationship to Other Dependencies. GFDs define topological and attribute dependence over *static* graphs without temporal semantics, involving a single match [57]. As expected, TGFDs subsume GFDs as a special case when $\Delta = (0, 0)$. Graph keys (GKeys), and their ontological variant [48, 83] specify topological and value constraints to enforce node equivalence but do not consider attribute dependencies. Graph Entity Dependencies (GEDs) extend GFDs to support equality of entity ids, and subsume GFDs, but again, are defined over static graphs. GTARs are soft rules that use an approximate subgraph isomorphism matching, in contrast to TGFDs which are hard constraints enforcing strict subgraph isomorphism matching based on Q. In addition, GTARs only consider matching with time intervals $p = 0$, and do not include historical matches as part of their temporal semantics.

2.3 Foundations

We next study the satisfiability, implication and validation problems for TGFDs, and present an axiomatization.

2.3.1 Satisfiability

A set Σ of TGFDs is *satisfiable*, if there exists a temporal graph $\mathcal{G}_T$, such that $\mathcal{G}_T \models \Sigma$. The satisfiability problem is to determine whether Σ is satisfiable. Satisfiability checking helps to decide whether a set of TGFDs Σ are "inconsistent" before being applied for error detection in temporal graphs.

Example 7: Consider the pattern Q' of Figure 2.1 and a TGFD $\sigma' = (Q'[\bar{x}], \Delta = (30, 120), [x.name, r.name, z.name = Covid19, y.name = Veklury] \rightarrow [w.\mathsf{val} = 20mL])$, with $Q \subseteq Q'$. The consequent literal requires the value of w to be equal to *100mg* and *20mL* simultaneously, leading to "conflicting" value constraints. Since any match of σ will also match σ', there is no temporal graph $\mathcal{G}_T$ that satisfies both. Thus, $\{\sigma, \sigma'\}$ are not satisfiable. □

The "conflicting" value constraints do not necessarily lead to unsatisfiable TGFDs. In the above example, suppose the Δ time interval durations for σ, σ' were $(30, 120)$ and $(20, 25)$, respectively, then one can verify that they do become satisfiable as the time intervals are not overlapping. This requires computing the time intervals of when derived values and literals are expected to hold. This illustrates that TGFD satisfiability analysis is more involved than GFDs: the "conflicting" value constraints are conditioned upon both *pattern matching and the temporal constraints.*

To characterize this, we introduce a notion of TGFDs *embeddings*. The notion has its foundation in [57], and is extended for TGFDs with temporal constraints.

Pattern embedding [57]. We say a graph pattern $Q'[\bar{x}'] = (V'_Q, E'_Q, L'_Q, \mu')$ is *embeddable* in another pattern $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, if there exists a mapping f from V'_Q to a subset of nodes in V_Q that preserves node labels of V'_Q, all the edges induced by V'_Q and corresponding edge labels. Moreover, f induces a "renaming" of each variable $x \in \bar{x}$ to a distinct variable x' in $\bar{x}'$, *i.e.,* for each variable $x \in \bar{x}$, $\mu'(f(\mu^{-1}(x))) = x' \in \bar{x}'$.

TGFDs Embedding. Given two TGFDs $\sigma = (Q[\bar{x}],\ \Delta,\ f(X') \rightarrow f(Y'))$ and $\sigma' = (Q'[\bar{x}], \Delta', X' \rightarrow Y')$, We say that σ is a (temporally) *overlapped* TGFD of σ' *w.r.t.* graph pattern Q, if (1) Q' is embeddable in Q, and (2) $(\Delta \cap \Delta') \neq \emptyset$. Moreover, σ is a (temporally) *embedded* TGFD of σ' *w.r.t.* Q if $\Delta' \subseteq \Delta$. Given a set of TGFDs Σ and a graph pattern Q, we denote as Σ_Q (resp. $\Sigma_{Q,\{\Delta,\Delta'\}}$) the set of embedded (resp. overlapped TGFDs) *w.r.t.* Q.

Example 8: In Figure 2.1, Q is an embedded pattern in Q' as there exists a subgraph isomorphism mapping from Q to a subgraph of Q'. Moreover, consider a TGFD σ'' with $\Delta'' = (20, 60)$, then σ'' is a (temporally) overlapped TGFD of σ. Lastly, σ is a (temporally) embedded TGFD of σ'. □

In contrast to GFDs, TGFD satisfiability requires checking whether conflicting literal values occur during overlapping time intervals, leading to non-satisfiability. However, if differing attribute values occur during non-overlapping intervals for a set of TGFDs, then each unique literal value is expected to hold over a distinct time interval, independent of time intervals from other TGFDs. This requires us to consider the notion of *temporal closure* for TGFDs, introduced next.

Definition 3.1: (*Temporal Closure*) Given a set of overlapped TGFDs $\Sigma_{Q,\{\Delta,\Delta'\}}$ *w.r.t.* a graph pattern Q, the *temporal closure* of a set $\Sigma_{Q,\{\Delta,\Delta'\}}$, denoted as *closure*$(X, \Sigma_{Q,\{\Delta,\Delta'\}})$, refers to the set of literals that are derivable via transitivity of equality atoms in X over $\Sigma_{Q,\{\Delta,\Delta'\}}$. The temporal closure of a set of TGFDs Σ (denoted as *closure(*Σ)), refers to all the literals *closure*$(X, \Sigma_{Q,\{\Delta,\Delta'\}})$ with Q ranging over the patterns from the TGFDs in Σ. □

We outline an algorithm below to compute closure(Σ). First, given a set of TGFDs Σ, it first adds all Y seen in a TGFD $\sigma \in \Sigma$, if $(Q[\bar{x}], \delta, \emptyset \rightarrow Y) \in \Sigma$. Second, for each pattern Q seen in Σ, it then computes the overlapped TGFDs $\Sigma_{Q,\{\Delta,\Delta'\}}$. For each literal Y seen in $X \rightarrow Y$ in a TGFD $\sigma \in \Sigma_{Q,\{\Delta,\Delta'\}}$, if $X \subseteq$ closure(Σ) or can be derived via transitivity of equality atoms in closure(Σ), then it adds Y to closure(Σ). For example, if $x.A = u$ and $y.B = u$ are in closure(Σ), then $x.A = y.B$ can be derived and added to closure(Σ). This will give us the set of literals that are to be enforced in $\Sigma_{Q,\{\Delta,\Delta'\}}$.

For each derived value, we must compute the time intervals over which each literal value is expected to hold. We say *closure*($X, \Sigma_{Q,\{\Delta,\Delta'\}}$) is *in conflict*, if *closure*($X, \Sigma_{Q,\{\Delta,\Delta'\}}$) contains literals $x.A = a$ and $y.B = b$, and $a \neq b$. Given a set of TGFDs Σ, the temporal closure $closure(\Sigma)$ is in conflict, if there exists a graph pattern Q in a TGFD from Σ such that $closure(X, \Sigma_{Q,\{\Delta,\Delta'\}})$ is in conflict. We show the following result.

Lemma 1: *A set Σ of* TGFDs *is not satisfiable if and only if closure(Σ) is in conflict.* □

The result below verifies that checking TGFD satisfiability is not harder than their GFD counterparts. For brevity, all proofs in this section are in the extended version [18].

Theorem 1: *The satisfiability problem for* TGFDs *is* coNP*-complete.* □

Proof sketch: We introduce an NP algorithm for checking the complement of TGFD satisfiability, which is to decide if a given set of TGFDs Σ is not satisfiable. Based on Lemma 1, the algorithm guesses (1) a graph pattern Q seen in Σ, (2) a subset $\Sigma' \subseteq \Sigma$, (3) a subgraph isomorphism mapping from each pattern Q' seen in Σ' to Q. It then verifies if Σ' is a set of overlapped TGFDs *w.r.t.* Q, by verifying, for each pair of TGFDs $\sigma_1, \sigma_2 \in \Sigma'$ with time intervals Δ_1 and Δ_2, (a) whether their respective patterns Q_1, Q_2 are isomorphic to a subgraph in Q, and (b) whether the time intervals $(\Delta_1 \cap \Delta_2) \neq \emptyset$. If so, It then invokes the aforementioned algorithm to check if the temporal closure *closure*$(X, \Sigma_{Q,\{\Delta,\Delta'\}})$ is in conflict. If so, it returns "yes" (Σ is not satisfiable). The above process is in NP given bounded number of pairwise verification and conflict checking in PTIME.

The lower bound is verified by a reduction from GFD satisfiability with constant literals and DAG pattern only [57]. The lower bound is verified by reduction from subgraph isomorphism to the complement of the satisfiability problem. A TGFD counterpart can be constructed by introducing, for each GFD, a time interval $[0, 0]$, and a temporal graph with a single snapshot. As GFD satisfiability with constant literals is already coNP-hard, the lower bound follows. $\square$

2.3.2 Implication

Given a set of TGFDs Σ and a $\sigma \notin \Sigma$, we say Σ implies σ, denoted $\Sigma \models \sigma$, if for any $\mathcal{G}_T$, if $\mathcal{G}_T \models \Sigma$, then $\mathcal{G}_T \models \sigma$. If $\Sigma \models \sigma$, then σ is a logical consequence of Σ. Given Σ and σ, the implication problem is to determine whether $\Sigma \models \sigma$. Implication analysis helps us to remove redundant TGFDs and perform error detection with a smaller number of rules.

To check whether $\Sigma \models \sigma$, we extend the temporal closure in Section 2.3.1 to a counterpart for embedded TGFD set Σ_Q (denoted as *closure(X, Σ_Q)*). The temporal closure *closure(X, Σ_Q)* is a set of pairs (Y, Δ), where Y is a literal, and Δ is an associated time interval called the *validity period* in which Y should hold. We outline a new algorithm that computes *closure*(X, Σ_Q). In contrast to GFD implication, we must compute the sub-intervals on which derived literals are expected to hold after applying each TGFD. The increased space of literals is managed by indexing literals, and searching for overlapping time intervals at each apply step.

(i) Initialize: for each literal $x \in X$, add (x, Δ) to *closure*(X, Σ_Q).

(ii) Apply: for each $\sigma' \in \Sigma_Q$, where $\sigma' = (Q'[\bar{x}],\ \Delta',\ X' \to Y')$, if all literals $x' \in X'$ can be derived via transitivity of equality of values from the literals in *closure*(X, Σ_Q), then add $(Y', \Delta' \cap \Delta'')$ to *closure*(X, Σ_Q).

(iii) Merge: for a literal Y'' and all $\{(Y'', \Delta''_1), \ldots, (Y'', \Delta''_m)\}$ in the *closure*(X, Σ_Q), merge the time intervals and replace the pairs with a single pair $(Y'', (\Delta''_1 \cup \ldots \cup \Delta''_m))$.

The above process is in PTIME. We say a literal Y is *deducible* from Σ and X, if there exists a Σ_Q derived from Σ, and a pair $(Y, \Delta'') \in$ *closure*(X, Σ_Q) such that $\Delta \subseteq \Delta''$.

Lemma 2: *Given a set of* TGFDs Σ *and a* TGFD $\sigma = (Q(\bar{x}), \Delta, X \to Y)$, $\Sigma \models \sigma$ *if and only if* Y *is deducible from* Σ *and* X. □

Theorem 2: *The implication problem for* TGFDs *is* NP*-complete.* □

Proof sketch: We provide an NP algorithm that, given Σ and $\sigma = (Q(\bar{x}), \Delta, X \to Y)$, guesses a subset Σ' of Σ, and a mapping from the patterns of each TGFD in Σ' to the pattern of σ, to verify if Σ' is the embeddable set of TGFDs in Σ_Q. If so, it invokes the aforementioned procedure to compute the temporal closure of Σ_Q, and verify if Y is deducible. Specifically, it checks the validity period of the enforced literals during the Apply step, i.e., checking whether $(\Delta' \cap \Delta'') \neq \emptyset$ for time intervals Δ', Δ'' from $\sigma' \in \Sigma_Q$, and literal (Y'', Δ''), respectively. The verification is in PTIME for a finite number of literals in the closure and $|\Sigma_Q|$. For the lower bound, the implication of TGFD is NP-hard by reduction from the implication of GFDs, which is known to be NP-complete [57]. For a GFD with constant literal and DAG pattern only, the lower bound is verified by reduction from a variant of subgraph isomorphism, which is shown NP-complete. Similar to the implication, a TGFD counterpart can be constructed by having a set of GFDs, where each GFD has a time interval $[0, 0]$, and a temporal graph with a single snapshot. $\square$

2.3.3 Axiomatization

We present an axiomatization for TGFDs. The first five axioms also apply to GFDs (no axioms were defined for GFDs [57]). Our axiomatization is sound and complete [18].

Axiom 1: *(Literal Reflexivity) For a given $Q[\bar{x}]$, X, Y are sets of literals, if $Y \subseteq X$, then $X \rightarrow Y$.* $\square$

In Axiom 1, a set of literals Y that is a subset of literals X, will induce a trivial dependency $X \rightarrow Y$ for any Δ.

Axiom 2: *(Literal Augmentation) If* $\sigma' = (Q[\bar{x}], \Delta, X' \rightarrow Y)$*,* $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$*, and* $X' \subseteq X$*, then* $\sigma' \models \sigma$*.* □

If $X' \rightarrow Y$ holds, then literals in X', Y are in *closure(*X, Σ_Q*)*. Since $X' \subseteq X$, we can derive Y, and σ holds.

Axiom 3: *(Pattern Augmentation) If* $\sigma' = (Q'[\bar{x}'], \Delta, X \rightarrow Y)$*, and* $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$*,* $Q' \subseteq Q$*, then* $\sigma' \models \sigma$*.* □

In Axiom 3, if Q' is isomorphic to a subgraph of Q, and if σ' (with pattern Q') holds, it will continue to hold under Q.

Axiom 4: *(Transitivity) If* $\sigma' = (Q'[\bar{x}'], \Delta, X \rightarrow W)$*,* $\sigma = (Q[\bar{x}], \Delta, W \rightarrow Y)$*, where* $Q' \subseteq Q$*, then for any* $\sigma'' = (Q[\bar{x}], \Delta, X \rightarrow Y)$*, it follows that* $\{\sigma, \sigma'\} \models \sigma''$*.* □

In Axiom 4, all matches that satisfy σ' will also be contained within matches satisfying σ since $Q' \subseteq Q$. By transitivity of equality *w.r.t.* the literals in W, these matches of Q will satisfy $X \rightarrow Y$, thereby showing $\{\sigma, \sigma'\} \models \sigma''$.

Axiom 5: *(Decomposition) If* $\sigma = (Q[\bar{x}], \Delta, X \rightarrow Y)$ *with* $Y = \{l_1, l_2\}$*, then for* $\sigma' = (Q[\bar{x}], \Delta, X \rightarrow l_1)$ *and* $\sigma'' = (Q[\bar{x}], \Delta, X \rightarrow l_2)$*, it follows that* $\sigma \models \{\sigma', \sigma''\}$*.* □

Verifying the literals in Y can be done simultaneously in one verification (via σ), or in conjunction (via σ', σ'').

Axiom 6: *(Interval Intersection) If* $\sigma = (Q[\bar{x}], (p,q), X \to Y)$*,* $\sigma' = (Q[\bar{x}], (p',q'), X \to Y)$*, then for* $\sigma'' = (Q[\bar{x}], (p'',q''), X \to Y)$*, where* $(p'',q'') = (p,q) \cap (p',q')$*, it follows that* $\{\sigma, \sigma'\} \models \sigma''$*.* □

For σ, σ' with matches satisfying $X \to Y$ over Δ, Δ', respectively, requires verifying all pairwise matches over all sub-intervals $\Delta'' \subseteq (\Delta \cap \Delta')$, thereby showing $\{\sigma, \sigma'\} \models \sigma''$.

Axiom 7: *(Interval Containment) If* $\sigma = (Q[\bar{x}], \Delta, X \to Y)$*, and* $\sigma' = (Q[\bar{x}], \Delta', X \to Y)$*,* $\Delta' \subseteq \Delta$*, then* $\sigma \models \sigma'$*.* □

If σ holds over a time interval Δ, then it will continue to hold over any subsumed sub-interval Δ' that is more restrictive. Pairwise matches over all sub-intervals $\Delta' \subseteq \Delta$ must also satisfy the dependency, thus showing σ' holds.

Theorem 3: *The axiomatization is sound and complete.* □

Proof: The axioms are sound as described above. To show completeness of Axioms 1-4, recall that the *closure(*X, Σ_Q*)* is defined over all embedded TGFDs for a pattern Q, i.e., by Axiom 3, if σ' holds over $Q' \subseteq Q$, then σ holds for pattern Q. In computing *closure(*X, Σ_Q*)*, inference of valid time intervals is done by checking Axiom 6, where the interval overlap is non-empty. For $\sigma = (Q[\bar{x}], \Delta, X \to y)$ can be inferred from Σ, if and only if there exists a literal $(y, \Delta'') \in$ *closure(*X, Σ_Q*)* such that $\Delta \subseteq \Delta''$.

To show completeness, it is sufficient to show that for any σ, if $\Sigma \not\models \sigma$, then there exists a $\mathcal{G}_T$ such that $\mathcal{G}_T \models \Sigma$, but $\mathcal{G}_T \not\models \sigma$, i.e, Σ does not logically imply σ. First, we show that $\mathcal{G}_T \models \sigma'$, for all $\sigma' \in \Sigma$, and then show σ is not satisfied by $\mathcal{G}_T$. Suppose $\sigma' = (Q[\bar{x}], \Delta, V \rightarrow W)$ is in Σ but not satisfied by $\mathcal{G}_T$. Then $(V, \Delta) \in$ *closure(X, Σ_Q)*, and (W, Δ) nor any subset of literals of W can be in *closure(X, Σ_Q)*, otherwise, σ' would be satisfied by $\mathcal{G}_T$. Let (w, Δ) be a literal of W not in *closure(X, Σ_Q)*. We have $\sigma'' = (Q'[\bar{x}], \Delta, X \rightarrow V)$, since V is in the closure, and by Axiom 3 and Axiom 4, we have $\sigma''' = (Q[\bar{x}], \Delta, X \rightarrow W)$. By Axiom 1, we can infer $W \rightarrow w$ over the interval Δ, and by transitivity w.r.t. W over pattern Q, we have $X \rightarrow w$, which implies that $(w, \Delta) \in$ *closure(X, Σ_Q)*, which is a contradiction. Hence, $\mathcal{G}_T \models \Sigma$, for all $\sigma' \in \Sigma$. Second, we show that for a σ, if $\Sigma \not\models \sigma$, then $\mathcal{G}_T \not\models \sigma$. Let's suppose $\mathcal{G}_T$ $\models \sigma$, then $(y, \Delta) \in$ *closure(X, Σ_Q)*. However, if this is true, then $\Sigma \models \sigma$, which is a contradiction. Hence, $\mathcal{G}_T \not\models \sigma$, and the axiomatization is complete.

2.3.4 Validation

Given Σ, and a temporal graph $\mathcal{G}_T$, the validation problem is to decide whether $\mathcal{G}_T$ $\models \Sigma$. A practical application of validation is to detect *violations* of Σ in $\mathcal{G}_T$. We say a pair of matches (h_i, h_j) is a violation ("error") of a TGFD $\sigma = (Q(\bar{x}), \Delta, X \rightarrow Y) \in \Sigma$, if (h_i, h_j) matches σ and $(h_i, h_j) \not\models Y$. We denote the set of violations of Σ in $\mathcal{G}_T$ as $\mathcal{E}(\mathcal{G}_T, \Sigma)$. The validation problem is to decide whether $\mathcal{E}(\mathcal{G}_T, \Sigma)$ is empty.

Theorem 4: *The validation of* TGFDs *is* coNP*-complete.* □

Proof sketch: The lower bound can be verified from the GFDs counterpart, where the validation problem is coNP-complete [57]. For TGFDs, an NP algorithm returns "yes" if $\mathcal{G}_T \not\models \Sigma$ as follows. It (1) guesses a TGFD σ, and guesses and verifies a pair of mappings (h_i, h_j) over a finite number of sub-intervals $\{[i, j] \mid 1 \leq i, j \leq T, |j - i| \in \Delta\}$, and (2) checks whether $(h_i, h_j \models X$, but $(h_i, h_j) \not\models y$, if so, return "yes". Checking all matches over the intervals is in PTIME. Since the complement to the decision problem still remains in NP, the validation problem for TGFDs is coNP-complete, no harder than its GFDs counterpart. □

2.4 Parallel TGFD Error Detection

We introduce a parallel TGFD error detection algorithm, ParallelTED, that includes fine-grained workload estimation, temporal pattern matching with match maintenance over evolving graph fragments, and incremental error detection that avoids all pairwise match enumerations.

A Sequential Algorithm. Given a TGFD $\sigma = (Q(\bar{x}), \Delta, X \rightarrow Y)$, a sequential algorithm computes $\mathcal{E}(\mathcal{G}_T, \Sigma)$ as follows. (1) For each snapshot $G_t \in \mathcal{G}_T$, it computes from scratch all matches of Q in G_t, and those in the subsequent snapshots G_j where $|j - t| \in \Delta$ (if not computed yet). (2) It then verifies, for each pair of matches of Q $\{(h_i, h_j)\}$ with $|j - i| \in \Delta$, if $(h_i, h_j) \models X$ (*i.e.*, (h_i, h_j) matches σ), and

$(h_i, h_j) \not\models Y$. If so, a violation (h_i, h_j) is identified and added to $\mathcal{E}(\mathcal{G}_T, \Sigma)$. This approach separately performs pattern matching and TGFD validation, and is infeasible for large graphs due to an excessive number of subgraph isomorphism tests (which is known to be NP-complete [62]), and comparisons over each pairs of snapshots. As most changes in real-world temporal graphs affect a small portion (*e.g.,* 5-10% of nodes weekly [51]) and 80-99% of the changes are highly localized [32], this naturally motivates incremental and parallel solutions for TGFD-based error detection.

We next present ParallelTED, a *Parallel TGFD Error Detection* algorithm, for error detection in large $\mathcal{G}_T$. The algorithm fully exploits the temporal interval constraints to interleave parallel pattern matching and incremental error detection, in at most T rounds of parallel computation.

Overview. The algorithm ParallelTED (Algorithm 1) works with a set of n worker machines $M_1 \ldots M_n$ and a coordinator M_c. It executes in total T supersteps. Each superstep t processes a *fragmentation* of a snapshot G_t as a set of subgraphs $\{F_{tr}\}$ of G_t ($t \in [1, T]$ and $r \in [1, n]$), Intuitively, we perform parallel computation to detect and maintain the set of violations ($\mathcal{E}(\{G_1, \ldots, G_t\}, \Sigma)$) over "currently observed" snapshots $\{G_1, \ldots, G_t\}$, at each superstep t. The algorithm terminates after T supersteps, and ensures the correct computation of $\mathcal{E}(\mathcal{G}_T, \Sigma)$ given the correct update of violation set at each superstep.

Algorithm. ParallelTED maintains the following structures that are dynamically updated. (1) Each worker M_r maintains (a) a set of local matches of Q of snapshot

G_t $M_r(Q, G_t)$, (b) a set of local violations $\mathcal{E}_r(\mathcal{G}_T, \Sigma)$, by only accessing local graphs (initialized as fragment $\{F_{tr}\}$), and (c) a set of "cross-worker" violations $\mathcal{E}_r(\mathcal{G}_T, \Sigma)$, which requires the comparison of two matches from M_r and a different worker. The coordinator maintains a set of global violations $\mathcal{E}(\mathcal{G}_T,\Sigma)$ to be assembled from local violations.

ParallelTED first invokes a procedure GenAssign to initialize and assigns a set of joblets (a set of lightweighted validation tasks; see Section 2.4.1) to all the workers, and initializes global structures for incremental maintenance of the pattern matches (see Section 2.4.2) (lines 1-2). This cold-starts the parallel error detection upon the processing of the first fragmented snapshot in $\mathcal{G}_T$. It then runs in T supersteps in parallel (lines 3-10), following a bulk synchronous model, and processes one fragmented snapshot a time.

In each superstep t, each worker M_r executes the following two steps in parallel (lines 4-7): (1) incrementally updates a set of local matches $M_r(Q, G_t)$ and violations $\mathcal{E}_r(\mathcal{G}_T,\Sigma)$ over fragmented snapshot $\{F_{tr}\}$, by invoking a procedure LocalVio (line 4); and (2) requests a small amount of edges from other workers, and invokes a procedure IncTED to incrementally detect the violations $\mathcal{E}_c(\mathcal{G}_T , \Sigma)$ across two workers (line 5; see Section 2.4.2). M_r then returns the updated local violations (augmented with cross-worker violations) to M_c (lines 6-7).

The coordinator M_c incrementally maintains a set of global violations upon receiving the updated local violations (line 8). It also invokes procedure GenAssign to rebalance the workload for the next superstep (lines 9-10) upon a triggering condition.

This completes a validation of Σ over the snapshots $\{G_1, \ldots, G_t\}$ "seen" so far.

Optimizations. To cope with skewed localized updates that lead to stragglers and reduce communication overhead, ParallelTED adopts two strategies below.

(1) An adaptive time-interval aware workload balancing strategy (procedure GenAssign; line 6) to automatically (re-)partition the TGFD validation workload to maximize parallelism. The idea is to decompose validation tasks to a set of highly parallizable, small subtasks that are induced by path patterns, time-intervals, and common literals ("Joblets"), and dynamically estimates a bounded subgraph induced by the joblets to be assigned and executed in parallel.

(2) ParallelTED also adopts an incremental pattern matching scheme (Algorithm IncTED; line 5) to maintain the local violations. The strategy performs case analysis of edge updates and only processes a necessary amount of validation, to reduce the communication and local detection cost. We next introduce the details of GenAssign and IncTED.

2.4.1 Time interval-Aware Workload Balancing

Given a set of TGFDs Σ and $\mathcal{G}_T$, procedure GenAssign creates a set of "joblets" and estimates their processing cost for balanced workload assignment (as illustrated in

```
Algorithm 1: ParallelTED ($\mathcal{G}_T$, $\Sigma$)
1  Initialize $\Pi_X$, $\pi_{XY}$, $\Gamma(\Pi_X)$, $\Gamma(\pi_{XY})$;
2  GenAssign($\mathcal{G}_T$, $\Sigma$);
3  foreach $t \in T$ do
       /* at worker $r$ in parallel */
4      $\mathcal{E}_r(F_r,\Sigma)$:= $\mathcal{E}_r(F_r,\Sigma) \cup$ LocalVio($F_{tr}$,$\mathcal{J}_{tr}$ ,$t$);
5      $\mathcal{E}_c(\mathcal{G}_T ,\Sigma)$:= $\mathcal{E}_c(\mathcal{G}_T ,\Sigma) \cup$ IncTED($G_t$, $\Sigma$,$\mathcal{M}_r(Q,G_t)$);
6      $\mathcal{E}_r(F_r,\Sigma)$:= $\mathcal{E}_r(F_r,\Sigma) \cup \mathcal{E}_c(\mathcal{G}_T ,\Sigma)$;
7      return $\mathcal{E}_r(F_r,\Sigma)$;
       /* at the coordinator side */
8      $\mathcal{E}(\mathcal{G}_T ,\Sigma)$ := $\mathcal{E}_c(\mathcal{G}_T ,\Sigma) \cup \bigcup_r \mathcal{E}_r(F_r,\Sigma)$;
9      if $(t_{\mathcal{J}_{tr}} < ((1-\zeta)\cdot t_l)) \vee (t_{\mathcal{J}_{tr}} > ((1+\zeta)\cdot t_u))$ then
10         Assign($\mathcal{J}_{tr}$, $CCost$); /* rebalance workload */
11 return $\mathcal{E}(\mathcal{G}_T ,\Sigma)$
   Procedure GenAssign($\mathcal{G}_T$, $\Sigma$)
   /* at the coordinator side */
1  $\mathcal{E}(\mathcal{G}_T ,\Sigma)$:= $\emptyset$; $\mathcal{E}_c(\mathcal{G}_T ,\Sigma)$:= $\emptyset$; $CCost := \emptyset$;
2  foreach $\sigma \in \Sigma$ do
3      Define $\mathcal{J}_{tr}(\sigma, F_{tr}, \mathcal{G}_T)$ and estimate $|\mathcal{J}_{tr}|$;
4      Estimate communication cost $CCost(\mathcal{J}_{tr})$
5  Assign($\mathcal{J}_{tr}$, $CCost$);
6  Send $\mathcal{J}_{tr}$ across workers $M_r$;
```

Algorithm 1).

Joblets and Jobs. A *joblet* characterizes a small TGFD validation task that can be conducted by a worker in parallel. A joblet at superstep t is a triple $\omega_{trk}(Q_k,F_{tr},\mathcal{G}(v',d))$, where

- $Q_k(v_k,d)$ is a sub-pattern of a pattern Q with a designated center node v_k and a radius d *w.r.t.* v_k, for a TGFD $\sigma = (Q[\bar{x}],\Delta,X \to Y)$ in Σ;
- F_{tr} is a fragment of snapshot G_t on worker M_r; and

- $\mathcal{G}(v', d) = \bigcup_{s,s' \leq t}\{\{G_s(v', d), G'_s(v', d)\}||s' - s| \in \Delta\}$, where v and v' have the same label, and $G'_s(v', r)$ is a subgraph of G_s induced by v' and its d-hop neighbors.

Intuitively, a joblet encodes a fraction of a validation task to identify violations of a TGFD $\sigma \in \Sigma$. It is dynamically induced by incorporating a small fragment of Q and only the relevant fraction of snapshots in $\mathcal{G}_T$ that should be checked to detect possible violations of a TGFD σ with pattern Q and time interval Δ.

A *job* $\mathcal{J}_{tr}(\sigma,\ F_{tr},\ \mathcal{G}_T\ )$ refers to a set of joblets that encode the workload to validate a TGFD σ with pattern Q. A job contains all joblets for all subqueries Q_k from Q. We denote as $\mathcal{J}_{tr}(\Sigma,\ F_{tr},\ \mathcal{G}_T\ )$ the jobs for validating a set of TGFDs Σ, *i.e.,* $\mathcal{J}_{tr}(\Sigma,\ F_{tr},\ \mathcal{G}_T\ ) = \cup_{\sigma\in\Sigma}\ \mathcal{J}_{tr}(\sigma,\ F_{tr},\ \mathcal{G}_T\ )$.

We next introduce a path decomposition strategy adopted by GenAssign. The joblets are created accordingly for a given decomposition of $Q = \{Q_1, \ldots Q_k\}$.

K-Path decomposition (not shown in Algorithm 1). It has been shown that the cost of graph query processing can be effectively estimated using path queries [101]. GenAssign adopts a K-path pattern decomposition strategy for joblet creation and cost estimation. For each $\sigma \in \Sigma$, GenAssign decompose a pattern Q into a set of paths $\{Q_1, \ldots Q_K\}$, such that Q is the union of Q_k ($k \in [1, K]$). Each Q_k is a maximal path from a node v_{k_1} to a destination node v_{k_m} that cannot be further extended by pattern edges. Each Q_k is augmented with value constraints posed by the literals from $X \cup Y$.

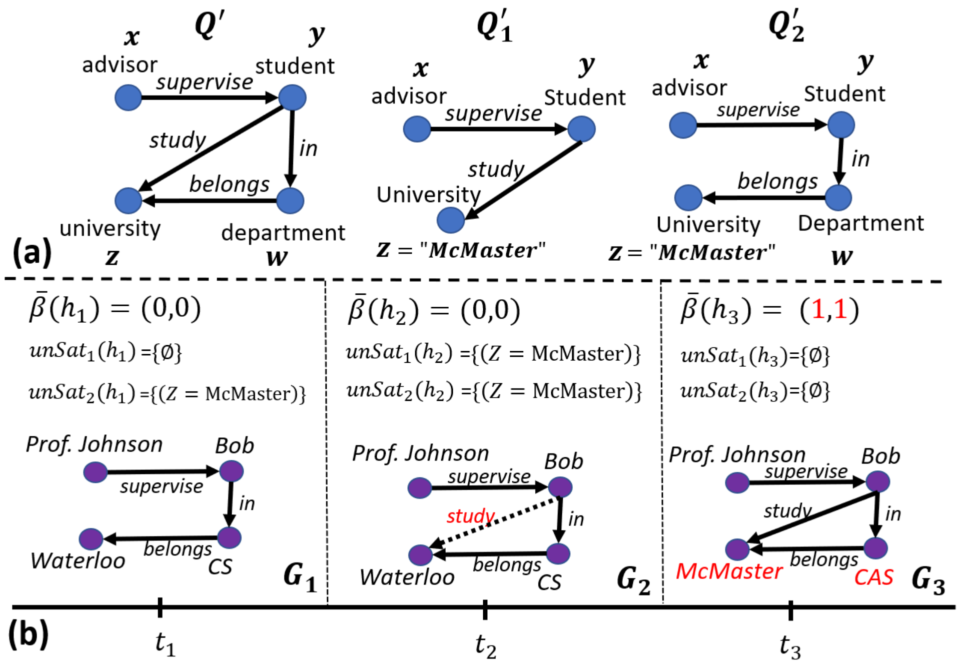


Figure 2.2: Pattern paths and partial matches.

Let v_k be a center node with minimum radius in each path pattern Q_k ($k \in [1, K]$), where the radius d is the longest shortest path between v_k and any node in Q_k. The joblets are then created with K-path decomposition and induced subgraphs *w.r.t.* v_k and Q_k ($k \in [1, K]$) accordingly. Figure 2.2(a) shows two (maximal) pattern paths Q'_1 and Q'_2 of the pattern Q' with the literal set $\{z = \textit{McMaster}\}$.

Workload Estimation (line 4 of Procedure GenAssign). The workload is estimated as the size of joblets/jobs, and the expected communication cost among the workers. To estimate the joblet size, we adapt the cardinality estimation for graph queries [101]. We compute a probability distribution of the expected number of matches for each edge (v_l, v_{l+1}) in Q_k with label a_l. The distribution provides a probabilistic estimation of the number of matches of v_{l+1}, given the matches of v_l that are connected via an edge with label a_l and satisfy the literals over v_l and v_{l+1}. We estimate using the mean and standard deviation of the number of matches of v_{l+1} *w.r.t.* v_l. We compute the distribution function of Q_k as the product of the distribution functions of each consecutive edge in Q_k. We estimate the size of a job $|\mathcal{J}_{tr}(\sigma,\ F_{tr},\ \mathcal{G}_T\ )\ |$ as the cardinality of the pattern Q, i.e., the number of matches of Q, computed as the upper-bound of the minimum cardinality of all Q_k.

Communication cost estimation (line 10 of Algorithm ParallelTED; line 4 of Procedure GenAssign). For a joblet $\omega_{trk}(Q_k, F_{tr}, \mathcal{G}(v', d))$, fragment F_{tr} may contain a partial match of a path Q_k, i.e., some nodes and edges that match Q_k reside in machine $M_{r'}, r' \neq r$. We must exchange these small number of edges between workers that we estimate as the communication cost of a joblet, defined as $CCost(\omega_{trk}) = \sum_{\substack{(v_1,v_2)\in G_t(v',d),\\ \neg((v_1,v_2)\in F_{tr})}} |(v_1, v_2)|$. We aggregate over all the joblets containing pattern paths Q_k of Q to define the communication cost of a job, $CCost(\mathcal{J}_{tr}) = \sum_k CCost(\omega_{trk})$.

Workload Assignment and Rebalancing (lines 9-10 of Algorithm ParallelTED; line 5 of Procedure GenAssign). We distribute jobs to workers such that the communication cost C, and the makespan τ is minimized. We solve a *general assignment (GA) problem* [102]. Given C and τ, the GA problem is to find an assignment of each job such that the total parallel cost is bounded by C, and minimizes the makespan. Following [102], we develop a pseudo-polynomial-time, 2-approximation algorithm. It performs a bisection search on the range of possible estimated makespan by keeping only the assignments with an objective function value less than half of the current best solution. It solves a linear program at most $log(wP)$ times (w is the number of jobs and P is the job with maximum estimated workload) and select the one with the smallest makespan.

The initial workload distribution guarantees the makespan is at most 2τ provided $t_l \leq t_{\mathcal{J}_{tr}} \leq t_u, \forall t_{\mathcal{J}_{tr}}$, for given (t_l, t_u). Changes among the fragments (increased density of edges, attributes updates) can create workload imbalance among the workers, and increase the makespan τ. To avoid excessive workload re-distributions, we define a *burstiness time buffer*, ζ, representing the allowable percentage change in job runtime due to workload bursts. We add ζ to (t_l, t_u) to define $t'_l = ((1- \zeta) \cdot t_l)$, and $t'_u = ((1+ \zeta) \cdot t_u)$. For any job where its runtime $t_{\mathcal{J}_{tr}}$ lies beyond (t'_l, t'_u), M_c triggers a workload distribution and rebalances the workload. In Section 2.5.3, we study the overhead and frequency of workload re-distributions over real workloads for $\zeta = 0.1$ under varying rates of change, and found the overhead is about 6% of the total runtime.

2.4.2 Parallelized Incremental Temporal Matching

Another bottleneck is to compute the pattern matches over evolving and distributed fragments. We next introduce the procedures LocalVio and IncTED that efficiently maintain the matches in ParallelTED.

Incremental Local Matching. The local violation computation (Procedure LocalVio (Algorithm 2), line 4 of Algorithm ParallelTED) relies on fast computation of local pattern matches. At each superstep t, jobs are executed at each worker to compute matches of each query path Q_k corresponding to Q. For each Q_k in a joblet, it performs subgraph isomorphism matching on $\mathcal{G}(v', d)$ to find local matches. Matches that require edges from multiple workers are shipped to a single worker. The matches of Q are obtained by (1) taking the intersection of the nodes identified by the matches across all paths Q_k of Q, and (2) a verification to find true matches.

Coping with edge updates. New matches of Q_k can appear and existing matches disappear (or be updated) as changes occur to fragment F_{tr}. To adapt to these changes, we present an incremental matching strategy that avoids redundant computations, and tracks partial matches for each Q_k with their missing topological and literal conditions.

We introduce a boolean vector $\bar{\beta}(h_i) = (b_1, \ldots, b_\kappa)$ over a candidate match h_i of Q in F_{tr}. $\bar{\beta}(h_i)$ contains κ components, where the k-th component is the matching status of $\omega_{trk}(Q_k, F_{tr}, \mathcal{G}(v', d))$. That is, if there exists a match h_i of Q_k in F_{tr}, then $\bar{\beta}(h_i) = 1$ (true), otherwise, $\bar{\beta}(h_i) = 0$ (false). In the latter case, a partial match h_i may be isomorphic to Q_k but not satisfy its literals. We define $\mathsf{unSat}_k(h_i)$ as the set of unsatisfied literals $x' \in \bar{x}$ in h_i *w.r.t.* Q_k, *e.g.*, $\mathsf{unSat}_k(h_i) = \{x'.A = c \mid h_i(x').A \neq c\}$. We use $\bar{\beta}$ and unSat_k when h_i is clear from the context.

Each worker initializes and populates $\bar{\beta}$ and unSat_k as matches are found. As changes occur over the fragments, ParallelTED checks $\bar{\beta}$, and unSat_k to determine if new matches arise, and updates existing matches to minimize the need for (expensive) subgraph isomorphism operations. If the k-th component of $\bar{\beta}$ equals false, there are

Algorithm 2: LocalVio($F_{tr}, \mathcal{J}_{tr}, i$)

```
  /* compute matches and errors over $G_1$ */
1 ($\mathcal{M}_r(Q, G_1)$, $h_i$) := IsoUnit ($Q$, $G_1$)
  /* incrementally compute matches for $G_2$ onwards */
2 foreach $c \in changes$ do
3     $\mathcal{M}_r(Q, G_i) := \mathcal{M}_r(Q, G_i) \cup$ LMatch($F_{tr}, \mathcal{J}_{tr}, \mathcal{M}_r(Q, G_{i-1}), c$);
4 $\mathcal{E}_r(F_r, \sigma)$:= $\mathcal{E}_r(F_r, \sigma) \cup$ IncTED($F_{tr}, \sigma, \mathcal{M}_r(Q, G_i)$); return $\mathcal{E}_r(F_r, \sigma)$;
```

two cases to consider: (i) a topological match of Q_k exists, but there are unsatisfying literals in unSat_k; or (ii) an empty unSat_k represents no topological match of Q_k in F_{tr}. For a job $\mathcal{J}_{tr}$, we evaluate the following cases for an input change c:

(a) attribute insertion/deletion/update: we check whether c adds/removes literals from unSat_k, and update $\bar{\beta}$ to denote the insertion/deletion of a match h_i *w.r.t.* Q_k.

(b) edge insertion: we perform subgraph isomorphism matching to determine whether a partial match is upgraded to a complete match *w.r.t.* Q_k.

(c) edge deletion: if c causes a prior match h_i *w.r.t.* Q_k to be removed, we then set $\bar{\beta}(h_i)[k]$ to false, and add any unsatisfying literals to $\mathsf{unSat}_k(h_i)$.

Algorithm 2 provides details of the local matching and error detection at each worker. For the first snapshot, we compute matches using a localized subgraph isomorphism matching algorithm, IsoUnit [51] (line 1). For the subsequent snapshots, matches are incrementally maintained based on a local change c via LMatch (lines 2-3). After processing all the changes, we compute the violations for σ via IncTED (line 4), and return the local error set (line 5).

Algorithm 3: $\mathsf{LMatch}(F_{tr}, \mathcal{J}_{tr}, \mathcal{M}_r(Q, G_{i-1}), c)$

```
1  $\mathcal{M}_r(Q, G_i) := \emptyset$;
2  Initialize $\bar{\beta}$ and $\mathsf{unSat}_k$ from $\mathcal{M}_r(Q, G_{i-1})$;
3  $\mathcal{M}_{chg} = \{h_{i-1} \in \mathcal{M}_r(Q, G_{i-1}) \mid c$ applied to $G_{t-1}\}$
4  switch $c$ do
5      when insert/update$(x.A = a)$;
6      if $b_k = 0$ and $(x.A = a) \in \mathsf{unSat}_k$ then
7          $\mathsf{unSat}_k := \mathsf{unSat}_k \setminus (x.A = a)$;
8          $b_k = 1$;
9      else if $b_k = 1$ and $(x.A = b) \in Q_k$ then
10         $\mathsf{unSat}_k := \mathsf{unSat}_k \cup (x.A = a)$;
11         $b_k = 0$;
12     when delete$(x.A = a)$;
13     if $b_k = 1$ and $(x.A = b) \in Q_k$ then
14         $\mathsf{unSat}_k := \mathsf{unSat}_k \cup (x.A = b)$;
15         $b_k = 0$;
16     when insert$(e)$;
17     if new match for $Q_k$ occurs then
18         Duplicate all vectors $\bar{\beta}$, $b_k = 1$;
19     when delete$(e)$;
20     if $\bar{\beta}(h_i)[k] = 1$ then
21         $b_k = 0$; $\mathsf{unSat}_k(h_i) = \emptyset$;
22 foreach $h_i \in \mathcal{M}_{chg}$ do
23     if $(\bar{\beta}(h_i)[k] = 1, \forall k)$ then
24         $\mathcal{M}_r(Q, G_i) := \mathcal{M}_r(Q, G_i) \cup h_i$;
25 return $\mathcal{M}_r(Q, G_i)$;
```

The algorithm LMatch is illustrated in Algorithm 3. We initialize (lines 1-2), and apply the change c to existing matches to determine whether the match remains or is removed (line 3). For attribute updates, we update $\bar{\beta}$ and unSat_k without performing any subgraph isomorphism operations (lines 5-15). For edge insertions, we perform subgraph isomorphism along Q_k to check for a new match (lines 16-18). For edge deletions, which can only remove matches, we update b_k for Q_k (lines 19-21). A match h_i of Q exists if $b_k = 1$ for all k, and we add h_i to $\mathcal{M}_r(Q, G_i)$ (lines 22-24).

Algorithm 4: IncTED($\mathcal{G}_T$,σ,$\mathcal{M}(Q, G_t)$)

```
1  $\mathcal{E}(\mathcal{G}_T , \sigma) := \emptyset$;
2  foreach $h_i \in \mathcal{M}(Q, G_t)$ do
3      $\{\Pi_X, \pi_{XY}\} := \{\Pi_X, \pi_{XY}\} \cup h_i$
4      $\{\Gamma(\Pi_X), \Gamma(\pi_{XY})\} := \{\Gamma(\Pi_X), \Gamma(\pi_{XY})\} \cup i$;
5      $\rho(i) := [\mathsf{max}(1, i-p), \mathsf{min}(i+q, T)]$;
6      if $\{\mu_X(h_{j)}\} \setminus \mu_X(h_i)\} \neq \emptyset, \forall j \in \rho(i)\}$ then
7          $\mathcal{E}(\mathcal{G}_T , \sigma) := \mathcal{E}(\mathcal{G}_T , \sigma) \cup \{(h_i, i), (h_j, j)\}$;
8      foreach $(x.A = a) \in Y$ do
9          if $h_i .A \neq a$ then
10             $\mathcal{E}(\mathcal{G}_T , \sigma) := \mathcal{E}(\mathcal{G}_T , \sigma) \cup \{(h_i, i)\}$;
11 return $\mathcal{E}(\mathcal{G}_T , \sigma)$;
```

Example 9: Figure 2.2(b) shows (partial) matches of patterns Q'_1, Q'_2. At t_1, $\bar{\beta}(h_1)$ denotes no match of either Q'_1 nor Q'_2, while $\bar{\beta}(h_1)[2]$ indicates a topological match of Q'_2 with non-empty unSat_2 containing literal $z = McMaster$. At t_2, with an edge insertion labeled *study* from *Bob* to *Waterloo*, we perform a subgraph isomorphism matching and find a topological match of Q'_1. However, $\bar{\beta}(h_2(\bar{x}))$ remains false due to the unsatisfying *McMaster* literal in $\mathsf{unSat}_1(h_2)$. Lastly, the update at t_3 to *University* from *Waterloo* to *McMaster* clears all unsatisfying literals, and updates $\bar{\beta}(h_3)$ to true for Q'_1 and Q'_2, without requiring any additional matching. □

Validating each $\sigma \in \Sigma$ may require us to store and pairwise compare all the local and cross-worker matches. We next introduce procedure IncTED, an Incremental TGFD Error Detection algorithm. The algorithm avoids the need to store the matches, and reduces the cost of the exhaustive pairwise match comparisons by performing efficient set difference operations.

Permissible Ranges. Rather than exhausting the comparison of any pair of matches, IncTED verifies match pairs with respect to a "current" time, $c = i$, and induces an allowable time range between h_i and h_j as defined by Δ. We define I_{h_i} and I_{h_j} as follows:

$$\begin{aligned} I_{h_i} &= [1, |T| - p] \\ I_{h_j} &= [(i+p), j'], \text{where } j' = \begin{cases} T, (i+q) > T \\ (i+q), o.w. \end{cases} \end{aligned} \tag{2.4.1}$$

If matches h_i, h_j have the same values in X, we want to identify matches h_j to compare against h_i by defining the allowable time range. We call this $\rho(i)$, the *permissable range* of h_i, as $\rho(i) = \{j \mid |j - i| \in \Delta\}$.

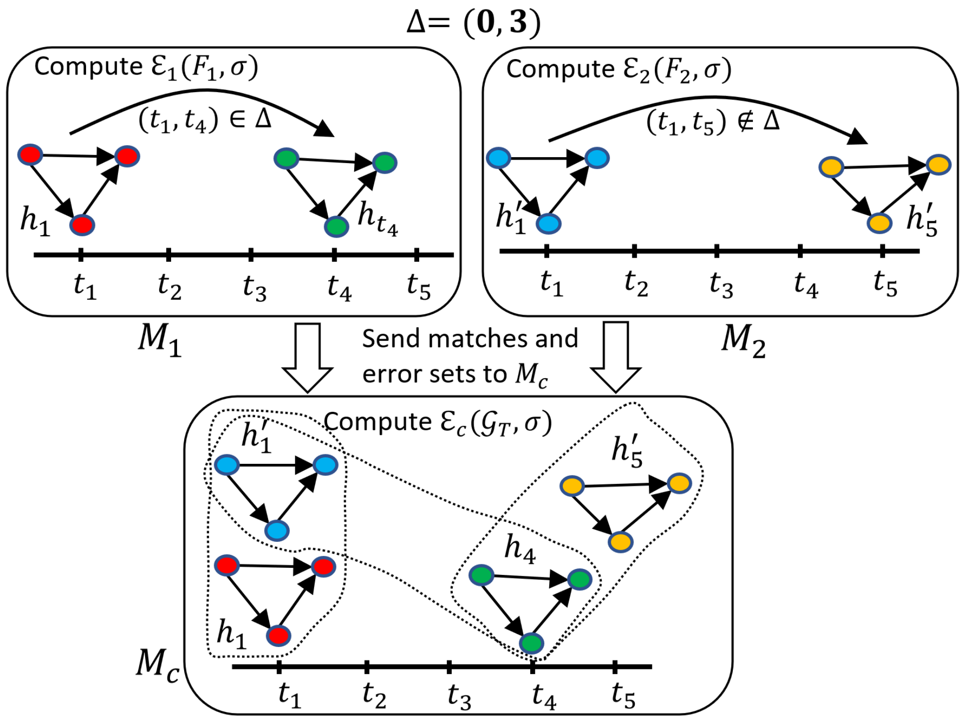
Auxiliary Structures. IncTED uses the following notations and auxiliary structures. To avoid enumeration of all pairwise matches in $\rho(i)$, we define a hash map, Π_X (i), that partitions all matches of Q according to their values in X up to and including i. Specifically, let $\Pi_X = \{h_j \mid h_i .A = h_j.A, \forall\ (h_i .A, h_j.A) \in X, i \leq j\}$. We simply use Π_X when i is clear from the context. We sub-partition the matches in Π_X according to their distinct values in Y to identify error matches with different consequent values within the permissable range. We record the timestamps of such matches Π_X (resp. π_{XY}) in $\Gamma(\Pi_X)$ (resp. $\Gamma(\pi_{XY})$), *i.e.*, $\Gamma(\Pi_X) = \{j \mid h_j \in \Pi_X\ \}$. We define a mapping function that returns the π_{XY} partition in which h_i belongs as $\mu_X(h_i) = \pi_{XY}$ such that $h_i \in \pi_{XY}$. For example, Figure 2.1 shows matches of σ with $\Pi_X = \{h_1, \ldots, h_6\}$, $\Gamma(\Pi_X) = \{t_1, \ldots, t_6\}$, and $\pi_{XY} = \{\{h_1, \ldots, h_5\}\{h_6\}\}$, $\Gamma(\pi_{XY}) = \{\{t_1, \ldots, t_5\}\{t_6\}\}$.

These global structures and variables are maintained by the coordinator M_c and shared among all workers.

Algorithm. The algorithm IncTED is illustrated in Algorithm 4. After initializing the local error set (line 1), it iterates over the local matches and records the timestamps of each match in the auxiliary structures (lines 3-4). For each match h_i, the $\rho(i)$ is bookkept based on the timestamp of h_i (line 5). For variable TGFDs, if there exists another match h_j within the permissable range of h_i such that h_i and h_j have different set of timestamps for π_{XY}, then IncTED adds the pair to the violation set (lines 6-7). For constant TGFDs, if the match h_i violates a constant literal in Y, then it adds h_i to the violation set (lines 8-10).

Using IncTED algorithm, each worker computes its local set of errors, $\mathcal{E}_r(F_r, \sigma)$. For matches that span two workers, the coordinator verifies matches $h_i, h_{i'}$ from M_r, $M_{r'}, r \neq r'$, respectively. This is done by selecting matches from $\mu_X(h_i) = \pi_{XY}\ (h_i)$ and $\mu_X(h_{i'}) = \pi_{XY}\ (h_{i'})$ such that $|i - i'| \in \Delta$, and computing $\{\mu_X(h_{i'})\} \setminus \{\mu_X(h_i)\}$, *i.e.,* checking whether the set difference is non-empty. If so, we add $\{(h_i, i), (h_{i'}, i')\}$ to the cross-machine error set $\mathcal{E}_c(\mathcal{G}_T\ , \Sigma)$. At each timestamp, we update $\mathcal{E}(\mathcal{G}_T\ , \Sigma)$ with $\mathcal{E}_c(\mathcal{G}_T\ , \Sigma) \bigcup_r \mathcal{E}_r(F_r, \Sigma)$.

We found that the incremental strategy effectively improves the efficiency of a



Δ= (0, 3)
Compute $\mathcal{E}_1(F_1, \sigma)$
$(t_1, t_4) \in \Delta$
h_1
h_{t_4}
t_1 t_2 t_3 t_4 t_5
M_1
Compute $\mathcal{E}_2(F_2, \sigma)$
$(t_1, t_5) \notin \Delta$
h'_1
h'_5
M_2
Send matches and error sets to M_c
Compute $\mathcal{E}_c(\mathcal{G}_T, \sigma)$
h'_1
h'_5
h_4
h_1
M_c

Figure 2.3: Error detection in ParallelTED

batch counterpart for TGFD-based error detection by 3.3 times (see Section 2.5).

Example 10: Figure 2.3 shows matches $\{h_1, h_4\}$ and $\{h'_1, h'_5\}$ for a σ with $\Delta = (0, 3)$ computed locally (via LMatch [18]) at M_1 and M_2, respectively. M_1 computes local error set $\mathcal{E}_1$ via IncTED, and $\mathcal{E}_2 = \emptyset$ since $|t_5 - t_1| \notin \Delta$. M_c receives $\{h_1, h'_1\}, \{h_4, h'_1\}, \{h_4, h'_5\}$, validates the pairs (shown by dotted lines) via IncTED, and adds the violations to $\mathcal{E}(\mathcal{G}_T\ , \sigma) \mathrel{+}= \mathcal{E}_c(\mathcal{G}_T, \sigma) \bigcup \mathcal{E}_1$. □

2.5 Experiments

We evaluate our algorithms with three objectives: (1) scalability and impact of varying parameters; (2) the effectiveness of TGFD-based error detection compared to GFDs and GTARs; and (3) case study that verifies real-world TGFDs and errors that can be captured.

2.5.1 Experimental Setup

Datasets. We use two real, and one synthetic graph. All datasets and source code are publicly available at [6].
(1) DBpedia [79]: The graph contains in total 2.2M entities with 73 distinct entity types, and 7.4M edges with 584 distinct labels from 2015 to 2016, with snapshots every 6 months.
(2) IMDb [5]: The data graph contains 4.8M entities with 8 types and 16.7M edges. IMDB provides diff files, where we extract 38 monthly updates from Oct. 2014 to Nov. 2017.
(3) Synthetic: We use the gMark benchmark [26], and generate data graphs with up to 21M vertices, 40M edges, and 11 attributes per node. We transform the static graph into a temporal graph of T timestamps by randomly generating updates 4% the size of the graph (*w.r.t.* the number of edges) to create subsequent graph snapshots.

TGFDs **generation**. We generated 40 TGFDs (DBpedia, IMDb), and 20 TGFDs

(Synthetic) by using a discovery algorithm in our pilot study [89]. The pattern size, time intervals and size of TGFDs are reported in Table 2.2.

Comparative Baselines. We implemented the following.

(1) NaiveTED: We compute matches at each snapshot using the VF2 matching algorithm [59]. We verify matches between two snapshots if their time intervals lie within Δ.

(2) SeqTED: We implement a sequential TGFD error detection algorithm by running IncTED over a single machine. We compute matches over a single node by using an incremental matching for subgraph isomorphism algorithm, IsoUnit [51]. We run SeqTED on a Linux machine with AMD 2.7 GHz, 256GB RAM.

(3) GFD-Parallel: We implement the parallel GFD error detection algorithm [57]. For a set of GFDs Σ_{GFD} (defined in [6]), we compute matches for each $\varphi \in \Sigma_{\mathsf{GFD}}$ over each G_i, and check whether $G_i \models \Sigma_{\mathsf{GFD}}$ (pairwise matches between snapshots are not compared).

(4) GTAR-SubIso: Given a set of TGFDs Σ, we transform each $\sigma \in \Sigma$ to a GTAR. We remove the $X \rightarrow Y$ dependency, and define a Δt time interval $(0, q)$, for a pattern Q (serving as both the antecedent and consequent) containing the same constants for

each literal in $\bar{x}$. We evaluate this class of of subgraph isomorphism-based GTARs for its error detection accuracy and performance.

Error Injection, Parameters, and Metrics. We inject positive and negative errors according to varying error rates. For instance, for positive errors, we randomly select err% of pairwise matches with equal values in X, and update their values in Y to create violations, and add the pair to the set of positive errors Γ^+. For negative errors, we similarly pick err% of pairwise matches *w.r.t.* a TGFD σ, and update a value in Y to a value in the domain of X' *w.r.t.* another TGFD σ', where $\{Y \cap X' \neq \emptyset\}$, and add to Γ^-.

We use the following commonly adopted measures: precision= $\frac{|\mathcal{E}(\mathcal{G}_T, \Sigma) \wedge \Gamma^+|}{|\mathcal{E}(\mathcal{G}_T, \Sigma)|}$ and rec= $\frac{|\mathcal{E}(\mathcal{G}_T, \Sigma) \wedge \Gamma^+|}{|\Gamma^+|}$. The false positive rate is computed as fpr= $\frac{|\mathcal{E}(\mathcal{G}_T, \Sigma) \wedge \Gamma^-|}{|\Gamma^-|}$, and F_1= $2 \times \frac{\text{precision} \times \text{rec}}{\text{precision} + \text{rec}}$.

Table 2.2 summarizes the parameters and their default values.

Implementation. We implement all our algorithms using Java v.13 and Scala. We run the tests on a cluster of 16 Amazon EC2 Linux machines, each with 32GB RAM, 8 cores at 2.5 GHz. The full set of data constraints including TGFDs, source code, and datasets are available at [6].

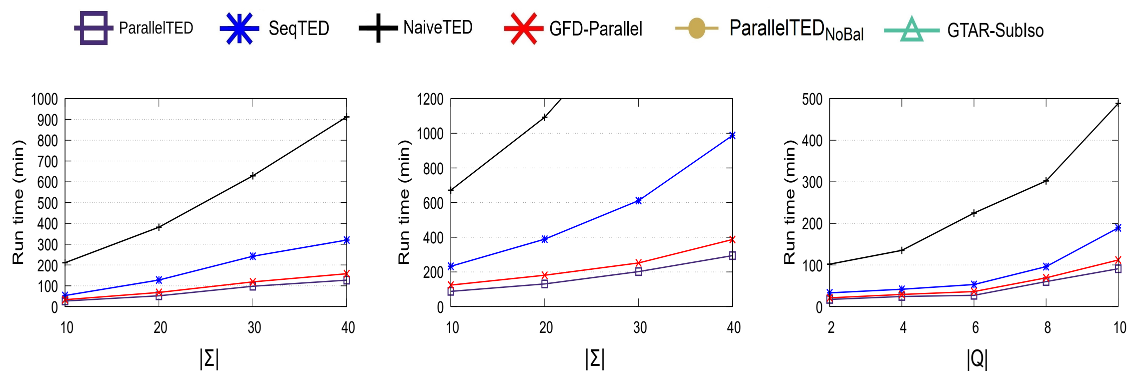
Table 2.2: Parameter values (defaults in bold)

Symbol	**Description**	**Values**
$\|\Sigma\|$	#TGFDs	**10**, 20, 30, 40 (DBpedia, IMDb) **20** (Synthetic)
$\|Q\|$	graph pattern size	2, 4, **6**, 8, 10
Δ	time interval	5, **10**, 15, 20, 25 month (IMDb)
T	total timestamps	5, **10**, 15, 20 (Synthetic)
$\|G\| = (\|V\|, \|E\|)$	graph size (in M)	**(5, 10)**, (10, 20), (15, 30), (20, 40)
chg	change rate	2%, **4%**, 6%, 8%, 10%
err($\mathcal{G}_T$)	error rate	1%, **3%**, 5%, 7%, 9%
n	#processors	2, **4**, 8, 16

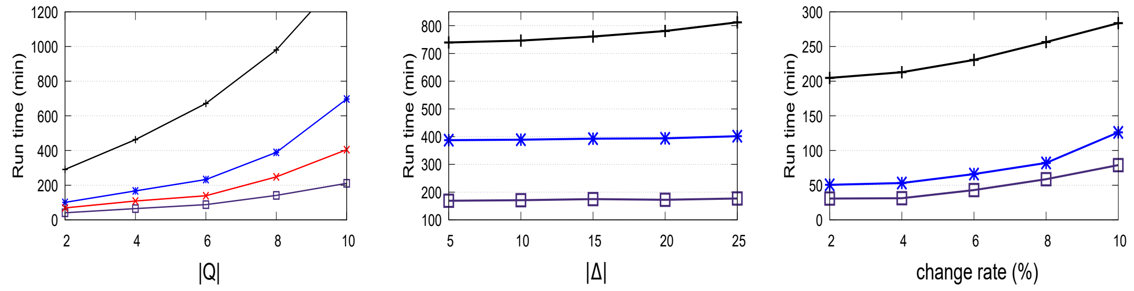
2.5.2 Exp-1: Scalability

Vary $|\Sigma|$: Figure 2.4a and Figure 2.4b show ParallelTED outperforming SeqTED by an average 170%, achieving larger gains in IMDb over a larger number of snapshots. We stopped executions of NaiveTED over IMDb after 22hrs. ParallelTED runs 22% and 29% faster than GFD-Parallel over DBpedia and IMDb, respectively, despite having to evaluate more matches. This demonstrates the effectiveness of IncTED, and our techniques for maintaining (query path) matches in the presence of changes.

Vary $|Q|$. Figure 2.4c and Figure 2.4d show that larger graph patterns incur higher runtimes as the cost of local pattern matching increases. This is especially evident for sequential algorithms NaiveTED and SeqTED. ParallelTED is 80% and 18% faster than SeqTED and GFD-Parallel, respectively.



(a) Vary $|\Sigma|$ (DBpedia) (b) Vary $|\Sigma|$ (IMDb) (c) Vary $|Q|$ (DBpedia)

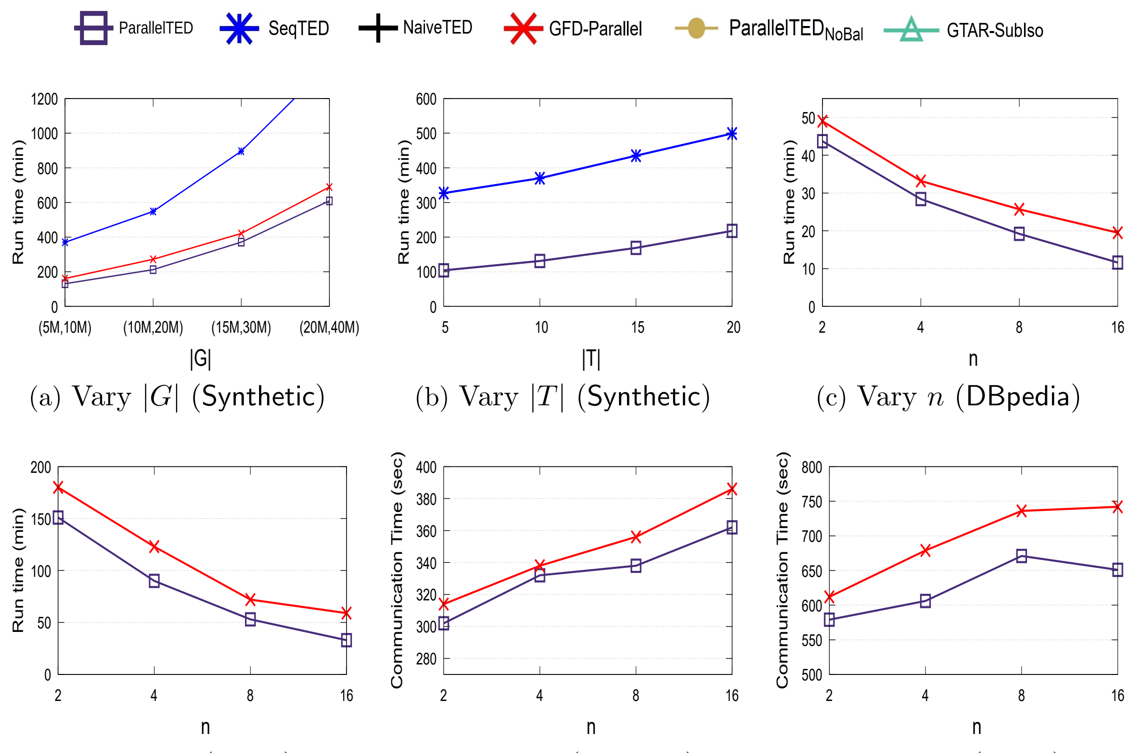


(d) Vary $|Q|$ (IMDb) (e) Vary Δ (IMDb) (f) Vary chg (DBpedia)

Figure 2.4: TGFD error detection efficiency and effectiveness (1).

Vary Δ. We vary Δ from 5 to 25 (Figure 2.4e), and found the performance of NaiveTED is sensitive due to the increased number of pairwise matches that need to be compared for larger Δ. In contrast, SeqTED and ParallelTED are largely insensitive due to their incremental checking strategies.

Vary chg. We vary the number of changes between snapshots, from 2% to 10%. Figure 2.4f shows that with more changes, all techniques incur longer runtimes, as expected. SeqTED and ParallelTED runtimes increase by 27% and 26%. respectively. SeqTED uses IsoUnit to compute matches which become more expensive as the number of changes localized to a subgraph increases. ParallelTED uses boolean vectors to track



(a) Vary $|G|$ (Synthetic) (b) Vary $|T|$ (Synthetic) (c) Vary n (DBpedia)

(d) Vary n (IMDb) (e) Vary n (DBpedia) (f) Vary n (IMDb)

Figure 2.5: TGFD error detection efficiency and effectiveness (2).

changes to existing matches to avoid isomorphic computations (Section 2.4.2), and achieves the fastest runtime.

Vary $|G|$. Figure 2.5a shows that for increasingly large graph sizes, sequential algorithms are not feasible, i.e., we stopped execution of SeqTED after 20hrs due to its exponential growth. However, ParallelTED shows that error detection and matching is feasible over large graphs, outperforming GFD-Parallel by 18%.

Vary T. Figure 2.5b shows that as we increase the number of snapshots T, the runtimes of SeqTED and ParallelTED scale linearly. With larger T, we expect more matches to occur (assuming a uniform spread of changes across all T timestamps), where validating a new match vs. existing matches in π_{XY} can be done in constant time.

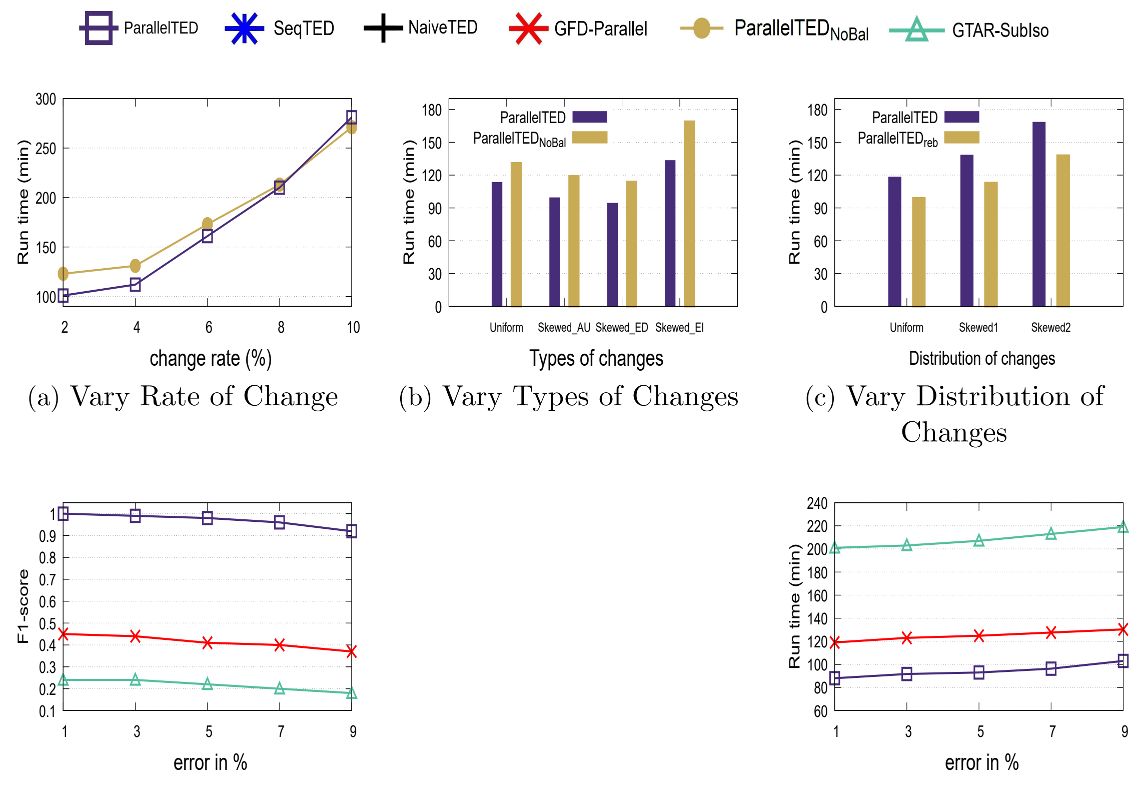
Vary n. Figure 2.5c shows that both ParallelTED and GFD-Parallel scale well with an increasing number of machine workers over the DBpedia dataset. ParallelTED runs 29% faster than GFD-Parallel despite evaluating matches both within and between graph snapshots (GFD-Parallel only compares matches within a snapshot). This highlights the efficiency of IncTED to avoid redundant pairwise comparisons, and adaptively tuning matches to avoid expensive subgraph isomorphism computations. Figure 2.5d shows that the same trend exists over the IMDb dataset, where ParallelTED runs 31% faster than GFD-Parallel.

Communication Overhead. We evaluate ParallelTED communication overhead for an increasing number of workers as shown in Figures 2.5e and 2.5f. Using DBpedia and IMDb, the communication cost comprises between 10-48%, and 5-28%, respectively, of the total error detection time. As expected, for an increasing number of workers, we see that both ParallelTED and GFD-Parallel incur increased overhead as the need to exchange data increases.

2.5.3 Exp-2: Adapting to Workload Changes

Rate of Change. We study the impact of the burstiness time buffer ζ by generating a graph of size (5M,10M), and implement a version of ParallelTED without ζ (no workload rebalancing) called $\text{ParallelTED}_{\text{NoBal}}$. Figure 2.6a shows ParallelTED runtimes with $\zeta = 0.1$ performs 16% faster as change rates less than 8% are accounted for by the burstiness buffer, requiring only at most two workload re-distributions. Beyond this change point, larger ζ values are needed, which require more frequent workload re-distributions (up to six), and incur approximately 6% overhead.

Vary Type and Distribution of Changes. Using the same generated graph, we distribute changes according to: (i) a Uniform distribution that assigns attribute updates (AU) (40%), edge deletions (ED) (30%), and edge insertions (EI) (30%); (ii) each of Skewed_AU, Skewed_ED, Skewed_EI assigns 85% of changes to their respective type, and 7.5% to the other two types. Figure 2.6b shows EI changes lead to the slowest runtimes as pattern matching becomes more expensive to find new matches. Attribute updates cause matches to be added/removed, and incur higher runtimes compared to ED, which only lead to fewer matches. Across all change types, Figure 2.6b shows ParallelTED is more efficient, about 20% on average, than $\text{ParallelTED}_{\text{NoBal}}$. We also evaluated data partitioning schemes that distribute the changes across the workers such that hot spots are: (i) uniform across four nodes; (ii) 80% of hot spots across two nodes; and (iii) 70% on a single node. As expected, ParallelTED incurs 13% more time, compared to $\text{ParallelTED}_{\text{NoBal}}$ at +19% (Figure 2.6c).



(a) Vary Rate of Change (b) Vary Types of Changes (c) Vary Distribution of Changes

(d) Vary *err* (IMDb) (e) Vary *err* (IMDb)

Figure 2.6: TGFD error detection efficiency and effectiveness (3).

2.5.4 Exp-3: Comparative Performance

Comparing with GFDs and GTARs. Figure 2.6d shows the comparative error detection F_1-score of ParallelTED, GFD-Parallel and GTAR-SubIso for varying error rates over IMDb. ParallelTED outperforms GFD-Parallel (resp. GTAR-SubIso) with an average gain of 55% (resp. 74%). With GFD-Parallel achieving 23% recall, TGFDs capture more errors “across” graph snapshots than GFDs. GFDs exhibit greater sensitivity to negative errors and false positives for increasing error rates as more match pairs are incorrectly detected as violations when temporal intervals are not considered.

In comparison to GTAR-SubIso, which achieved an average recall of 12%, TGFDs have greater expressive power with *e.g.,* variable and constant literals, to capture errors with non-zero lower bounds, and detect historical errors located in "past" matches. Figure 2.6e shows ParallelTED achieves 32% and 119% faster runtimes than GFD-Parallel and GTAR-SubIso, respectively. Overall, ParallelTED incurs a 4% loss in F_1-score at a 3% cost in runtime.

2.5.5 Case Study: Example TGFDs

We profiled DBpedia to identify TGFDs, and to validate their prevalence, and errors in real data [89, 15].

(1) TGFD 1 Figure 2.7(a) shows a sample pattern Q_{ts} for $\sigma_{ts} = (Q_{ts}[\bar{x}], (1, 365), [x.name, y.season, z.name] \rightarrow [w.actor]$, which specifies *"if a TV series in a given season is broadcast over a one-year period with a character role, then the actor playing this role must be unique."* Figure 2.7(c) shows a violation in *Season 3* of the series *Trollhunters: Tales of Arcadia*, where character *Jim Lake Jr.* was first played by *Anton Yelchin* in *May 2018*, and then by *Emile Hirsch* in *Dec. 2018.*

(2) TGFD 2 Figure 2.7(b) shows Q_{sp} for $\sigma_{sp} = (Q_{sp}[\bar{x}], (1, 4), [x.name] \rightarrow [y.league, z.team]$ that specifies *"for a given football player, who has played between one to four months in a given year, must play for the same team and league."* This requirement enforces player to team consistency for a given duration of time, and only permits team changes during authorized transfer windows in a year.

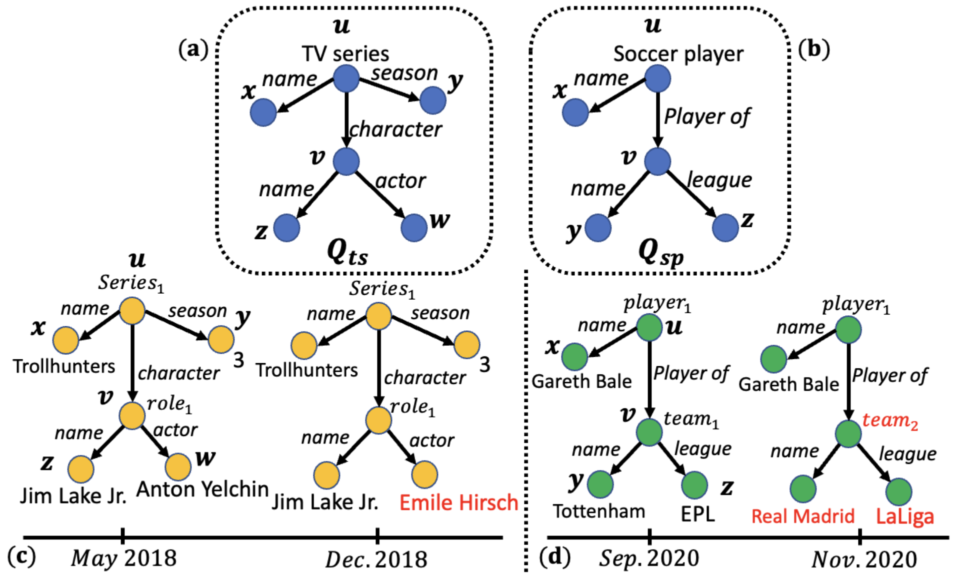


Figure 2.7: Case study: TGFDs in real-world graphs.

Figure 2.7(d) shows a violation for football (soccer) player *Gareth Bale* who played for the team *Tottenham Hotspur* on Sep. 2020, and then transferred to *Real Madrid* on Nov. 2020. The latter team change happened within four months, therefore leading to a violation. In both cases, existing graph constraints are unable to capture such inconsistencies either due to the lack of support to temporal constraints, or historical matches are not verified for data consistency, particularly with variable literals [57, 87].

2.6 Related Work

Graph Dependencies. Existing integrity constraints are mainly designed for static graphs. GEDs [53] extend GFDs by the support of literal equality of *entity id* over two nodes in the graph pattern to subsume GFDs and GKeys. Graph keys (GKeys) and their ontological variants are defined to uniquely identify entities [48, 83]. NGDs [52] are defined to extend GFDs with the support of linear arithmetic expressions and built-in comparison predicates. None of these dependencies are applicable to catch inconsistencies over temporal graphs. Hence, TGFDs are based on pairwise comparisons across timestamps satisfying the time duration bounds, which induce the scope of historical and future matches relative to a current time. Association rules GPAR [56] are defined over static graphs. GTARs are soft rules designed for predictive analysis, and are unable to capture inconsistencies, particularly rare occurrences, across a given time duration.

Temporal Dependencies. FDs over temporal databases include *Dynamic Functional Dependencies* (DFDs) that hold over consecutive snapshots (states) of a database, where attribute values from a current state determine values in the next state, e.g., a new salary is determined by the last salary and the last merit increase [75]. Wijsen et. al., define *Temporal Functional Dependencies* (TFDs) that rely on object identity, include a valid-time on tuples, and apply to the sequence of snapshots defining a temporal relation [117]. DFDs constrain pairs of adjacent states, while TFDs constrain a sequence of multiple valid-time states. TGFDs impose topological constraints that are not captured in relational settings, and TGFDs model a different matching and temporal semantics. DFDs and TFDs impose consistency over tuples from adjacent snapshots. TGFDs are not restricted to compare consecutive matches, but constrain the time difference of matches to lie within the Δ interval.

Second, TGFDs model the time interval duration when topological and attribute consistency is expected, whereas DFDs and TFDs do not model such semantics.

Temporal Graph Mining. Frequent dynamic subgraphs extend pattern mining over static graphs to dynamic graphs, which discover frequent patterns that occur beyond a given threshold over a time interval [33]. Mining temporal motifs capture dynamic interactions with patterns [63, 90], with extensions to time windows known as δ-temporal motifs [90]. Various mining algorithms have studied finding dense temporal cliques with partitioning schemes over temporal graphs such as GraphScope [108], MDL-based approaches to identify associations between changes [58], and mining for cross-graph quasi-cliques [92]. TimeCrunch associates each subgraph across time to a template structure, and uses clustering to stitch these temporal graphs together to create an entity trajectory [100]. While our work shares a similar spirit to identify consistent temporal patterns, TGFDs impose data dependency and temporal constraints over identified matches of a given pattern. We performed a pilot study for a special class of size-bounded TGFDs to assess their frequency in practice [89]. We validated the presence of these TGFDs in large-scale graphs such as DBpedia, as shown in the case study. In this work, we present a formal model, and complete study of general TGFDs, and propose new error detection algorithms.

Constraint-Based Graph Cleaning. There has been extensive work to infer missing data in graphs [91]. Graph Fact Checking rules [81] have been introduced to answer *true/false* to generalized fact statements. [105] proposed a constraint based approach to infer missing data in graphs. Graph identification [96] has been introduced to use probabilistic approaches to infer missing facts in knowledge graphs. Graph Quality Rules (GQRs) [54] are defined to deduce *certain* fixes over graphs by supporting conditional functional dependencies, graph keys and negative rules. These techniques are designed for static graphs, with a goal to suggest changes or provide useful provenance information based on enforcing a set of rules. TGFDs serve a different purpose to model time-dependent data consistency requirements over temporal graphs that are conditioned by a given time interval.

Parallel Algorithms for Subgraph Isomorphism. There has been an extensive line of research on parallel algorithms for subgraph isomorphism [98, 109] and SPARQL queries [60, 64, 71, 78]. In Trinity memory cloud [109], they used twig decomposition to prune the intermediate results in order to reduce the latency of the parallel algorithm of the subgraph isomorphism. In [98], they developed an in-memory algorithm to parallelize a backtracking procedure. They used the memory to distribute partial answers in a balanced way among threads for local expansion. Moreover, by replicating the partial answers to a global storage, they allowed a balanced workload distribution for the next round. For parallel SPARQL on RDF data, past techniques have used strategies such as hash-based partitioning, query decomposition and load balancing [60, 71]. Query decomposition is introduced in [64] to avoid communication cost by replicating graphs in the distributed setting. For multi-pattern matching, optimization techniques introduced in [78], by finding common sub-patterns and leveraging common matches to find the final set of matches. However, our algorithms are different from these techniques as TGFDs use general graph pattern and general property graphs. SPARQL-based techniques leverage RDF schema and query semantics, which are not available in a general graph pattern and property graphs.

2.7 Conclusion

We have proposed TGFDs, a class of graph dependencies to characterize errors induced by graph patterns and specific time intervals for temporal graphs. We have established complexity results for fundamental problems, and introduced a sound and complete axiom system to infer TGFDs. We introduced a parallel and incremental algorithm for TGFD-based error detection. Our experimental results have verified the effectiveness and efficiency of our error detection algorithms. As next steps, we intend to explore non-parametrized methods to extend our workload rebalancing scheme to dynamically adapt to workload burstiness. Given the utility of TGFDs, and the identified errors, graph data cleaning with respect to TGFDs would be an interesting next step.

Chapter 3

Discovery of Keys for Graphs

3.1 Introduction

Keys are a fundamental integrity constraint defining the set of properties to uniquely identify an entity. Keys serve an important role in relational, XML and graph databases to maintain data quality standards to minimize redundancy and to prevent incorrect insertions and updates. In addition, keys are helpful for deduplication (also referred to as entity resolution) and have been widely studied for entity identification [42, 13, 23]. While keys are often defined by a domain analyst according to application and domain requirements, manual specification of keys is expensive and laborious for large-scale datasets. Existing techniques have explored mining for keys in relational data (as part of functional dependency discovery) [72], and in XML data [34].

The expansion of graph databases has lead to the study of integrity constraints over graphs, including functional dependencies [53, 18], keys [48] and their ontological invariant [83]. The theoretical foundation of these constraints have been studied and there has been a wide application of key constraints for deduplication, citation of digital objects, data validation and knowledge base expansion [42, 68]. Graphs such as knowledge bases and citation graphs require keys to uniquely identify objects to ensure reliable and accurate deduplication and query answering. There is a need to automatically discover keys from such graphs as manual specification of keys is expensive and labor intensive. Although recent work has proposed techniques to find keys over RDF data [23], these techniques are not applicable for graphs as they do not support: (i) topological constraints; and (ii) recursive keys (a distinct feature in graph keys). Consider the following example on how keys help us to identify entities in a graph.

Example 11: Consider a knowledge graph consisting of triples (subject, predicate, object) where subject and object are nodes, and predicate is an edge connecting subject to object. Figure 3.1 shows a sample of such graph from the DBpedia dataset [79] of three colleges $\{college_1, college_2, college_3\}$, five cities $\{city_1, \ldots, city_5\}$, and three countries $\{country_1, country_2, country_3\}$ along with the attributes of each entity. Consider graph keys with patterns P_1 and P_2 in Figure 3.1. P_1 states that *if two colleges share the same name and motto, then they refer to the same college.* P_2 states that *if two colleges share the same name and city, then they refer to the same*

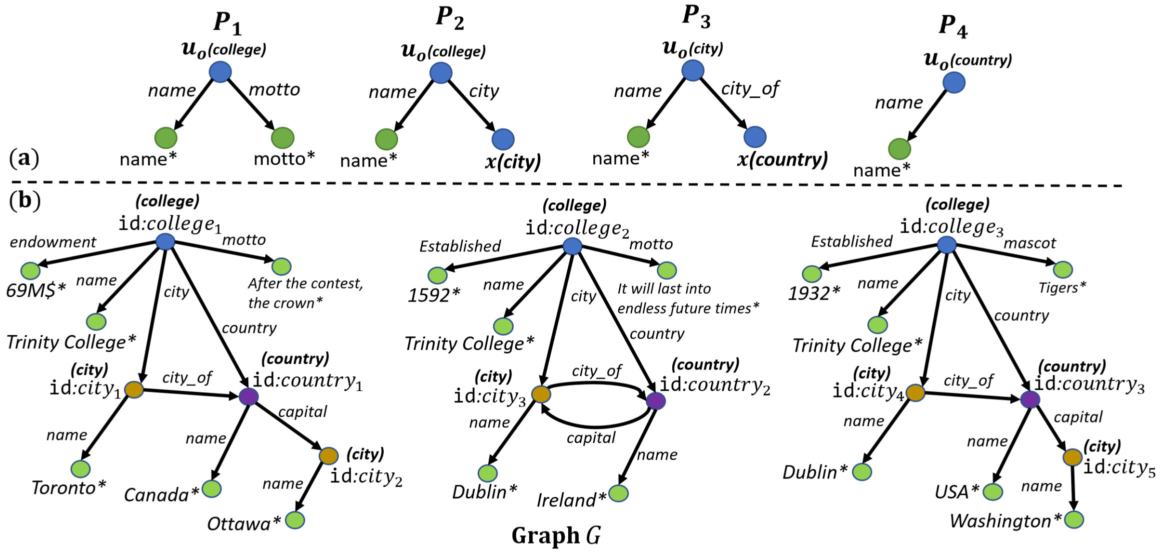


Figure 3.1: Sample graph from DBpedia.

college., Similarly, a city can be identified by its name and country as it is shown in P_3. Moreover, P_4 states that *a country can be identified by its name.* Note that P_2 is dependant to P_3 and P_3 is dependant on P_4, which reflects the recursiveness of graph keys [48]. However, one can confirm that not all combinations of the attributes can form a graph key for an entity. For example, *name* cannot uniquely identify *college* (resp. *city*) as we have three colleges (resp. two cities) with the same name. □

The example highlights that many keys are possible to identify entities, and this depends on the data and its semantics. For example, P_1 uses the *name* and *motto* to uniquely identify the college. However, due to missing values, not all colleges have motto. This leads us to null values for some colleges, thereby leading to poor support and representation across all colleges. This highlights the need to define meaningful properties for a key and an efficient algorithm to discover such keys over graphs. However, the main challenges to automatically discover keys in large scale graphs include: (1) How to reduce the search space? (2) How to generate the most likely candidate keys? (3) How to efficiently evaluate each candidate key?

Contributions. To address the challenges above, our contributions are as follows:

- We define new properties for graph keys, *support* and *minimality*, and formalize the discovery problem of graph keys (Section 3.4.1).
- We develop an algorithm called GKMiner with optimizations that mines graph keys (Section 3.4.2).
- We show that our algorithm is scalable and feasible to mine keys in graphs of millions nodes and edges. We show GKMiner runs up to six times faster with up to 61% gain in F_1-score than the existing technique that mines keys over the RDF data (Section 3.5).

The rest of this chapter is structured as follows. We discuss related work in Section 3.2, and preliminaries in Section 3.3. In Section 3.4, we provide key properties and then the discovery algorithm. We present our experimental evaluations in Section 3.5, and conclude in Section 3.6.

3.2 Related Work

Keys and Dependencies. Keys are defined to uniquely identify entities in a database. For relational data, keys are defined as a set of attributes over a schema [12], or by using unique column combinations [30, 115] to uniquely identify the tuples. For XML data, keys are defined based on path expressions in the absence of schema [34]. Traditional keys are also defined over RDFs [24, 93, 106] in the form of a combination of object properties and data properties defined over OWL ontology. Recent works have studied functional dependencies for graphs (GFDs) that define value constraints on entities that satisfy a topology constraint [53, 57]. Keys for graphs (GKeys) aim to uniquely identify entities represented by vertices in a graph, using the combination of recursive topological constraints and value equality constraints. GKeys are a special case of GFDs [48]. The recursiveness of GKeys makes it more complex compared to relational and RDF based counterparts. Graph matching keys, referred to as GMKs, are extension of graph keys using similarity predicates on values, and supporting approximation entity matching [39].

PG-Keys [19] proposes a modular and flexible model to formalise keys for property graphs. Their keys are defined to be used for a property graph query language that is currently underway through the ISO Graph Query Language (GQL) project. PG-Keys define keys that are applicable to nodes, edges, and properties in a property graph. However, they do not consider any topology constraints to define a key, while GKeys are focused to uniquely identify entities (*i.e.,* nodes) in the absence of schema. For the property graphs, a uniqueness constraint is a set of attributes whose values uniquely identify an entity in the collection. Neo4j keys [82] are based on uniqueness constraints and require the existence of such constraints for all vertices the graph. A new principled class of constraints called embedded uniqueness constraints have been proposed that separates uniqueness from existence dimensions and are used in the property graphs to uniquely identify entities [104]. However, GKeys are different than these constraints by supporting topology constraints via a graph pattern.

Dependency Discovery and Pattern Mining. Key mining approaches have been studied for relational databases as data-driven [67] and schema-based [103] techniques. TANE [72] proposed a level-wise schema-based approach to mine keys in relational data (as part of functional dependencies) and it has been extended for RDFs [24]. KD2R [93] extends the relational data-driven approach of [103] by exploiting axioms (such as the subsumption relation) and considers multi-valued properties. SAKey [110] extends K2DR by introducing additional pruning techniques to discover approximate keys with exceptions. VICKEY [111] has extended SAKey to mine conditional keys over RDFs. To avoid scanning the entire dataset, all three techniques (*i.e.,* K2DR, SAKey, and VICKEY) first discover the maximal non-keys and then derive the keys from this set. Non-keys are the set of attributes that are not keys and maximal non-keys are super-sets of all other non-keys. Instead of exploring the whole set of combinations of properties, the idea behind these techniques is to find those combinations that are not keys and then derive the keys from that set. To verify that a set of properties is a non-key, it suffices to find two subjects that share values for these properties.

Comparison with GFDs. Fan et. al, have developed a parallel algorithm to discover GFDs in graphs [50]. Although GKeys are a special case of GFDs, their technique is not able to mine GKeys as (1) the GFD discovery algorithm cannot mine recursive patterns; and (2) to model a GKey with a GFD, the GFD must have a pattern consist of two connected components to define equality over pairs of matches. However, the GFD discovery algorithm only mines GFDs with a single connected component pattern [50]. Therefore, there needs to be an extended semantics of GFDs to compare pair of matches in the discovery algorithm.

To the best of our knowledge, there is only one technique to discover keys for graphs [16], which is our preliminary work published at a VLDB workshop called Advances in Mining Large-Scale Time Dependent Graphs (TD-LSG). This work differs from [16] as we define new metrics to mine GKeys, and propose an efficient algorithm with optimizations and perform extensive experimental evaluations over real world graphs and compare with SAKey [110].

3.3 Preliminaries

Graphs. A *directed graph* is defined as $G = (V, E, L, F)$ with labeled nodes and edges, and attributes on its nodes. The set V is a finite set of vertices and L is a finite set of labels. A set of edges is denoted as $E \in V \times L \times V$, *i.e.*, $e = (u, l, v)$ represents an edge from u to v with the label l that is not equal to edge (v, l, u). Each node $v \in V$ may have a label $l \in L$ referred as $v.\mathsf{type}$. For a node v, $F(v)$ is a tuple to specify the set of attributes as $(A_1 = a_1, ..., A_n = a_n)$ of v. More specifically, A_i with a constant a_i determines the attribute A_i of v written as $v.A_i = a_i$. Attributes can carry the properties of a node such as *name*, *age*, etc., as found in social networks and knowledge graphs. We represent each attribute as a separate node with no type, *i.e.,* for each attribute $(A_i = a_i) \in F(v)$, there exists a node v_i with the value of a_i and there exists a corresponding edge $(v, A_i, v_i) \in G$.

Example 12: We return to Figure 3.1, where we have three colleges with unique ids $\{college_1, college_2, college_3\}$ and all of them have the property *name* = *Trinity College*. However, $college_1$ and $college_2$ have *motto*, while $college_3$ has *mascot*. Moreover, entities are connected to each other via edges, *e.g.,* $city_1$ with the *name* = *Toronto* has an edge *city_of* to $country_1$ named *Canada*. □

Graph pattern. A *graph pattern* is defined as a connected, directed graph $P(u_o) = (V_P, E_P, L_P)$ where (1) V_P is a finite set of pattern nodes; (2) E_P is a finite set of pattern edges; (3) L_P is a function which assigns a specific label $L_P(v)$ (resp. $L_P(e)$) to each vertex $v \in V_P$ (resp. each edge $e \in E_P$). Note that we consider graph patterns to be a general pattern (e.g., pattern with loop, tree, DAG, etc.).

The pattern nodes V_P may be one of three types: (1) a *center* node $u_o \in V_P$, representing the main entity to be identified; (2) a set of *variable nodes* $V_x \subseteq V_P$; and (3) a set of *constant nodes* $V_c = V_P \setminus (\{u_o\} \cup V_x)$. A variable node is being mapped to an *entity* and it carries the label as a type along with an eid, while a constant node only contains a value without any eid to map to a value.

Example 13: Consider the two entity patterns P_1 and P_2 in Figure 3.1(a), characterizing entities of type *college*. P_1 contains constant nodes *name* and *motto*, while P_2 has a constant node *name* and a variable node *city*. P_3 is a pattern for the entities of type *city* with constant node *name* and variable node *country*, while *country* has pattern P_4 with a constant node *name*. □

Graph pattern matching. Given two labels ι and ι' from L_P, we say ι matches ι', denoted as $\iota \asymp \iota'$ if either (1) $\iota = \iota'$; (2) $\iota =$ '_', *i.e.,* wildcard matches any label. Given a graph G and a pattern $P(u_o)$, a match h is a subgraph $G' = (V', E', L', F'_A)$, which is isomorphic to P, *i.e.,* there exists a bijective function h from V_P to V' such that (i) for each node $v \in V_P$, $L_P(v) \asymp L'(h(v))$; and (ii) for each edge $e(u, u') \in E_P$, there exists an edge $e'(h(u), h(u')) \in G'$ such that $L_P(e) = L'(e')$.

Example 14: Given pattern P_1 of Figure 3.1(a), we can find matches h_1 and h_2 in graph G of Figure 3.1(b), such that $h_1(college) = college_1$ and $h_2(college) = college_2$. $college_3$ is not a match of P_1 as there is no match for the node *mascot*. However, there exist three matches h_1, h_2 and h_3 for pattern P_2 in G for $college_1$, $college_2$ and $college_2$ respectively. Similarly, we have three cities $city_1$, $city_3$, $city_4$ matched with the pattern $P_3(city)$ in G and all three countries $country_1$, $country_2$ and $country_3$ are matched with pattern $P_4(country)$. □

Graph keys (GKeys). A key for a graph is defined using a pattern $P(u_o)$ for a designated entity u_o [48]. Given two matches h_1 and h_2 of $P(u_0)$ in graph G, (h_1, h_2)

satisfies $P(u_0)$ denoted as $(h_1, h_2) \models P(u_0)$, if (a)$\{\forall v \in V_x, h_1(v).\mathsf{eid} = h_2(v).\mathsf{eid}\}$; (b) $\{\forall v \in V_c, L(h_1(v)) \asymp L(h_2(v))\}$; and (c)$\{\forall e \in E_P, L(h_1(e)) = L(h_2(e))\}$; then $h_1(u_o).\mathsf{eid} = h_2(u_o).\mathsf{eid}$. This means the two matches refer to the same entity in G. We say a graph G satisfies a key $P(u_o)$, denoted as $G \models P(u_o)$, if for every pair of matches $(h_1, h_2) \in G$, we have $(h_1, h_2) \models P(u_0)$. Similarly, G satisfies a set of GKeys Σ denoted $G \models \Sigma$, if $G \models P(u_o)$ for each $P(u_o) \in \Sigma$. A GKey $P(u_o)$ is considered as a *recursive* key if it contains at least one variable $v \neq u_o$, otherwise, $P(u_o)$ is called a *value-based* key [48].

Example 15: Going back to Figure 3.1 and continuing with Example 14, a GKey $P_1(college)$ can uniquely identify $college_1$ and $college_2$ as they have different *motto*, despite the same *name*. $P_2(college)$ is a recursive GKey that can identify all three colleges. It is recursively dependant to *city* in GKey $P_3(city)$, while *city* is recursively defined via *country* of the GKey $P_4(country)$. Although $city_3$ and $city_4$ have the same name *Dublin*, but they belong to different countries *USA* and *Ireland*, respectively. With two levels of recursiveness (*i.e.,* from *college* to *city* and then from *city* to *country*), $P_2(college)$ is able to uniquely identify all three colleges in the graph G. □

3.4 Discovery of GKeys

In this section, we discuss the discovery problem for GKeys. The discovery problem is to find a set of GKeys for a given type u_o in an input graph G. Graph keys impose topological constraints along with attribute value bindings that are needed to identify entities. Existing works miss the topology and only discover keys as a set of attribute value that work over RDF data. While we mine keys by considering both topology and attribute values in the form of a graph pattern [48]. However, it is not desirable to mine all GKeys for u_o as a large amount of them are redundant and not meaningful. Mining meaningful keys in graphs relies on defining key properties independent of the application domain. We propose two key properties: *minimality* and *support*, and a key discovery algorithm over graphs. The algorithm calls itself recursively to mine recursive keys and keeps track of recursive calls to avoid falling into an infinite loop.

3.4.1 Key Properties

We now present our approach to mine all minimal GKeys Σ in G for a given entity type u_o such that $G \models \Sigma$. *Minimality* avoids mining redundant GKeys and reduces the discovery time. *Support* mines keys that satisfy a minimum number of instances in G. We define the key properties followed by the discovery algorithm.

GKey embedding. We say a GKey $P(u_o) = (V_P, E_P, L_P)$ is *embeddable* in another GKey $P'(u_o) = (V'_P, E'_P, L'_P)$, if there exists a subgraph isomorphic mapping f from V_P to a subset of nodes in V'_P that preserves node labels/values of V_P, and all the edges that are induced by V_P with the corresponding edge labels.

Minimality. A GKey $P(u_o)$ is minimal if there exists no GKey $P'(u_o)$ such that

$P'(u_o)$ is embeddable in $P(u_o)$. A set Σ of GKeys with $G \models \Sigma$ is minimal, if it does not contain any redundant GKeys. A redundant GKey $P(u_o)$ exists in Σ, if removing $P(u_o)$ from Σ results in a Σ' that is logically equivalent to Σ, *i.e.,* Σ' uniquely identifies the same entities as Σ in G.

Support. For a candidate GKey $P(u_o)$, we define support to represent the number of entities in the graph G that are uniquely identified by $P(u_o)$ over the total number of entities of type u_o. We define $|P(u_o)|$ as the total number of entities that are uniquely identified by $P(u_o)$. for a GKey $|P(u_o)|$ such that $G \models P(u_o)$, we define support as:

$$\mathsf{sup}(P(u_o)) = \frac{|P(u_o)|}{N} \tag{3.4.1}$$

Let N be the total number of instances with the type u_o in graph G.

k-bounded **GKeys.** For a given user defined natural number k, a GKey $P(u_o)$ is *k-bounded* if $\mathsf{size}(P(u_o)) \leq k$, where size is defined as:

$$\mathsf{size}(P(u_o)) = |E_p| + \mathsf{size}(P(v_P)) \ \forall v_p \in V_P \tag{3.4.2}$$

This equation counts the number of edges in the pattern $P(u_o)$ and in the pattern of all variable nodes *i.e.,* recursive GKey. To validate a recursive GKey, one must validate the matches of the recursive patterns[48]. A set Σ of GKeys is *k-bounded*, if each $P(u_o) \in \Sigma$ is *k-bounded*.

Problem statement. Given a graph G, a node type u_o, a support threshold δ, and a natural number k, mine all minimal *k-bounded* GKeys Σ of the node type u_o, such that for each GKey $P(u_o) \in \Sigma$, $P(u_o)$ has the minimum support δ in G.

3.4.2 Algorithm

For a given entity type u_o, the naive algorithm mines all frequent graph patterns centered by u_o and explores all combinations of variable and constant nodes in each pattern to verify whether they form a GKey. The naive approach leads us to explore a large search space, which is shown to be infeasible in real world graphs [50]. We introduce GKMiner, an efficient algorithm to mine all minimal GKeys in a graph. Our algorithm takes as input a graph G, an entity type u_o, a natural number k and a support threshold δ to discover GKeys. It proceeds in three steps: (a) Create a summary graph $\mathcal{S}$ to explore the structure of G. This will help us to prune nodes that cannot form a GKey based on the given threshold δ. (b) Create a lattice $\mathcal{L}$ of candidate GKeys from $\mathcal{S}$ that prunes further candidate GKeys. (c) Mine minimal *k-bounded* GKeys from $\mathcal{L}$ in a level-wise search. By traversing the lattice in a level-wise manner, we implement strategies to cut the space of candidates to prune redundant candidates (supersets of already discovered keys) from lower levels of the lattice. Our experiments show that our algorithm runs up to six times faster than SAKey, despite mining topological constraints of GKeys compared to the value constraint based keys mined by SAKey.

Summary Graph. As the first step of mining GKeys, we traverse G to create a summary graph $\mathcal{S}$ that reflects the structure of G. $\mathcal{S}$ provides an abstract graph of G, where: (1) nodes represent the entity types that exist in G, and (2) an edge between two nodes in $\mathcal{S}$ shows that there exists at least one edge between two entities with the corresponding types in G. $\mathcal{S}$ helps us to model the relationship between entity types in a smaller graph. $\mathcal{S}$ will be used to define graph patterns for candidate GKeys. $\mathcal{S}$ is built in $O(V + E)$ time and is an auxiliary data structure $\mathcal{S}(V_S, E_S)$, where V_S (resp. E_S) is a set of nodes (resp. edges) and have the following properties:

1. For each node type $t \in L$ in the graph G, there exists a node v_t in V_S.
2. For each node $v_t \in V_S$, v_t.count is the number of nodes in G of type t.
3. For each edge $e(u_1, l_e, u_2) \in G$:
   (a) if u_1 is of type t_1 and u_2 does not carry a type, *i.e.,* a constant node, then create an attribute A_e with the name l_e and without any value (*e.g.,* set value as $*$) and add to v_{t_1} in V_S. Increase the A_e.count by one (initial value is 0).
   (b) if u_1 is of type t_1 and u_2 is of type t_2, then add an edge $e(v_{t_1}, l_e, v_{t_2})$ to E_S and increase the e.count by one (initial value is 0).

Example 16: Figure 3.2(a) shows the summary graph generated for the graph G of Figure 3.1 where we have three entities of type *college*, five entities of type *city* and three of type *country*. Out of three colleges, one of them has the attribute *endowment*, one has *mascot*, two have *motto* and all three have attribute *name*. □

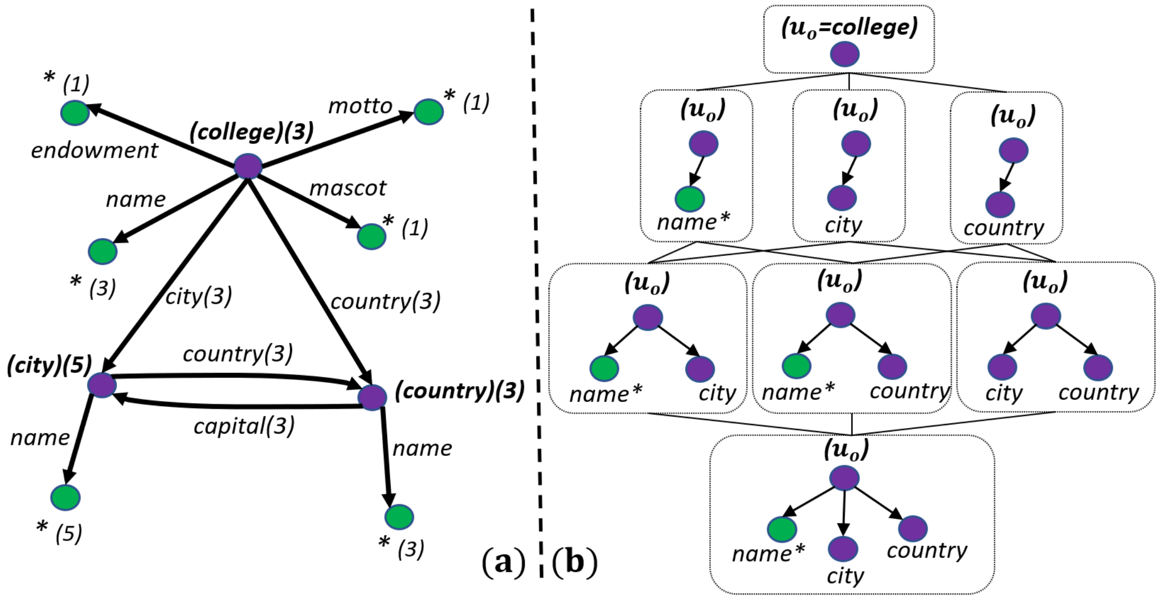


Figure 3.2: (a) Summary graph $\mathcal{S}$ of graph G of Figure 3.1 (b) Lattice for the type *college* based on $\mathsf{sup} = 75\%$

3.4.3 Pruning Strategy

After computing the summary graph, we prune the summary graph to find a set of attributes and variable nodes that meet the support threshold δ. For a given summary graph $\mathcal{S}(V_S, E_S)$, and a given center node v_{u_o}, we first prune the attributes of v_{u_o} based on δ. Following the support definition of Equation 3.4.1, we compute the support of an attribute A of v_{u_o} in V_S as following:

$$\mathsf{sup}(A) = \frac{A.\mathsf{count}}{v_{u_o}.\mathsf{count}} \tag{3.4.3}$$

This equation computes the support of an attribute A, if we add A as a singleton attribute in a candidate GKey. If we have $\mathsf{sup}(A) < \delta$, then adding A to any candidate GKey $P(u_o)$, makes $\mathsf{sup}(P(u_o)) < \delta$, hence $P(u_o)$ will not be a valid GKey. Therefore, we select a set of candidate attributes $\mathcal{A} = \{A_1, \ldots, A_n\}$ of v_{u_o} in V_S such that for each A_i, $\mathsf{sup}(A_i) \geq \delta$.

Similar to the Equation 3.4.3, we compute the support of the variable nodes, that are immediate neighbors of v_{u_o}. If v_{u_o} is connected to a node v with an edge e, then the support of v is computed as:

$$\mathsf{sup}(v) = \frac{e.\mathsf{count}}{v_{u_o}.\mathsf{count}} \tag{3.4.4}$$

Following the same reasoning of Equation 3.4.3, adding a variable node v with $\mathsf{sup}(v) < \delta$ to a GKey $P(u_o)$, makes $\mathsf{sup}(P(u_o)) < \delta$. Hence, we define a set of variable nodes $\mathcal{V} = \{v_1, \ldots, v_n\}$, where v_{u_o} is connected to each v_i and $\mathsf{sup}(v_i) \geq \delta$.

Lattice $\mathcal{L}$. For the entity type u_o, we create a lattice $\mathcal{L}(u_o)$ of candidate patterns based on the set $\mathcal{A}$ and $\mathcal{V}$ that are extracted from the summary graph $\mathcal{S}(V_S, E_S)$. $\mathcal{L}(u_o)$ is rooted at node u_o and expands level-wise based on the attributes in $\mathcal{A}$ and immediate variable nodes connected to v_{u_o} in $\mathcal{V}$. We create the lattice $\mathcal{L}(u_o)$ as follows:

1. Create a lattice $\mathcal{L}(u_o)$ rooted at node x of type u_o (level 0).
2. At the first level, we create a candidate GKey using the attributes and variable

nodes in $\mathcal{A}$ and $\mathcal{V}$. For each attribute $A_i \in \mathcal{A}$, we create a candidate GKey by connecting u_o to A_i with an edge labeled by the name of A_i. For each variable node $v_i \in \mathcal{V}$, we connect u_o to v_i with the corresponding edge label from $\mathcal{S}$.

3. At level l, we create a graph pattern for each l-combinations of the attributes and nodes in $\mathcal{A}$ and $\mathcal{V}$ respectively. Similarly, we connect u_o to each of the nodes with a direct edge and add the pattern to $\mathcal{L}$. A candidate pattern $P(u_o)$ of level $l-1$ is connected to a pattern $P'(u_o)$ of level l with a direct edge, if $P(u_o)$ is embedded in $P'(u_o)$.

4. Each pattern $P(u_o) \in \mathcal{L}(u_o)$ has a boolean flag $P(u_o)$.prune set by default to false. This flag helps us to mine minimal GKeys and prune the candidates in the lattice.

The lattice $\mathcal{L}(u_o)$ is created for the entity type u_o to generate candidate GKeys that initially meet the support threshold δ. However, since $\mathcal{L}(u_o)$ might contain other recursive entity types from the set $\mathcal{V}$, we need to create a lattice $\mathcal{L}(v_i)$ for each entity type $v_i \in \mathcal{V}$.

Example 17: Figure 3.2(b) shows the sample lattice created for the type *college* based on the summary graph of Figure 3.2(a) given the support threshold $\mathsf{sup} = 75\%$. If we calculate the sup for the attributes of the *college*, we have $\mathsf{sup}(name) = \frac{3}{3}$, $\mathsf{sup}(endowment) = \frac{1}{3}$, $\mathsf{sup}(motto) = \frac{1}{3}$, and $\mathsf{sup}(mascot) = \frac{1}{3}$. Based on the sup $=75\%$, we have $\mathcal{A} = \{name\}$. Similarly, if we compute the support of variable nodes connected to *college*, we have $\mathsf{sup}(city) = \frac{3}{3}$, and $\mathsf{sup}(country) = \frac{3}{3}$, leads us to have

$\mathcal{V} = \{city, country\}$. Using $\mathcal{A}$ and $\mathcal{V}$, we created the lattice in Figure 3.2(b), where we have three levels in the lattice with seven candidates GKeys. □

3.4.4 GKMiner Algorithm

GKMiner is a sequential GKey mining algorithm that traverses a lattice in a level-wise manner to mine all GKeys for a given type u_o. We first create the summary graph $\mathcal{S}$ from the input graph G. Next, we create the main lattice $\mathcal{L}(u_o)$ and traverse the lattice level by level to discover GKeys and prune redundant candidates containing already discovered, embedded keys. For each candidate $P_i(u_o)$ at level i, we check if it forms a GKey via incremental matching algorithm IsoUnit which enables localized subgraph isomorphism [55]. For each candidate $P_i(u_o)$ that has the prune flag equal to false, we first check size$(P_i(u_o))$ to ensure it is k-bounded. If size$(P_i(u_o)) > k$, then we set prune=true for all the descendant nodes of $P_i(u_o)$ in $\mathcal{L}(u_o)$. Next, we calculate sup$(P_i(u_o))$ by computing the matches as described in Section 3.4.1. If sup$(P_i(u_o)) \geq \delta$, then we report $P_i(u_o)$ as a GKey and prune its descendant nodes in $\mathcal{L}(u_o)$ to ensure the minimality of GKeys. However, if sup$(P_i(u_o)) < \delta$, we ignore $P_i(u_o)$ and continue with the next candidate.

Handling recursive GKeys. In the process of mining GKeys for the type u_o, if the candidate $P_i(u_o)$ contains a variable node of type t (*i.e.,* $P_i(u_o)$ is a recursive key), we first need to evaluate and find the GKeys for the dependant type t. To this end, we create the lattice $\mathcal{L}(t)$ and recursively call the GKMiner for the type t. We maintain a data structure called *dependency graph* $\mathcal{D}(V_D, E_D)$, where V_D is the set of nodes representing entity types and E_D is the set of edges to capture the dependencies between the types from the recursive calls. $\mathcal{D}$is being used to detect and avoid cycles in recursive calls as cycles lead us to fall into an infinite loop of recursive calls similar to deadlocks in process management [95]. Cycle happens when there exists a set of types that the GKey of each type is dependent to the GKey of another type in the cycle. Using dependency graph, we follow a cycle prevention strategy and avoid cycles in recursive calls [95]. To avoid such cycles, whenever we call GKMiner for the type t while mining GKeys for type u_o, we add u_o and t to V_D of $\mathcal{D}$and then add a direct edge (u_o, t) to E_D. In general, if adding an edge (t_i, t_j) leads us to have a cycle in $\mathcal{D}$, we break the cycle by removing the dependency (t_i, t_j). To this end, we remove t_j from the nodes in $\mathcal{L}(t_i)$. In this case, the GKeys of t_i won't be dependant to the GKeys of t_j.

Limitations. If the recursive calls in GKMiner algorithm form a cycle in the dependency graph, we used a heuristic approach to avoid such cycle by removing the last edge that created the cycle. However, all the edges that form the cycle are a candidate edge to be removed. Our method heuristically removes the last edge as we observed:

1. The last edge connects the node with the largest recursion depth to u_o and there exists a smaller chance for it to contribute in a recursive key for the given type.
2. If we expand the recursive GKey of u_o by substituting all the recursive nodes with the corresponding GKeys, then connecting the last node to u_o will not make

Algorithm 5: GKMiner (G, u_o, k, δ)

```
1  $\Sigma := \emptyset$; /* set of keys for each type*/
2  Initialize $\mathcal{D}(V_D, E_D) := \emptyset$ /*empty dependency graph*/
3  Initialize $\mathcal{S}(V_S, E_S) := \emptyset$ /*empty summary graph*/
4  foreach node $v \in G.V$ do
5      $t = v$.type;
6      if $t \neq$ null then
7          if $u_t \notin V_S$ then
8              add $u_t$ to $V_S$;
9          $u_t$.count $++$;
10 foreach edge $(v_1, l, v_2) \in G.E$ do
11     $t = v_1$.type;
12     if $v$.type == null then
13         if $l \notin F(u_t)$ then
14             add $A(l, *)$ to $F(u_t)$; /*add $l$ as an attribute of $u_t$*/
15         $A$.count $++$;
16     else
17         $t' = v_2$.type;
18         if $e(u_t, l, u_{t'}) \notin E_S$ then
19             add $e(u_t, l, u_{t'})$ to $E_S$;
20         $e$.count $++$;
21 Discovery ($G$, $u_o$, $\mathcal{S}$, $\mathcal{D}$, $\Sigma$, $k$, $\delta$, 0);
22 return $\Sigma$;
```

more matches for that GKey.

Considering these observations, we used the aforementioned heuristic approach to handle cyclic recursive GKeys.

Example 18: Going back to Figure 3.2(b) and assuming sup =60%, when we want to check a GKey for the type *college* that contains the type *city*, we find that there exists no GKey for *city* yet. Hence, we need to call GKMiner for *city* and we add an edge $(college, city)$ to the dependency graph $\mathcal{D}$ as shown in Figure 3.3(a). While mining

GKey for *city*, we need to call GKMiner for the type *country* and we add an edge $(city, country)$ to $\mathcal{D}$ in Figure 3.3(b). However, while mining GKeys for the *country* and as there is no GKey for the type *city* yet, we cannot call GKMiner for *city*. As shown in Figure 3.3(c), the edge $country, city$ makes a cycle in $\mathcal{D}$. Hence, we need to remove *city* from all the candidate for the type *country* and continue the mining to avoid cycle in $\mathcal{D}$. □

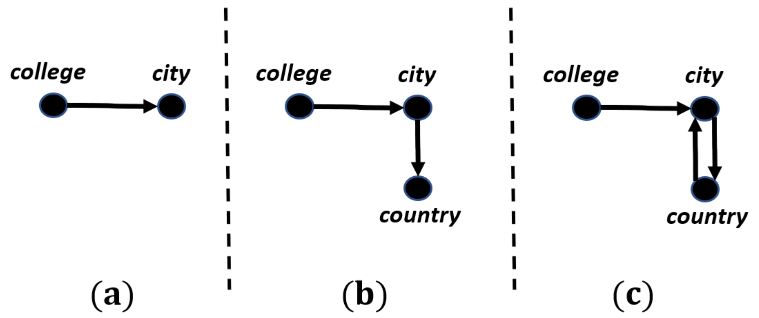


Figure 3.3: Dependency graph $\mathcal{D}$

Algorithm 5 provides the pseudo code of the GKMiner. After initialization (line 1-3), we create the summary graph $\mathcal{S}$ by iterating over the nodes and edges of G. For each node $v \in G.V$, we add a node of the corresponding type of v to $\mathcal{S}$ and maintain the count of the nodes (lines 4-9). Next, we iterate over the edges of G and add/maintain the edges and their count in $\mathcal{S}$ based on the type of the two end points of each edge in G (lines 10-20). After creating the summary graph $\mathcal{S}$, we call the Discovery algorithm and pass $\mathcal{S}$ and u_o along with other inputs to find GKeys.

The pseudo code of the Discovery algorithm is provided in Algorithm 6. It is a recursive algorithm to evaluate *k-bounded* GKeys for a given type. The algorithm takes as input the graph G, a center type u_o, summary graph $\mathcal{S}$, (initially empty)

Algorithm 6: Discovery $(G, u_o, \mathcal{S}, \mathcal{D}, \Sigma, k, \delta, \mathsf{size})$

```
1  $\mathcal{L}(u_o) := \mathsf{createLattice}(\mathcal{S}, u_o, \delta)$; /*create lattice for the given type $u_o$*/
2  foreach pattern $P(u_o) \in \mathcal{L}(u_o)$ do
3      if $P(u_o).\mathsf{prune} == \mathsf{false}$ then
4          if $|E_{P(u_o)}| + \mathsf{size} > k$ then
5              $P(u_o).\mathsf{prune} = \mathsf{true}$; continue;
6          if $P(u_o)$ contains type $t$ and $\Sigma[t] == \mathsf{null}$ then
7              add $(u_o, t)$ to $\mathcal{D}$;
8              if $\mathcal{D}$ has cycle then
9                  remove $(u_o, t)$ from $\mathcal{D}$;
10                 remove $t$ from $P(u_o)$ and $\mathcal{L}(u_o)$;
11             else
12                 Discovery $(G, t, \mathcal{S}, \mathcal{D}, \Sigma, k, \delta, |E_{P(u_o)}| + \mathsf{size})$;
13                 if $\Sigma[t] == \mathsf{null}$ then
14                     $P(u_o).\mathsf{prune} = \mathsf{true}$; continue;
15         $\mathcal{M} := \mathsf{IsoUnit}(G, P(u_o))$;
16         $\mathsf{computeUniqueEntities}(\mathcal{M}, P(u_o))$;
17         if $\mathsf{sup}(P(u_o)) \geq \delta$ then
18             $\Sigma[u_o].add(P(u_o))$; /*$P(u_o)$ is a valid GKey for $u_o$*/
19 return $\Sigma$;
```

dependency graph $\mathcal{D}$, (initially empty) set of keys, and three integers k, δ, and size. The value of size is initially set to 0 and it will be updated for the recursive calls to avoid mining recursive GKeys of size greater than k. We first create a lattice for the given type u_o (line 1). The function takes u_o, the summary graph $\mathcal{S}$ and δ as an input. It first computes the set of attributes $\mathcal{A}$ and variable nodes $\mathcal{V}$ from $\mathcal{S}$ based on the support parameter δ. It then creates the lattice based on the l-combinations of $\mathcal{A}$ and $\mathcal{V}$ at each level l. Next, we traverse the lattice in a level-wise manner and check whether each candidate pattern forms a GKey. Despite traversing the lattice level-wise, the candidate key patterns are of height one.. For each pattern that is not to be pruned (line 3), we first check if it is *k-bounded* (lines 4-5). If the pattern contains a recursive type t without a GKey, then we need to call the Discovery algorithm for

t. We first check if adding the edge (u_o, t) creates a cycle in $\mathcal{D}$. If so, we remove the edge from $\mathcal{D}$ and remove t from the pattern to avoid cycles in recursive calls (lines 7-10). Otherwise, we call the Discovery algorithm by passing t and the current size of the GKey (line 12). If we were not able to find a GKey for t, then we prune the pattern and its descendants (lines 13-14). After these steps, we find the matches of the pattern and compute the number of entities that are uniquely identified by the candidate GKey (lines 15-16). We add the pattern as a GKey for u_o if it meets the support threshold δ (lines 17-18). At the end, we return the set of keys that are found.

3.4.5 Optimizations

In this section, we propose an optimization for the GKMiner algorithm. While creating the summary graph $\mathcal{S}$, and as we check the existence of the attributes for each node in G, we maintain a hash-map of the values in the attribute domain. This helps us to find which values are unique for each specific attribute. For an attribute A, we hash the values $\{a_1, \ldots, a_n\}$, where a_i is the value of the attribute A for the node v_i in G. The result of the hash is a set of classes $\{\pi_1, \ldots, \pi_m\}$, where each π_j has one or more equal attribute values, assuming the collision is handled in the hashing process. If a value a_i uniquely exists in a class π_j, then a_i is a unique value for the attribute A among all the nodes that carry A. For each node $v_i \in G$, we maintain a bit vector flag called unique. We set $v_i.\mathsf{unique}(A) = \mathsf{true}$, if the corresponding value a_i is unique among all nodes that share the same type as v_i and carry attribute A. We can use the unique bit vector when computing the set of matches for $P(u_o)$. Assume $P(u_o)$ contains a set of constant nodes $\{v_{c1}, \ldots, v_{cn}\}$. For a match $h \in \mathcal{M}$, if we have $h(u_o).\mathsf{unique}(v_{ci}) = \mathsf{true}$ for any attribute v_{ci}, then h is uniquely identified by $P(u_o)$ without further exploration. If an attribute v_{ci} is unique for a node $h(u_o)$ in G, then any combination of the attributes that contains v_{ci} is unique for $h(u_o)$. Note that hashing is done in constant time. Hence, we maintain the unique bit vector for all the attributes in G while creating the summary graph $\mathcal{S}$ with the same time complexity $O(V + E)$.

3.5 Experiments

We have implemented the GKMiner algorithm in Java 17. We used the JGraphT [8] library to load input graph data and used the built-in VF2 implementation in the library for subgraph isomorphism. Besides the library, we have implemented the following indices for faster data access and query runtimes.

1. Index on node's URI: In all of our graphs, we used node's URI to access the nodes. The URI is considered as string and we created a hash-map based on the URI to access the nodes.
2. Attributes' name: For each node, we have created an index on the name of its attributes so we have constant time access to the attribute value by attribute name.

3. For each match of a given pattern, we have created an index from the nodes of the match to the corresponding node in the pattern for a faster access when checking the attribute values.

We use real world graphs to evaluate our algorithm on (1) the efficiency of GKMiner compared to the existing general rule-based mining approach SAKey [110]; and (2) the effectiveness of GKMiner for the task of data linking compared to SAKey.

Experimental Setup. We implement all our algorithms in Java v17, and ran our experiments on a Linux machine with AMD 2.7 GHz CPU with 128 GB of memory. Our source code and test cases are available online[1].

Datasets. We used three real graphs for our experiments.

1. DBpedia [79]: The graph contains in total 5.04M entities with 421 distinct entity types, and 13.3M edges with 584 distinct labels. DBpedia is extracted from the Wikipedia pages.
2. IMDb [5]: The data graph contains 6.1M entities with 7 types and 21.3M edges. This dataset contains information of the movies extracted from the IMDB website and in total we have 44.2M facts.
3. DBpediaYago [111]: This dataset contains entities from the DBpedia [79] and

[1]`https://github.com/mac-dsl/GraphKeyMiner.git`

Yago [84] datasets that are linked together. There exists a gold standard available for the entity links between these two datasets on the Yago Web page [10]. This dataset uses the ground truth to link the entities across the two knowledge bases. For each entity, we rewrite the properties of the entity in the Yago using its DBpedia counterparts.

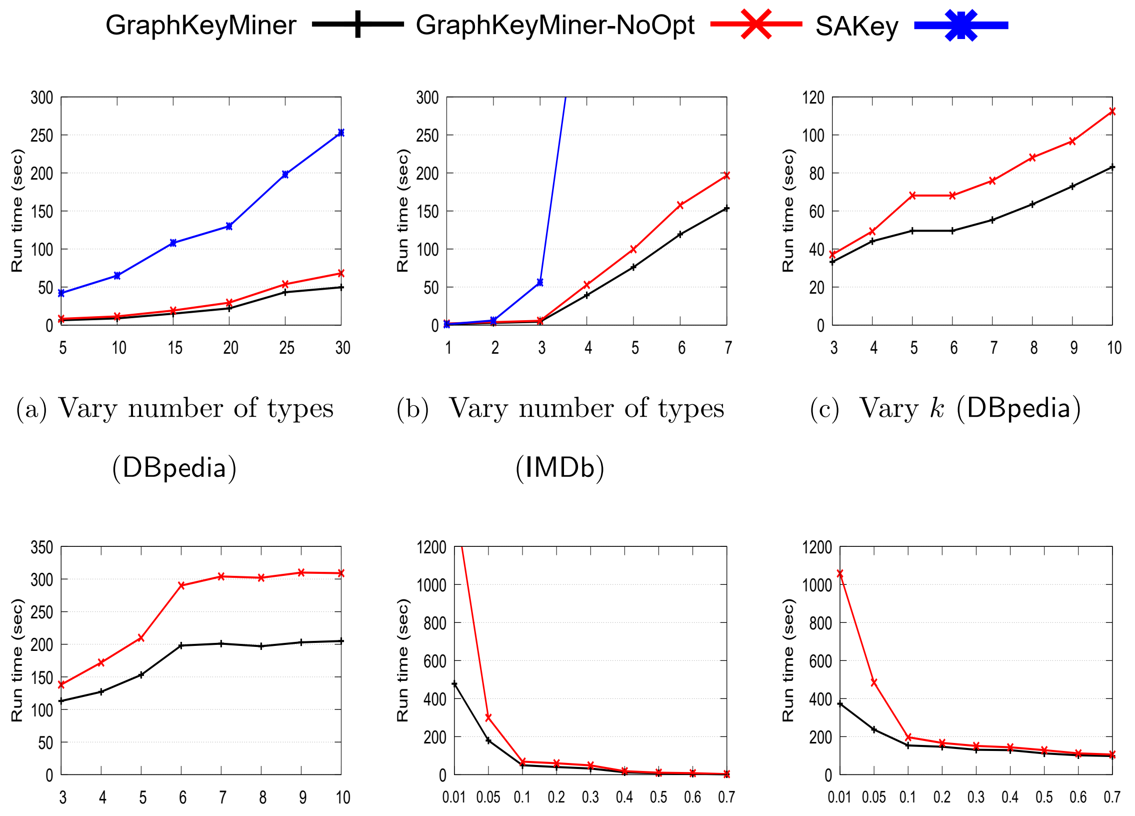
Algorithms. We implemented the following algorithms for the experimental evaluations.

1. GKMiner : Our mining algorithm of Section 3.4 with the optimization.
2. GKMiner-NoOpt : the GKMiner algorithm without the optimization and usage of the unique vector of Section 3.4.5.
3. SAKey [110]: Discovers maximal non-keys first and then derive the keys from this set. SAKey does not consider topological constraints to mine keys for graphs.

We excluded Vickey [111] from our tests as it mines conditional keys over RDFs. Vickey works on top of SAKey by first finding non-keys and then mines conditional keys. As we do not mine conditional graph keys, we do not compare the evaluation of our method with Vickey.

Experimental Results. Firstly, we evaluate the efficiency of GKMiner against GKMiner-NoOpt and SAKey. Next, we compare the quality of the mined keys in data linking using the DBpediaYago dataset with ground truth [83, 17].

Exp-1: Number of types. All three algorithms take an entity type as input and mine keys for that type. To compare the scalability of the algorithms, we vary the number of types and evaluate the runtime. Using DBpedia (resp. IMDb) dataset, we fixed the sup $= 10\%$ and $k = 5$ and vary the number of types from 5 to 30 (resp. 1 to 7). For SAKey, we set $n = 1$ to find exact keys as we do in GKMiner. Figure 3.4a shows the runtime of the three algorithms. GKMiner is on average 30% faster than GKMiner-NoOpt and 6 times faster than SAKey. This demonstrates the effectiveness of our method with optimizations over the existing method to find graph keys. We stopped executions of SAKey over IMDb after 120 minutes. SAKey was only able to finish mining keys for the types *distributor*, *genre*, and *country* which in total contain only 6.77% of the facts in the IMDb dataset. However, both GKMiner and GKMiner-NoOpt were able to mine GKeys for all types in less than 200 seconds.



(a) Vary number of types (DBpedia)

(b) Vary number of types (IMDb)

(c) Vary k (DBpedia)

(d) Vary k (IMDb)

(e) Vary sup (DBpedia)

(f) Vary sup (IMDb)

Figure 3.4: GKMiner efficiency.

Exp-2: Size of pattern. By fixing sup $= 10\%$, we varied the size of the pattern k from 3 to 10 over DBpedia and IMDb dataset on 30 and 7 types respectively. We excluded SAKey from this test as there is no pattern size on the keys that SAKey mines. Figure 3.4c shows the runtime over the DBpedia dataset, and here is our findings: (1) By increasing the value of k, the runtime increases as we have larger patterns to match against in G. (2) On average, GKMiner runs 33% faster than GKMiner-NoOpt due to an efficient approach to find unique values for the attributes, which helps to reduce the number of entities to be checked for the validity of each candidate GKey. The same trend exists in the IMDb dataset of Figure 3.4d, except the fact that the runtime does not increase for $k > 6$. We have only 7 types in this dataset and recursion depth (*i.e.,* the maximum diameter of the dependency graph) is limited compare to the DBpedia dataset with over 400 distinct types.

Exp-3: Support of GKey. In this experiment, we varied the sup value from 0.01 to 0.7 (*i.e.,* 1% to 70%) on the DBpedia and IMDb datasets with 30 and 7 types resp., and a fixed pattern size $k = 5$. The results are shown in Figure 3.4e for DBpedia and Figure 3.4f for the IMDb dataset. We also excluded SAKey from this test, as there was no option to control the support of a key mined by SAKey. The following is our findings: (1) By increasing the value of sup, the runtime decreases on both datasets as we have more pruning and fewer number of candidates need to be checked through the lattice. (2) On average, GKMiner runs 66% and 42% faster than GKMiner-NoOpt on DBpedia and IMDb respectively.

Exp-4: Effectiveness of GKMiner To investigate the quality of the GKeys, we compare the keys mined by GKMiner with the keys of SAKey in the application of entity linking. Primary application of keys is to link entities across two knowledge bases. If two entities are uniquely identified by a key in two different knowledge bases and they share the same attributes, then they refer to the same entity. For this test, we used DBpediaYago dataset with the available ground truth [111].

Table 3.1 shows the precision (P), recall (R) and F_1-score(F) measure of the entity linking task using keys mined by SAKey [110] against GKeys mined by GKMiner. Here are our findings: (1) The precision is always over 98% in both algorithms. (2) The

Entity Type	GKMiner	SAKey
(# triples)	P/R/F	P/R/F
Book(258.4K)	0.99/**0.07**/**0.13**	**1**/0.03/0.06
Actor(57.2K)	**1**/**0.36**/**0.52**	0.99/0.27/0.43
Museum(12.9K)	**1**/**0.21**/**0.34**	**1**/0.12/0.21
Scientist(258.5K)	**0.99**/**0.09**/**0.16**	0.98/0.05/0.11
University(85.8K)	**0.99**/**0.12**/**0.21**	**0.99**/0.09/0.16
Movie(832.1K)	**0.99**/**0.12**/**0.21**	**0.99**/0.04/0.08

Table 3.1: Comparative accuracy of GKMiner against SAKey

recall is low in some cases. This happens as we use a strict string equality when comparing the values of properties. Moreover, the incompleteness of the data in both Yago and DBpedia leads to lower recall as well. However, the use of recursive keys in GKMiner leads to an increase in recall. For example, for the class *Movie*, recall increases from 4% to 12% when recursive keys are considered. (3) On average, we observe an increase of 7 percentage points in recall, and of 9 points in F_1-score using GKMiner against SAKey. This shows the effectiveness of the GKeys mined by our proposed algorithm GKMiner when we consider recursive keys compared to the classical attribute based keys mined by SAKey.

3.6 Conclusion and Future Work

We proposed a new algorithm GKMiner to mine graph keys (GKeys) over real world graphs that is efficient and scalable. We introduced the notion of minimality and support for GKeys and adapt GKMiner for early termination and pruning of candidate keys. As next steps, we intend to extend GKMiner to mine conditional GKeys and study the the application of conditional GKeys to data linking, and the parallel discovery of GKeys in distributed graphs. We also intend to study the foundations analysis, including implication of GKeys, and an algorithm to compute the minimal cover for GKeys based on our implication analysis.

Chapter 4

Ontology-based Entity Matching in Attributed Graphs

4.1 Introduction

Keys are a fundamental integrity constraint defining the properties to uniquely identify an entity. Keys serve an important role in relational and XML databases during database design, normalization, and query optimization where they are commonly used for object reconciliation, to minimize redundancy, and to improve query runtimes. All these benefits transfer to graphs, where key constraints have been studied for entity identification [13, 34, 93, 23, 48]. The application of keys to graphs extends beyond deduplication to include emerging knowledge fusion [42] and fact checking [81].

Keys for graphs are inherently more complex than their relational counterparts due to the absence of schema, variances in topology and node types, and they may be *recursively defined*. Keys for graphs incorporate a topological constraint expressed by a graph pattern Q to uniquely identify entities. For example, keys for XML data identify duplicate entities via regular paths [34]; and for graphs, key constraints are posed on node matches induced by subgraph isomorphism [48]. Existing work has studied the theoretical foundations and applications of key constraints, and their generalized counterpart, graph functional dependencies [68, 57, 119]. These constraints have been applied to data cleaning [66], data validation [22], and entity matching [48].

Entities in real-world knowledge graphs often contain heterogeneous labels and multiple attributes. This poses two challenges for entity matching over graphs: (1) nodes that should refer to the same entity may not be captured by key constraints that only enforce label equality [88]; and (2) nodes with equal labels that match key patterns may not necessarily refer to the same entity, due to differing attributes. Furthermore, such graphs are often interpreted with respect to (w.r.t) an ontology that provide domain specific concepts and relationships, defining semantic equivalence among node labels. Consider the following example.

Example 19: Consider a knowledge graph G consisting of triples (subject, predicate, object) where subject and object are nodes, and predicate is an edge connecting subject to object. Figure 4.1 illustrates a fraction of DBpedia G, with three subgraphs describing three music entities $\{v_1, v_2, v_3\}$, where each node has an associated type denoted in parentheses. For example, entities *omg_mike* and *omg* in v_1 and v_3,

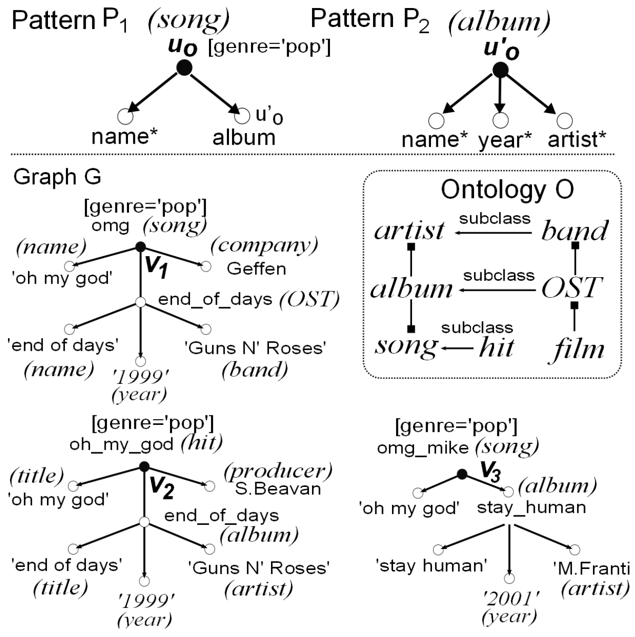


Figure 4.1: Entity matching with ontologies.

respectively, are both of type *song*.

Consider graph keys φ_1 and φ_2 depicted as graph patterns P_1 and P_2 in Figure 4.1. φ_1 states that "*if two songs share the same name and album, then they refer to the same song*". Similarly, an album can be identified by its name, year of release and artist, characterized by φ_2. Note the dependence of φ_1 on φ_2, to identify a song, we need to first identify its artist, reflecting the recursive property of graph keys [48]. Applying φ_1 and φ_2 via subgraph isomorphism on G, we obtain only v_3 as a match, since (1) v_1 *end of days* is of type *OST* (rather than *album*); (2) v_2 is of type *hit* rather than *song*; and (3) the v_1 album predicate *band* fails to match the required label *artist*. Hence, φ_1 and φ_2 fail to identify v_1 and v_2 as the same song as they rely only on label matching.

Given an ontology O, as shown in Figure 4.1, we exploit ontological relationships and semantic equivalance to extend graph keys. For example, we recognize an *OST* is a type of *album* participating in a hyponym (subClassOf) relationship. Similarly, we note that labels *artist* and *band*, and types *hit* and *song* are semantically similar. By extending φ_1 and φ_2 with these ontological equivalences, we identify v_1 and v_2 are indeed the same song.

Not all labels in O are useful. Ontological similarity is often characterized within a scope such that concepts that are 'far apart' in O (*w.r.t.* a distance function) are not conceptually close. For example, if we replace the *OST* entity with a *film* entity to create an entity v_4, this will not match φ_2 using a distance threshold of two, since the distance between *album* and *film* in O exceeds this bound. □

Beyond entity deduplication, ontological extension of graph keys enrich the neighborhood of equivalent entities. For example, merging v_1 and v_2 yield two new edges for the *song* entity: one with *company Geffen*, and the second with *producer S. Beavan*, completing the *song* information. These semantic extensions have widespread applications to link prediction, learning and inference for knowledge base completion [77, 61, 81], and knowledge fusion [42].

The above example highlights the need for a new class of dependencies for graphs that go beyond existing subgraph isomorphism to consider entity matching with ontological similarity. We extend existing notions of keys for graphs, which uniquely identify an entity, to exploit the relationships among node labels given in an ontology.

The recursive property of graph keys allows us to precisely define related entities for the keys. By incorporating ontologies, we further increase the scope of entities that can be matched to recursive keys to include matches that are ontologically similar. While ontological extensions have been studied for traditional functional dependencies [35, 27], little work has been done to enrich graph keys with ontologies.

Contributions. We extend keys for graphs with ontological pattern matching. We define the semantics, and study the foundations of these new ontological key constraints, and demonstrate their practical applications.

(1) We propose *Ontological Graph Keys* (OGKs), a new class of key constraints that exploit ontologies to enhance keys for graphs. An OGK includes an event pattern that defines an entity u, which is used to identify similar concepts w.r.t. an ontology O. We characterize entity equivalence by matching: (i) pairs of equivalent subgraphs w.r.t. the key constraints, which may be recursively defined; and (ii) value constraints defined on the node attributes.

(2) We formally introduce the entity matching problem using OGKs. Given a set of OGKs Σ, a *scope* (G, O, θ) that consists of graph G, ontology O, and a matching cost threshold θ to ensure semantic closeness, the problem is to compute an equivalence relation R, such that the quotient graph induced by R of G satisfies Σ. While this problem is NP-complete, we introduce efficient algorithms to enforce Σ.

(a) To characterize the matching process, we revise the Chase process of conventional data dependencies for OGKs, by incorporating ontology matching that trigger a sequence of non-destructive "merge" operations over equivalent entities. We show that the Chase with OGKs satisfies the *Church-Rosser* property, i.e., Chase sequences are finite and terminating, resulting in a unique graph satisfying the OGKs.

(b) We define *early terminating* criteria for the revised Chase, and the corresponding entity matching algorithms over recursively defined OGKs. Our dynamic programming algorithm consists of two efficient phases: (i) a top-down phase that decomposes OGKs to smaller, tree constraints to refine matches, and perform early validation; and (ii) a bottom-up synthesizing phase that assembles the matches, and induced entity equivalence classes for recursive entity matching. We develop optimization techniques to prune unpromising matches that reduce the verification cost.

(c) Given limited resources for entity matching, we propose a practical variant of the Chase that includes a cost model for matching and editing entities in G. We compute a Chase sequence that minimizes the cost under budget B, enforcing OGKs that tend to merge highly similar entities. We develop an anytime algorithm that can be interrupted to return the Chase sequence identified thus far, with tunable memory.

(3) We experimentally verify the efficiency and effectiveness of our OGK entity matching algorithms using two real-world graphs. We compare against two existing baselines, demonstrate the improved efficiency of our techniques, and our ability to identify semantically equivalent entities that are ignored by existing solutions. We show how these missed entities enable entity fusion over disparate knowledge bases, and facilitate knowledge base completion.

4.2 Ontological Graph Keys

We provide definitions, and introduce ontological graph keys.

4.2.1 Preliminaries

Graphs. We consider directed, attributed graphs $G = (V, E, L, F_A)$, where V is a set of nodes, and $E \subseteq V \times V$ is a set of edges. For each node $v \in V$ (resp. edge $e \in E$), $L(v)$ (resp. $L(e)$) is a *type* (resp. a relation) from a finite alphabet τ. For each node v, its value is denoted as $v.\mathsf{val}$. The $v.\mathsf{val}$ is an example of an *attribute* of v, describing a node property. For each node v, its attributes $A_i \in \mathcal{A}$, $i \in [1, n]$ are captured in its *property tuple*, $F_A(v)$, defined as a sequence of attribute-value pairs $\{(v.A_1, a_1), \ldots (v.A_n, a_n)\}$. Each pair $(v.A_i, a_i)$ states that the attribute $v.A_i = a_i$.

Entity identifiers. To define the mapping between a node v and a real-world entity

u, we introduce entity identifiers. Given a set of entities $\{u_1, \ldots, u_m\}$, we associate a unique entity identifier eid_i to entity u_i ($i \in [1, m]$). Each node v carries a (possibly empty) list of *entity identifiers* $\{v.\mathsf{eid}_1, \ldots, v.\mathsf{eid}_m\}$, For each $\mathsf{eid}_i \neq \mathsf{Null}$, this indicates that v encodes an instance of entity u_i. We enforce two types of node equality: (1) two nodes v and v' are *value equivalent* if $v.\mathsf{val} = v'.\mathsf{val}$; and (2) v and v' are *entity equivalent w.r.t.* entity u (with entity identifier eid_u), if $v.\mathsf{eid}_u = v'.\mathsf{eid}_u$.

Remarks. Nodes in G may encode a *node identifier*, and an *entity identifier*(eid) to distinguish different nodes and different entities, respectively. In real world graphs, a single node may model instances of two different entities, and similarly, two distinct nodes may model the same instance of an entity. These specifications often occur in multi-typed entities, which are common in property graphs [20], knowledge bases [70] and social networks [80]. Existing keys for graphs only enforce node identity, and do not differentiate between entity vs. node identifiers [34, 48].

Ontologies. An ontology is a directed graph $O = (V_o, E_o)$, where V_o is a set of *concept labels* and $E_o \subseteq V_o \times V_o$ is a set of semantic relations among the concept nodes. In practice, an edge $(v, v') \in E_o$ may encode three types of relations [76]: (a) *equivalence*, which state that v and v' are semantically equivalent, representing relations such as "refers to" or "known as"; (b) *hyponyms* that state v is a kind of v', modeling "is-a" or "'subClassOf" relations that define a preorder over V_o; and (c) *descriptive*, which state that v is described by v' in terms of 'association' or 'part-of' relations. In practice, an ontology may encode a taxnonomy, thesauri, or RDF schema. By incorporating ontologies, OGKs are more expressive than traditional graph keys [48], and capture semantic similarity relations during entity matching.

Example 20: We return to Figure 4.1, where entities $v_1 - v_3$ all have the property *genre* = *pop*, but v_1 and v_3 are both of type *song*, and v_2 is of type *hit*. By using the ontology O associated with G, we expand the notion of similarity to include semantic relationships among the node labels. For example, *OST* is a subclass of *album* via a hyponym edge, and *band* is semantically similar to *artist*. □

Relevant set. Given an ontology O and a concept label l, the *relevant set* to l refers to the set of concepts similar to l in O, denoted as $\mathsf{lsim}(l)$, according to a distance function $\mathsf{dist}(\cdot)$. Formally, $\mathsf{lsim}(l)$= $\{l' | \mathsf{dist}(l, l') \leq \alpha\}$, where $\mathsf{dist}(\cdot) : V_o \times V_o \rightarrow [0, 1]$ computes the distance between l and l', and the threshold α defines the scope of similarity. Possible definitions of $\mathsf{dist}(l, l')$ include the normalized sum of the edge weights along the shortest, undirected path between l to l' in O [70, 118]. To differentiate among the relations in O, we can assign weights w_1, w_2, w_3 to the edges representing equivalence, hyponym, and descriptive relations, respectively [76].

Example 21: For ontology O in Figure 4.1, we set the weights $w_1 = 0.1$, $w_2 = 0.3$ and $w_3 = 0.6$ to represent the relative cost among the ontological relations. The $\mathsf{dist}(album,\ OST) = 0.3$, since there is a path length 1 between these two concepts sharing a hyponym relation. Similarly, $\mathsf{dist}(film, OST) = 0.6$, for a descriptive relation. Given threshold $\alpha = 0.3$, the relevant set to OST is $\mathsf{lsim}\ (OST) = \{OST, album\}$. □

Entity patterns. An *entity pattern* $P(u_o)$ is a connected general graph (V_P, E_P, L_P) containing a set of pattern nodes V_P, and pattern edges E_P. Each pattern node $u \in V_P$ (resp. pattern edge $e \in E_P$) has a label $L_P(u)$ (resp. $L_P(e)$). The pattern nodes V_P may be one of three types: (1) a designated *center* node $u_o \in V_P$, representing the primary entity to be identified; (2) a set of *variable nodes* $V_x \subseteq V_P$; and (3) a set of *constant nodes* $V_c = V_P \setminus (\{u_o\} \cup V_x)$. Note that the entity pattern could be in any forms *e.g.,* general pattern with loop, tree, DAG, etc.

Example 22: Consider the two entity patterns P_1 and P_2 in Figure 4.1, characterizing instances of *song*, and *album*, respectively. P_1 contains a constant node *name*, and a variable node *album*. Intuitively, the equivalent instances of *song* should be *recursively* determined by the equivalent instances of *album* as defined by pattern P_2. □

Matching cost. To identify entities in a graph G that match an entity pattern $P(u_o)$, we must define a mapping function from nodes and edges in $P(u_o)$ to those in G. Formally, a *matching* between $P(u_o)$ and G is an *injective* function f from V_P to V, such that, for each node $u \in V_P$, $L(f(u)) \in \mathsf{lsim}(L_P(u))$ (concepts in G are similar to the concept modeled by u), and if $(u, u') \in E_P$, then $(f(u), f(u')) \in E$.

We quantify the matching cost by applying the the principle of spreading activation [97], which propagates concept relevance by following links of semantic networks to quantify concept closeness. We treat a pattern node u_o with concept label l as a "compound" concept, characterized by its neighboring pattern nodes. Given a matching f and entity pattern $P(u_o)$, the *matching cost* of f is quantified by the distance between u_o and $f(u_o)$, which is defined as

$$c(u_o, f(u_o)) = \frac{1}{|V_P|} \sum_{u' \in V_P} c_r(u', f(u'))$$

where $c_r(u', f(u'))$ is the *relative cost* of matching u' with $f(u')$ *w.r.t.* u_o, and is computed as

$$c_r(u', f(u')) = \begin{cases} \beta^{d_{u'}} \cdot \mathsf{dist}(L_P(u'), L(f(u'))) & u' \notin V_x \\ \beta^{d_{u'}} \cdot c(u', f(u')) & u' \in V_x \end{cases}$$

Here $d_{u'}$ is the distance between u' and u_o in $P(u_o)$ (treated as an undirected graph), $\beta \in [0, 1]$ is a decay factor, and dist computes the distance of concept labels in O. When $u' = u_o$, $c_r(u', f(u))$ is simply $\mathsf{dist}(L_P(u_o),\ L(f(u_o)))$. Intuitively, $c(u_o, f(u_o))$ simulates a partial spreading activation focused on u_o, by aggregating the propagated cost ("dissimilarity") between each pattern node and its matches to u_o. When u' is a variable node, the cost is aggregated by recursively expanding the pattern(s) modelling u'.

Example 23: Consider $w_2 = 0.3$, and decay factor β=0.9. (1) Given a matching f between P_2(*album*) in G such that f(*album*) = *end_of_days(OST)*, f(*name*)=*oh my god(name)*, f(*year*) = *1999(name)*, and f(*artist*) = *Guns N' Roses(band)*, (a) For constant nodes *name* and *year* in P_2, the matching cost c_r(*name*, *oh my god(name))* = c_r(*year*, *1999(year))* = $0.9^1 * 0 = 0$ (the concept labels *name* and *year* are omitted in ontology O); and c_r(*artist*, *Guns N' Roses(band))* = $0.9^1 * 0.3 = 0.27$, due to matching *band* to *artist* via the subclassOf relation. (b) c(*album*, *end _ of _ days(OST))* is thus

Notation	Description
G=(V, E, L, F_A)	attributed graph G
$P(u_o)$=(V_P, E_P, L_P)	entity pattern $P(u_o)$; u_o: center node
$V_x \subseteq V_P$; $V_c \subseteq V_P$	variable nodes V_x; constant nodes V_c
$Q(u_o, G)$	query answer of Q in G
$c(u_o, f(u_o))$, $c_r(u, f(u))$	matching cost & relative cost
$\varphi(u_o) = (P(u_o), X)$	ontological graph key with literals X
(G, O, θ)	scope of OGKs with cost bound θ

Table 4.1: Summary of notation.

computed as $\frac{1}{4}$ $(0.9^0{*}0.3{+}0.9{*}0{+}0.9{*}0{+}0.9{*}0.3) = 0.14$, where $0.9^0{*}0.3$ is the relative matching cost from *album* to *end_of_days(OST)*. (2) Similarly, given a match between P_1(*song*) and G that matches *song* to v_1, *name* to *oh my god(name)* and *album* to *end_of_days(OST)*, c(*song*, *omg*) $= \frac{1}{3}(0.9^1{*}0{+}0.9^1{*}0{+}0.9 * 0.14){=}0.042$. That is, the matching cost of a *song* depends on the propagated cost from its relevant variable and constant nodes. □

To identify matches of $P(u_o)$ in G w.r.t. ontology O, where the matching cost is within a given threshold θ, we define the notion of a matching *scope*, (G, O, θ). A matching of $P(u_o)$ under scope (G, O, θ) is an *injective* function f from V_P to V, such that: (i) for each node $u \in V_P$, there exists a *node match* $f(u) \in V$ where $L(f(u)) \in \mathsf{lsim}(L_P(u))$; (ii) for each edge $e_p{=}(u, u') \in E_P$, there exists an *edge match* $e = (f(u), f(u')) \in E$; and (iii) $c(u_o, f(u_o)) \leq \theta$. A *match* of P in G induced by f, denoted as $P(G, f)$, is the induced subgraph of G with nodes and edges from matching function f. We summarize the main notations in Table 4.1.

4.2.2 Ontological Graph Keys

We extend keys for graphs with ontologies, and present their semantics, matching criteria, and properties. Lastly, we highlight their relationship to existing graph dependencies.

An *ontological graph key* (OGK) $\varphi(u_o)$ for an entity u_o is a pair $(P(u_o), X)$, where $P(u_o)$ is an entity pattern with center node u_o that is associated with a unique identifier eid_o, and X is a set of literals. Each literal $l \in X$ is either a constant literal of the form of $u.A = c$ (for a constant c), or a variable literal $u.A = u'.A'$, where u and u' are two nodes in $P(u_o)$, and A and A' are node attributes from $\mathcal{A}$.

Semantics. Given an OGK $\varphi(u_o) = (P(u_o), X)$, and scope (G, O, θ), a matching function f *satisfies* X, denoted as $f \models X$, if (i) $f(u_o) \neq \emptyset$ under threshold θ; and (ii) for each constant and variable literal in X, $f(u).A = c$ and $f(u).A = f(u').A'$, respectively.

Ontological Bisimilarity. Let $\varphi(u_o) = (P(u_o), X)$ be an OGK defined on the entity u_o with identifier eid_o. We define the criteria to identify equivalent entities. We say two matches $P(G, f_1)$ and $P(G, f_2)$ are *bisimilar* under scope (G, O, θ), denoted as $P(G, f_1) \sim P(G, f_2)$, if the following hold: (a) $f_1 \models X$, and $f_2 \models X$; and (b) for each pair of nodes (v_1, v_2) where $f_1(u) = v_1$ and $f_2(u) = v_2$, (i) if u is a constant node, then v_1, v_2 are *value equivalent*, i.e., $v_1.\mathsf{val} = v_2.\mathsf{val}$; or (ii) if u is a variable node, then v_1, v_2 are *entity equivalent*, i.e., $v_1.\mathsf{eid}_u = v_2.\mathsf{eid}_u$ (eid_u is the entity identifier of u).

A scope (G, O, θ) *satisfies* φ, denoted as $(G, O, \theta) \models \varphi$, if and only if for every pair of matches $P(G, f_1)$ and $P(G, f_2)$ are bisimilar ($P(G, f_1) \sim P(G, f_2)$), and the matches are entity equivalent *w.r.t.* eid_o ($f_1(u_o).\mathsf{eid}_o = f_2(u_o).\mathsf{eid}_o$) Intuitively, OGK $\varphi(u_o)$ enforces the requirement that "*all nodes participating in bisimilar matches satisfying* X *under given scope* (G, O, θ) *should refer to the same entity as* u_o".

OGK **Properties**. We introduce two properties of OGKs.

Non-trivial. An OGK $\varphi = (P(u_o), X)$ is *non-trivial*, if $P(u_o)$ contains u_o, and at least one variable or constant node ($|V_P| \geq 2$), and X is *satisfiable.* To define satisfiability, we first define the closure of X (denoted as $\mathsf{cl}(X)$) as all literals that can be derived via transitivity of the equality relation. We can perform a fixed point inference that includes X in $\mathsf{cl}(X)$, and then adds $v.A{=}v'.A'$ to $\mathsf{cl}(X)$ if $v.A = c$ and $v'.A'{=}c$ in $\mathsf{cl}(X)$, or $v.A{=}v''.A''$ and $v''.A''{=}v'.A'$ are both in $\mathsf{cl}(X)$, until $\mathsf{cl}(X)$ no longer changes. We say X is satisfiable if there is no pair ($v.A{=}c$, $v.A{=}c'$) in X such that $c \neq c'$.

Well-defined. A set of OGKs Σ is *well-defined*, if for every OGK $\varphi(u_o) = (P(u_o), X)$ in Σ, and every variable node u' in $P(u_o)$, there exists an OGK $\varphi(u') \in \Sigma$. Henceforth, we consider well-defined Σ that contain nontrivial OGKs. We denote $\varphi(u_o)$ as φ, when u_o is clear.

Example 24: The two pattern constraints in Figure 4.1 extend to two OGKs: (1) $\varphi_1 = (P_1(\mathit{song}), \mathit{song.genre} = \mathit{pop})$, and (2) $\varphi_2 = (P_2(\mathit{album}), \emptyset)$. φ_1 states that

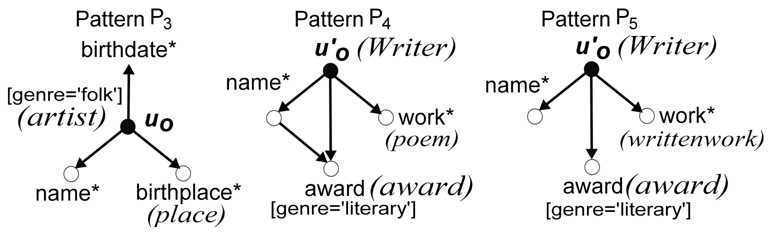


Figure 4.2: Ontological Graph Keys

if two nodes *ontologically* match *song*, with the same song name, and refer to the same *album* entity, then they are equivalent songs. Figure 4.2 shows three additional OGKs φ_3, φ_4 and φ_5, with entity patterns $P_3(artist)$, and P_4, P_5 referring to entity *Writer*, respectively. When φ_2 and φ_3 both exist in Σ, the *artist* node in P_2 necessarily becomes a variable node. □

Relationship to other dependencies. We highlight the relationship between OGKs and other graph dependencies. For an OGK φ with $\theta = 1$, ontology O is $\emptyset$ (f only enforces label equality), and all eids refer to node identifiers, φ can be considered as: (i) a general case of keys for graphs that only enforces X [48]; (ii) a special case of graph functional dependencies (GFDs), by "duplicating" its pattern P to $P(u_o)$ and $P'(u'_o)$ via graph isomorphism, and by enforcing attribute equality on eid (*i.e.,* u_o.eid $=$ u'_o.eid) [57]; and (iii) a special case of graph entity dependencies (GEDs), which subsume GFDs, using more general graph homomorphism [53].

Example 25: OGKs φ_3 and φ_4 in Figure 4.2 define constraints for *artist* and *Writer*, respectively. Consider nodes v_4, v_5 and v_6, all referring to the name *Bob Dylan*, where (v_4, v_5) are entity equivalent *w.r.t.* *artist*, and (v_5, v_6) are entity equivalent *w.r.t.*

writer. Entity v_5 is both a *Writer* and *artist*, thereby enforcing eid_{artist} and eid_{writer}. Existing graph keys that enforce only node identity are unable to differentiate distinct entities encoded within a node. □

4.3 Entity matching with OGKs

We apply OGKs to entity matching, and introduce dynamic matching that extends the Chase process [12] over graphs.

Entity graphs. We define the notion of an entity (hyper) graph that identifies nodes from a graph $G = (V, E, L, F_A)$ referring to the same entity in each hyperedge. Formally, given a set of entities $\mathcal{E} = \{u_1, \ldots, u_m\}$, and scope (G, O, θ), the *entity graph* $V_{\mathcal{E}}$ is a *hypergraph* $(V, \bigcup_{u \in \mathcal{E}} V[u])$, where $V[u]$ the quotient set of V induced by the entity equivalent relation $R(u)$. Two nodes, $(v, v') \in R(u)$ if and only if $v.\mathsf{eid}_u = v'.\mathsf{eid}_u$; (v, v') are entity equivalent *w.r.t.* u. We can obtain a *base graph* G' of $V_{\mathcal{E}}$ that models entity equivalence for each pair of nodes in the hypergraph. We obtain G' by enforcing entity identifier equivalence for each pair of nodes in $V[u] \in V_{\mathcal{E}}$ in the original graph G.

The Chase for OGKs. We characterize entity matching by extending the Chase to an entity graph $V_{\mathcal{E}}$. Consider a set of OGKs, Σ defined on a set of center entities $\mathcal{E} = \{u_1, \ldots, u_m\}$, and scope (G, O, θ). Intuitively, given an initial hypergraph containing singleton nodes in a hyperedge (for each entity u), the $\mathsf{Chase}(\Sigma, G)$ continually merges edges containing entity equivalent nodes, according to $\varphi \in \Sigma$, until no further changes are induced by φ. Specifically, we start with an initial hypergraph $V_{\mathcal{E}}^0$, where each hyperedge $[v]_u$ in $V_{\mathcal{E}}^0$ is a singleton $\{v\}$ for every entity $u \in \mathcal{E}$. Given a triple $(\varphi, (v, v'))$, where $\varphi = (P(u), X)$, for $\varphi \in \Sigma$, f (resp. f') are two ontology matchings of $P(u)$, $v = f(u)$, $v' = f'(u)$, and $P(G, f) \sim P(G, f')$, a Chase *step* of G by $(\varphi, P(G, f), P(G, f'))$ at a hypergraph graph $V_{\mathcal{E}}^i$ is

$$V_{\mathcal{E}}^i \overset{(\varphi,(v,v'))}{\Longrightarrow} V_{\mathcal{E}}^{i+1}$$

Specifically, the following two Chase rules must be satisfied:

(1) if $v.\mathsf{eid}_o$ is not in $f_A(v)$, create a new equivalence class $[v]$ with $v.\mathsf{eid}_u = c_u$, where eid_u is the entity identifier for $u \in \mathcal{E}$ with a unique value c. (Inclusion of new eids.)

(2) for all existing equivalence classes, $[v]_u$ and $[v']_u$ in $V_{\mathcal{E}}^i$ for entity u, merge $[v]_u$ and $[v']_u$ to a single equivalence class $[v]_u$, and update the set of edges $E_{\mathcal{E}}$ accordingly. (Merge hyperedges with entity equivalent nodes.)

By following these rules, the $\mathsf{Chase}(G, \Sigma)$ algorithm will generate a sequence of Chase steps that induce a sequence of hypergraphs $\{V_{\mathcal{E}}^0, \ldots, V_{\mathcal{E}}^n\}$. The $\mathsf{Chase}(G, \Sigma)$ *terminates* if there exists no $(\varphi, (v, v'))$ that elicits changes to $V_{\mathcal{E}}^n$ (no new eids, nor merges to enforce entity equality). Intuitively, $\mathsf{Chase}(G, \Sigma)$ verifies whether a set of nodes match the same pattern node via bisimilar matches. Since the matching process reduces to a Boolean function to determine whether two nodes a and b match pattern node u (via bisimilarity), a transitive closure holds over $\mathsf{Chase}(G, \Sigma)$. [1]

Example 26: Figure 4.3 illustrates a fraction of the entity graphs induced by two Chase steps over graph G in Figure 4.1 (with changed fraction marked in red). The first Chase step (φ_1, *end_of_days(OST)*, *end_of_days(album)*) creates three equivalent classes in $V^1_{\mathcal{E}}$ that merges equivalent constant nodes *e.g., 1999*, and a pair of equivalent *OST* and *album* entities. The second Chase step enforces OGK φ_2, and further merges equivalent *song* and *hit* entities given the equivalent classes in $V^1_{\mathcal{E}}$, and yields $V^2_{\mathcal{E}}$. □

We now introduce Lemma 3, which verifies that Chase with OGKs under scope (G, O, θ) preserves the *Church-Rosser property.* That is, all Chase sequences are terminating, and all terminating Chase produce the same $V^n_{\mathcal{E}}$.

Lemma 3: *Given scope* (G, O, θ)*, (1) chasing with any set of* OGKs Σ *is finite and has the Church-Rosser property; and (2) any terminating* Chase *guarantees* $(G', O, \theta) \models \Sigma$*, where* G' *is the base graph of* $V^n_{\mathcal{E}}$ *when the* Chase *terminates.* □

Entity matching with Chase. Given scope (G, O, θ), and OGKs Σ, defined on a set of entities $\mathcal{E}=\{u_1, \ldots, u_m\}$, the entity matching problem is to compute the entity graph $V_{\mathcal{E}}$ induced by a terminating Chase sequence $\mathsf{Chase}(G, \Sigma)$.

[1] "If a and b match pattern node u via bisimilar matches (i.e., a and b are equivalent), and b and c match u via bisimilar matches (b and c are equivalent), then a and c match u via bisimilar matches (a and c are equivalent)".

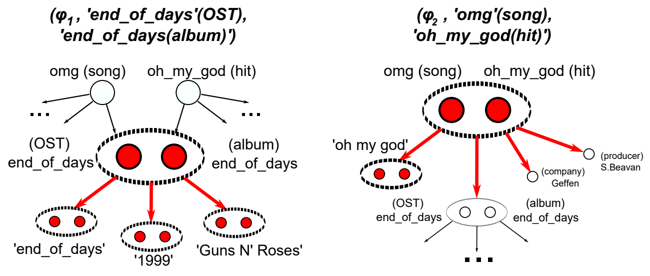


Figure 4.3: Entity graphs induced by two Chase steps.

This problem is, not surprisingly, NP-hard, as the validation for OGKs alone is already NP-hard. Even for small OGKs (subgraph isomorphism test is in polynomial time), there are $O(|\Sigma|N^2)$ pairwise comparison of matches, where N is the maximum number of subgraph isomorphism for a given pattern in OGK, and can be up to $(2^{d+1}-1)^{\frac{|V|}{d+1}}-|V|$ for G with $|V|$ nodes [31]. We show that the number of comparisons can be significantly reduced by pruning matches given existing equivalent classes (Section 4.4).

4.4 Entity Matching Algorithm

A naive approach to entity matching enumerates all possible pairs $(\varphi, (v, v'))$ to evaluate a Chase step. This is clearly infeasible for large G. Given the recursive nature of OGKs, an OGK cannot be enforced before all variable nodes are resolved. In this section, we introduce the OGK-*Entity Matching* (OGK-EM) algorithm that avoids exhaustive match enumeration of candidate pairs, and provides early termination.

Our OGK-EM algorithm focuses *early Chasing with non-recursive (sub)keys*. These

Chase sequences (involving constant nodes) can be enforced directly, once bisimilar matches are identified, or help to prune matches for their recursive counterparts. Moreover, such sequences can be computed independently without being "blocked" by dependent entities, and thus can be computed as early as possible.

4.4.1 Dependency Graph

To model interactions among OGKs, OGK-EM maintains an auxiliary structure called *dependency graph*. Given a set of OGKs Σ, we define a dependency graph $G_\Sigma = (\Sigma, E_\Sigma)$, where (i) each node represents an OGK $\varphi(u_o)$ in Σ, and (ii) there exists an edge $(\varphi(u_o), \varphi'(u'_o)) \in E_\Sigma$ if $u'_o \in V_x$, where V_x is the set of variable nodes in $P(u_o)$. For each OGK $\varphi = (P(u_o), X)$, OGK-EM maintains: (a) match set $P(u, G)$ for each pattern node u in $P(u_o)$; (b) $V[u_o]$, a partition (hyperedge) of $P(u_o, G)$ induced by relation $R(u_o)$; and (c) a pair of boolean flags (isC, Val) that is set to true when, respectively, φ contains only constant nodes, and $(G', O, \theta) \models \varphi$ given all enforced Chase steps thus far.

For ease of presentation, we construct a directed acyclic graph (DAG) G_d from G_Σ by collapsing each strongly connected component (SCC) into a single SCC node. Abusing terms from trees, we say a *root* (resp. *leaf*) of G_d is a node without an incoming (resp. outgoing) edge. We set the constant node flag, isC = true for all leaves in G_d. Moreover, for each node v_d in G_d, a rank $r(v_d)$ is defined as: (i) $r(v_d)$ = 0 if v_d is a root; (ii) $r(v_d)$ = $\max(r(v'_d))$+1 otherwise, where v'_d ranges over the parents of v_d in G_d; and (iii) for an SCC node v_d, all nodes in v_d have the same rank

$r(v_d)$.

4.4.2 Matching Algorithm

$\mathsf{OGK\text{-}EM}$ initializes entity graph, $V_{\mathcal{E}}$, with over estimated equivalence classes, where each class represents the set of nodes having concept labels l in the relevant set of entity node $u_i \in \mathcal{E}$, $l \in \mathsf{lsim}(l_{u_i})$. We dynamically refine these classes by retaining true ontological matches and splitting equivalence classes, with two major phases. First, a "top-down" *decomposition* phase decomposes each OGK to a set of *tree keys* (entity keys with tree patterns). We efficiently chase bisimilar tree matches to refine $V_{\mathcal{E}}$ by merging equivalence classes whenever possible. Second, in a "bottom-up" *synthesizing* phase, $\mathsf{OGK\text{-}EM}$ refines $V_{\mathcal{E}}$ by assembling bisimilar matches from the leaves of the dependency graph G_d, until all the nodes in G_d are processed.

Algorithm Phases. The main phases of $\mathsf{OGK\text{-}EM}$ is illustrated in Figure 4.4. It first constructs G_d and computes the node ranks. The decomposition and partial validation phase proceeds in ascending node rank order similar to a breadth-first traversal of G_d. The synthesizing phase starts at the leaf level proceeding in descending node rank order.

(1) *Top-down Decomposition.* For each root $\varphi = (P(u_o), X)$ in G_d, $\mathsf{OGK\text{-}EM}$ initializes the match sets $P(u, G)$ for each node u in $P(u_o)$ as $\{v|L(v) \in \mathsf{lsim}(L_P(u))\}$. It initializes $V[u_o]$ as $\{[v]|[v] = \{v\}\}$ for each $v \in P(u_o, G)$. The procedure Decomp

```
Algorithm OGK – EM
Input: a set of OGKs $\Sigma$ over entities $\mathcal{E}$, scope $(G, O, \theta)$;
Output: the entity graph $V_{\mathcal{E}}$ induced by Chase$(\Sigma, G)$.
1.  initializes $V[u_i]$ $(u_i \in \mathcal{E})$; integer $i$=0;
2.  construct $G_d$; integer $r_m$:=max$(r(v_d))$ $(v_d$ in $G_d)$;
3.  while $(i < r_m)$ do /* "top-down" phase*/
4.    $\Sigma_i := \{\varphi | r(\varphi) = i\}$;
5.    for $(\varphi \in \Sigma_i)$ do
6.      $\mathcal{P}_\varphi :=$ Decomp$(\varphi)$; /*spawning tree keys*/
7.      $P(u_o, G) :=$ PVal$(\mathcal{P}_\varphi)$; $i := i + 1$; /*partial validation;*/
8.  while $(i >= 0)$ do /* "bottom-up" phase*/
9.    $\Sigma_i := \{\varphi | r(\varphi) = i\}$; $i := i - 1$;
10.   for $\varphi(u_o) \in \Sigma_i$ do
11.     $V[u_o]$:= SVal$(\varphi, G_d)$;
12. $V_{\mathcal{E}} := \bigcup_{u_i \in \mathcal{E}} V[u_i]$;
13. return $V_{\mathcal{E}}$;
```

Figure 4.4: Algorithm OGK-EM

decomposes φ to a set of *tree keys* $\mathcal{P}_\varphi = \{\varphi_1, \ldots, \varphi_n\}$. Each tree key $\varphi_i = (P_i(u), X_i)$ contains: (i) a tree-structured pattern $P_i(u)$ centered on entity u; and (ii) $X_i \subseteq X$ with literals involving pattern nodes in P_i only. Intuitively, these tree patterns constitute a tree cover of $P(u_o)$ with shared pattern node u_o, and $\bigcup_i^n X_i{=}X$.

(2) *Partial Validation.* Given a set of tree keys $\mathcal{P}_\varphi = \{\varphi_1, \ldots, \varphi_n\}$ of $\varphi = (P(u_o), X)$, OGK-EM first partitions constant tree keys $\mathcal{P}_c \subseteq \mathcal{P}_\varphi$, from variable tree keys $\mathcal{P}_v = \mathcal{P}_\varphi \setminus \mathcal{P}_c$, which contain at least a variable node besides u_o (via procedure PVal). We then partially validate each tree key and refine the match sets to update $V[u_o]$.

Specifically, for each constant tree key φ_i, PVal sets its flag isC = true. Next, for each pattern node u, we refine match set $P(u, G)$ as $\bigcap_{i=1}^{n} P_i(u, G)$, from all tree keys in $\mathcal{P}_\varphi$ containing pattern node u. For each child $\varphi' = (P'(u'_o), X')$ of φ in G_Σ, we initialize the match set $P'(u'_o, G)$ with the refined match set of the corresponding variable node in P. Lastly, this decomposition and partial validation process is then repeated for all children of φ following a breadth-first traversal of G_d.

(3) *Bottom up Synthesizing.* After completing the top-down decomposition where all leaves in G_d are processed, OGK-EM traverses G_d "bottom up" from the leaves to the root. For each node $\varphi = (P(u_o), X)$ in G_d, visited in descending node rank, OGK-EM invokes procedure SVal to compute $P(u_o, G)$ and the bisimilar matches, to update $V[u_o]$ by enforcing equivalence classes. For each tree key φ, we refine its match set by using the match sets from its children. We update $V[u_o] \in V_\mathcal{E}$ by iteratively merging two equivalence classes $([v], [v'])$ induced by pairs (v, v') from bisimilar matches $P(u_o, G, f)$ and $P(u_o, G, f')$, where each pair simulates a Chase step. When we can no longer enforce such pairs, OGK-EM sets Val=true for φ. The bottom up traversal terminates once Val=true for all root nodes in G_d, *i.e.,* the enforced base graph G', $G \models \Sigma$.

We illustrate the running of top-down phase below and present the details of Decomp, PVal, and SVal.

Procedure Decomp. Decomp extracts the non-recursive portion of OGKs as tree patterns to aggressively prune matches. Given an OGK φ with pattern $(P(u_o)$, it generates a set of *tree keys* $\mathcal{P}_{\varphi}$ by computing a set of tree covers $\mathcal{P}_{\varphi} = \{\varphi_1, \ldots, \varphi_n\}$ for $P(u_o)$ [45]. To this end, it invokes a 4-approximation to compute tree cover.

Procedure PVal. Given a (tree key) OGK $\varphi = (P(u_o), X \rightarrow l_o)$, PVal simulates a partial Chase process that refines the matches and initializes equivalence classes (representing equivalent entities) enforced by φ, without having to enumerate all possible pairwise matches. Given the match sets initialized by Decomp, PVal verifies using the subgraph isomorphism test (with early termination) whether each candidate node $v \in P(u, G)$ is a match under the scope (G, O, θ). It further prunes $P(u, G)$ by retaining only those matches satisfying the literals in X. If any set $P(u, G)$ becomes $\emptyset$, the verification of the current OGK ends.

Example 27: Consider the OGKs φ, φ' and φ'' for entities *album*, *producer*, and *band* in Figure 4.5. A partial dependency graph G_d is illustrated with two (dotted) edges (φ, φ') and (φ, φ''), for entities *producer* and *band*, respectively. In the top-down phase, Decomp decomposes φ to two tree keys with patterns P_1 and P_2. Their common matches, computed by PVal, refine the match set of P to $\{a_2, \ldots, a_5\}$. The process continues following G_d until it reaches leaves. □

Procedure SVal. Once the traversal reaches the leaf level (each interior variable node has been decomposed into its own tree keys, and every child φ' of φ in G_d has φ'.Val = true), SVal verifies the bisimilar matches for $P(u_o, G)$ with the following invocation cases (Figure 4.6).

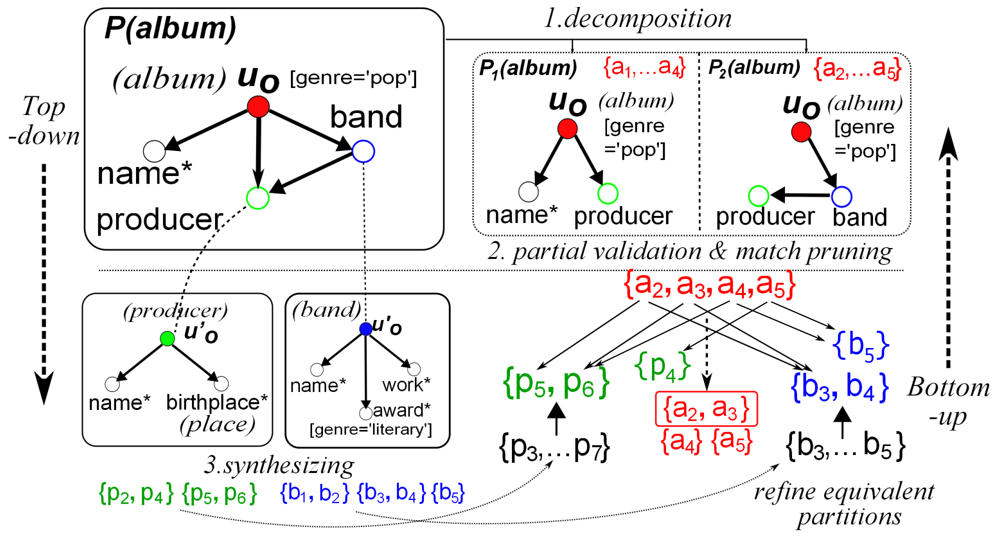


Figure 4.5: OGK-EM: Entity matching and pruning.

φ is a leaf. For $\varphi = (P(u_o), X)$, PVal computes matches by verifying for each pair (v, v') from $P(u_o, G)$, whether there exists a pair of bisimilar matches $P(G, f) \sim P(G, f')$. To avoid enumerating all pairwise comparisons, we extend the VF2 algorithm with backtracking [38] to consider partial matches (v, v') with an added feasibility condition: for all (v_1, v_2) that match a constant node u in $P(u_o)$, v_1.val = v_2.val. If (v, v') are induced by bisimilar matches, we merge $[v]$ and $[v']$ in $V[u_o]$. After all pairs have been evaluated, SVal sets φ.Val = true in G_d. We further optimize the matching by using an ontology index [118] (see details in [1]).

Leveraging children equivalence classes. If each child $\varphi'(u'_o)$ of $\varphi(u_o)$ in G_d has φ'.Val = true, then no further equivalence classes for $\varphi'(u'_o)$ can be merged from bisimilar matches. SVal uses these equivalence classes to refine $V_{\mathcal{E}}(u_o)$. First, for each variable node $u'_o \in V_x$ of $P(u_o)$, SVal induces a *local view* $V_\varphi[u'_o]$ of enforced equivalence classes

$V[u'_o]$, where $V_\varphi[u'_o]$ is defined as $\bigcup_{v[u]\in V[u]} v[u] \cap P(u,G)$, *i.e.,* the partitions $V[u]$ that are induced by nodes in $P(u'_o, G)$. Second, SVal then constructs an equivalence relation $R(\varphi, u_o)$, where a pair of nodes $(v_o, v'_o) \in R(\varphi, u_o)$ if the following hold:

- $\{v_o, v'_o\} \subseteq P(u_o, G)$, and
- for each variable node $u_x \in V_x$, there exists v and v' such that $(v, v') \in V_\varphi[u'_o]$, and v_o, v (resp. v'_o, v') are from the same match $P_i(G, f)$ (resp. $P_j(G, f')$) of at least a tree key P_i (resp. P_j) of $P(u_o)$.

The relation $R(\varphi, u_o)$ partitions $V_\varphi[u_o]$ of $P(u_o, G)$, and SVal verifies each pair (v_o, v'_o) using only equivalence classes $v_\varphi[u] \in V_\varphi[u_o]$, *without* having to do pairwise comparisons from $P(u_o, G)$ (using the extended VF2 algorithm). Lastly, we update $V[u_o]$ to enforce equivalence classes of u_o, and set $\varphi.\mathsf{Val} = \mathsf{true}$. Lemma 4 shows that the reduced number of verification steps continues to preserve correctness.

Lemma 4: *For any* OGK $\varphi(u_o) \in \Sigma$*, and any entity equivalent pairs* $(v, v') \in R(u_o)$*,* $(v, v') \in R(\varphi, u_o)$*.* □

Early termination. Given Lemma 4, OGK-EM terminates the matching process early for φ whenever $R(\varphi, u_o)$ is an identity relation without verification. Indeed, entity graph $V_\mathcal{E}$ will remain unchanged for all such OGKs. The estimated $V[u_o]$ (with nodes from $P(u_o, G)$ that may contain non-matches) is used to prune the match sets of its parents.

```
Procedure SVal($\varphi(u_o), G_d$)
1.  if $V[u_o] = \emptyset$ then $V[u_o] := \{[v] | [v] = \{v\}, v \in P(u_o, G)\}$;
2.  if $r(\varphi) = r_m$ then /*a leaf node in $G_d$*/
3.   for $(v_o, v'_o) \in P(u_o, G)$ do
4.     if (Verify($v_o, v'_o, u_o$)=true) then
5.      $[v_o]_{u_o} := [v'_o]_{u_o} \cup [v'_o]_{u_o}$;
6.      enforce equality of $\mathsf{eid}_o$ on $[v_o]_{u_o}$;
7.      update $V[u_o]$ with $[v_o]_{u_o}$; break;
8.  if $(r(\varphi) < r_m)$ then
9.    for (variable node $u'_o \in V_x$) do
10.    compute $V_\varphi[u'_o]$ and $V_\varphi[u_o]$ $(R(\varphi, u_o))$;
11.    for $(v_\varphi[u_o] \in V_\varphi[u_o]$ and $(v, v') \in v_\varphi[u_o])$ do
12.     if (Verify($v, v', u_o$)=true) then
13.      merge $[v]_{u_o}$ and $[v']_{u_o}$; update $V[u_o]$;
14. return $V[u_o]$;
```

Figure 4.6: Procedure SVal

Coping with SCC nodes. For an SCC node v_d in G_d that contains multiple OGKs, OGK-EM resolves equivalence classes in a similar manner as single OGKs, but conducts a fixpoint computation. For each $\varphi \in v_d$, we execute Decomp once, propagate the refined match sets, and update $V_{\mathcal{E}}$ among the OGKs via PVal and SVal, where the match sets monotonically decrease due to refinement. When no further changes can be made to any node match sets, the process terminates. We enforce the equivalence classes for all OGKs in v_d in a single batch, and set $\varphi.\mathsf{Val} = \mathsf{true}$ for all $\varphi \in v_d$.

Example 28: Continuing our example in Figure 4.5, in the bottom-up phase, a set of equivalence classes for *producer* ($\{\{p_2, p_4\}$,$\{p_5, p_6\}\}$) and *band* ($\{\{b_1, b_2\}$, $\{b_3, b_4\}$,

$\{b_5\}\}$) are derived from the two children of OGK φ, respectively. For the node *producer* in φ with potential matches $\{p_3, \ldots, p_7\}$, SVal first induces a local equivalence partition given that all equivalent *producer* entities are known by enforcing φ'. This induces a non-singleton equivalence class $\{p_5, p_6\}$ and a singleton $\{p_4\}$. Similarly, it induces local equivalence classes $\{b_3, b_4\}$ and $\{b_5\}$ derived from φ'' for *band*.

The match set for *album* $\{a_2, \ldots, a_5\}$ (obtained in the top-down phase; Example 27) is then refined to $\{\{a_2, a_3\}$, $\{a_4\}$,$\{a_5\}\}$. Specifically, (1) a_2 and a_3 have a child p_5 and p_6 respectively, both from a same equivalent class for *producer*; and share a same child b_3 (*band*); (2) a_4 and a_5 each has children either from *producer* or *band* that distinguish them from equivalent entities. Thus, only a single pair of entities $\{a_2, a_3\}$ need to be verified for entity equivalence. $\square$

Analysis. The algorithm $\mathsf{OGK\text{-}EM}$ correctly computes a Chase process and enforce Σ to update G. It suffices to observe the following invariants. (1) Procedures PVal and SVal correctly identifies entity equivalent pairs (v, v') for each OGK. (2) Whenever $\mathsf{OGK\text{-}EM}$ sets $\varphi.\mathsf{Val} = \mathsf{true}$, $G' \models \varphi$, where G' is the base graph of the current hypergraph $V_{\mathcal{E}}$. (3) When $\mathsf{OGK\text{-}EM}$ terminates, Σ is correctly enforced ($G \models \Sigma$).

Let the set $C(u_o)$ be $\{v | L(v) \in \mathsf{lsim}(u_o), v \in V\}$ for scope (G, O, θ). Denote the maximum diameter of a pattern in Σ as d. It takes $O(|\Sigma|^2)$ time to construct G_d, and for each OGK $\varphi \in \Sigma$, in total $O(|N_d(C(u_o))|^{2|P(u_o)|})$ time to identify entity equivalent pairs and enforce φ to G, where $N_d(C(u_o))$ refers to the d hop neighbors of node set $C(u_o)$. The overall time cost of $\mathsf{OGK\text{-}EM}$ is thus $O(|\Sigma|^2 + |\Sigma||N_d(C_m)|^{|P_m|})$, where C_m

(resp. P_m) refers to the largest $C(u_o)$ (resp. $P(u_o)$ in Σ).

In practice, $|\Sigma|$, $|P_m|$ and d are typically small. It is quite feasible to compute Chase over large graphs, as verified by our experimental study (Section 4.6).

4.5 Budgeted Entity Matching

Resources are often limited in practice, and constraints are imposed to minimize the effort and cost to perform entity matching [85]. In this section, we introduce a budgeted version of OGK-EM that performs entity matching with bounded matching cost.

4.5.1 Adding a Cost Model to Chase

We start with a cost model for Chase with OGKs.

Cost Model. Given OGK $\varphi = (P(u_o), X)$, scope (G, O, θ), let $\mathsf{Chase}(i) = (\mathcal{G}^i \overset{(\varphi,f)}{\Longrightarrow} \mathcal{G}^{i+1})$, represent a single Chase step. We define the cost $c(\mathsf{Chase}(i))$ of a single Chase step as $c(u_o, f(u_o))$, which is the cost to match u_o and $f(u_o)$. For a Chase sequence,

$\mathsf{Chase}(ij)$, we define $\rho = (\mathcal{G}^i \overset{(\varphi,(v_1,v_2))}{\Longrightarrow} \dots \overset{(\varphi',(v_1',v_2'))}{\Longrightarrow} \mathcal{G}^j)$, and the cost is computed as

$$c(\rho) = c(\mathcal{G}^i, \mathcal{G}^{i+1}) \cdot \sum_{i}^{j-1} c(\mathsf{Chase}(i))$$

where $c(\mathcal{G}^i, \mathcal{G}^j) = \frac{|U|}{|V|}$, and U refers to the total number of entity identifiers ($v.\mathsf{eid}$) updated in $\mathcal{G}^i$ to enforce the Chase rules. Intuitively, we measure the cost to enforce Σ, which requires merging and transforming nodes in $\mathcal{G}^i$ to those in $\mathcal{G}^j$, and includes the matching cost of each Chase step.

Budgeted Entity Matching Problem. Given a set of OGKs Σ, scope (G, O, θ), and a budget B, we want to compute a Chase sequence ρ that generates a $\mathcal{G}$, such that ρ has a smallest cost $c(\rho)$ bounded by B.

Our goal is consistent with evaluating entity resolution with "merging" cost of entities [85], while (1) $c(\rho)$ aggregates the editing cost weighted by ontological matching cost; and (2) B encodes a threshold to distinguish "good" entity matching results and infeasible ones. Intuitively, we identify feasible entity matching result with a minimum cost. Budgeted entity matching generalizes the entity matching problem in Section 4.3: the latter carries unit Chase step cost with $B = \infty$, This leads to the following result.

Lemma 5: *Budgeted matching with* OGKs *is* NP*-hard.* □

To find a Chase with minimal cost, we model this as a *planning* problem that outputs a sequence of OGKs $\pi = \{\varphi_1, \ldots, \varphi_n\}$ to be enforced with minimum cost. We show that the search can be optimized by revising OGK-EM with *beam search and backtracking.*

4.5.2 Budgeted Entity Matching Algorithm

Overview. We introduce a *budgeted* version of OGK-EM called BOGK-EM, optimized by *beam search with backtracking.* Beam search is a heuristic optimization of breadth-first search that traverses a search tree by expanding and exploring the most promising nodes, up to a fixed number b, called the *beam size.* BOGK-EM follows beam search to select the top-b "best" OGKs following node ranks in dependency graph, but dynamically reduces the allowed budget according to current best solution, and backtracks to explore alternative Chase sequences. Moreover, BOGK-EM dynamically estimates an upper bound of Chase cost to prioritize the selection of promising beam elements as OGKs.

Auxiliary structures. BOGK-EM uses the dependency graph G_d to coordinate the search. For each node v_d in G_d, it extends the auxiliary information of v_d to a vector $\{\mathsf{isC}, \mathsf{Val}, U)\}$, where U is an estimate of the additional Chase cost to enforce all OGKs in v_d. At each layer i, BOGK-EM records: (1) an open set $\mathsf{open}(i)$ of candidate OGKs to be explored; (2) a set $\mathsf{L} \subseteq \mathsf{open}(i)$ of OGKs to be validated and enforced with tunable memory size; and (3) a stack S containing values (i, c_{min}, c_{max}) that define the allocated cost range to OGKs ($\mathsf{open}(i)$) evaluated at layer i.

Algorithm BOGK – EM

Input: dependency graph G_d, scope (G, O, θ); budget B; beam size b;
Output: the entity graph $V_{\mathcal{E}}$ under budget B.

```
1.  initializes $V[u_i]$ ($u_i \in \mathcal{E}$);
2.  integer $r$ :=$l+1$ (l: the maximum node rank of $G_d$);
3.  set $\pi*$ :=$\emptyset$; set $\pi$ := $\emptyset$; stack $S$.push$(r, 0, B)$; cost $U$:=0;
4.  **while** $S.top() \neq \emptyset$ **do**
5.    $\pi$:= Bchase $(\pi, r, S, b, G_d)$;/*compute a Chase under budget*/
6.    **if** $\pi \neq \emptyset$ **then**
7.      $\pi^* = \pi$; $U$:= $\pi$.cost; /* current optimal chase*/
        /* set new cost range to explore Chase */
8.    **while** $S.top()$.cmin $\geq U$ **do** $S$.pop();
9.    $S$.top().cmin:= $S$.top.cmax; $S$.top().cmax:= $U$;
10.   **if** $S$=$\emptyset$ **then**
11.     construct $V_{\mathcal{E}}$ by enforcing sequence $\pi^*$;
12.     **return** $V_{\mathcal{E}}$.
```

Procedure Bchase(π, r, S, b, G_d)
/*compute a chase from level $l+1$ of G_d*/

```
1.  open$(r)$:=$\{v_s\}$;
2.  set open$(r-1)$ as the leaves of $G_d$;
3.  **while** open$(r) \neq \emptyset$ **do**
4.    set $L(r)$ as top $b$ nodes with smallest $v_d.U$ in sorted open$(r)$;
5.    $S.top()$.cmax:=$v_{b+1}.U$ ($v_{b+1} \in$ open$(r)$ is the $b+1$th node);
6.  $\pi$:=$\pi \cup \{L\}$;
7.  open$(r+1)$:= setopen(open (r);
8.  S.push(r+1, 0, B);
```

Figure 4.7: Algorithm BOGK-EM

Algorithm. The algorithm BOGK-EM is illustrated in Figure 4.7. Let the leaves in G_d be of rank l, and v_s be a pseudo node of rank $l+1$, connecting to the leaf nodes. BOGK-EM initializes the auxiliary structures, as well as the current best Chase and the newly constructed Chase with π^* and π, respectively (lines 1-3). It then executes a layered, beam search with backtracking starting at v_s, invoking procedure Bchase (lines 4-9) to update π^*. It then enforces π^* to perform entity matching (lines 10-12). We next describe procedure Bchase.

Beam selection. At each layer i, Bchase evaluates all nodes in G_d with rank i. If $i = (l+1)$, we initialize the candidate set of OGKs to explore, open(l), to all the leaves in G_d with rank l, set OGKs π and π^* to $\emptyset$, and push $(l, 0, B)$ to stack S. For each candidate node v_d in open($i-1$), representing an OGK $\varphi = (P(u_o), X \rightarrow l_o)$, we compute the matches for each node u in $P(u_o)$ using Decomp and PVal. We derive an upper bound of the matching cost $v_d.U$ as

$$v_d.U = \frac{|P(u_o, G)|}{|V|} \max_{v_o \in P(u_o, G)} \hat{c}(u_o, v_o)$$

where $\hat{c}(u_o, v_o)$ is an overestimated cost, computed as

$$\hat{c}(u_o, v_o) = \frac{1}{|V_P|} \sum_{u' \in V_P} \max_{v' \in P(u', G)} c_r(u', v')$$

We overestimate the matching cost by assuming that every pruned node in Decomp and PVal are indeed matches. The cost of an SCC node v_d is computed similarly, as the sum of the estimated cost for each OGK $\varphi \in v_d$.

Bchase initializes L with the b OGKs in open(l) of smallest cost within the range (S.$top()$.cmin, S.$top()$.cmax]. For beam selection when backtracking to layer i, we define the cost range for the next batch of OGKs by pushing a triple (i, $\max_{v_d \in \mathsf{L}}(v_d.U)$,

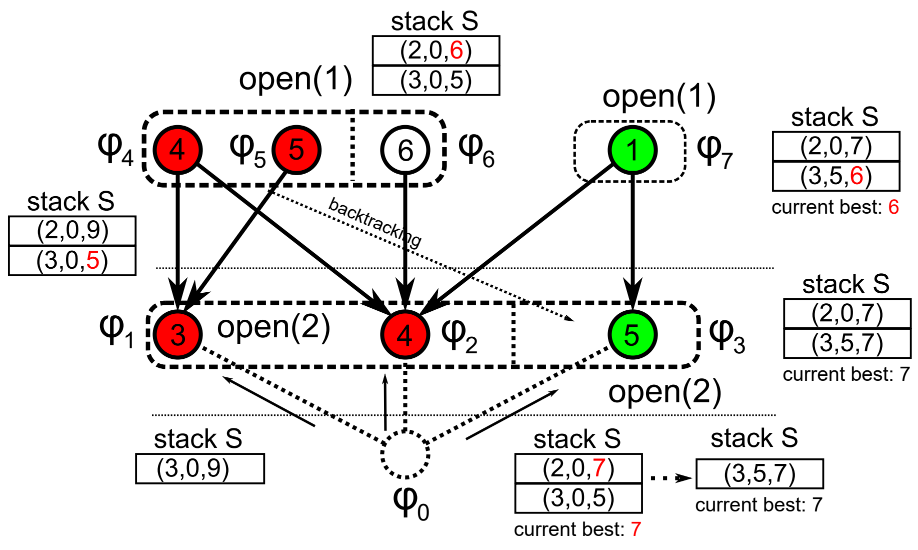


Figure 4.8: BOGK-EM: budgeted Chase with backtracking.

B) onto the stack S.

Backtracking. We refine each OGK in $\mathsf{L}(i-1)$ by refining its matches using procedure SVal. We also update $v_d.U$ to its true cost B', and add the selected OGK to the current Chase π. We must update the list of candidates for the next level, $\mathsf{open}(i-2)$, as the set of nodes in G_d with validated children (using flag Val), and deduct B' from the current budget B. If $B > 0$, and candidates remain in $\mathsf{open}(i-2)$ or $\mathsf{open}(i-1)$, Bchase will continue processing the next layer $i-2$. Otherwise, we set the optimal Chase $\pi^* = \pi$, if $\pi^*=\emptyset$, or $c(\pi) < c(\pi^*)$. We backtrack to layer $i-1$, to populate the cache $\mathsf{L}(i-1)$ with the b OGKs of lowest cost $v_d.U \in (\mathsf{S}.top().\mathsf{cmin}, \mathsf{S}.top().\mathsf{cmax})$, and update the stack S. After processing all candidate OGKs in layer $i-1$, we set $\mathsf{open}(i-1) = \emptyset$. BOGK-EM terminates when $\mathsf{open}(l)$ is $\emptyset$, and enforces the equivalence classes for each OGK in π^*.

Example 29: Consider the dependency graph G_d in Figure 4.8. Let the beam size $b = 2$ and a budget $B = 9$, BOGK-EM computes a budgeted Chase as follows. (1) Starting from the pseudo node v_s (φ_0), it initializes the stack S with $(3, 0, 9)$, stating "Chase with cost in [0,9] will be explored". As the costs of φ_1, φ_2, and φ_3 are validated to be 3, 4 and 5 respectively and b=2, It sets open (2) as $\{\varphi_1, \varphi_2, \varphi_3\}$, $L(2) = \{\varphi_1, \varphi_2\}$, and updates $S.top()$ to be $(3, 0, 5)$, to prevent traversing Chase with cost less than 5 when backtracking to level 2. (2) At level 2, BOGK-EM pushes $(2, 0, 9)$ to S, identifies $L(2)$ to be $\{\varphi_4, \varphi_5\}$, and updates the top of S to $(2, 0, 6)$ similarly. (3) At level 3, it constructs the current best Chase $\{\varphi_1, \varphi_4\}$. As the only successor of φ_4 exceeds the budget, it backtracks to level 1. (4) Given the current minimum cost 7, it updates $S.top()$.cmin to 5, and $S.top()$.cmax to 7, "switching" to explore Chase with cost in [5, 7]. This finally yields a better Chase $\{\varphi_5, \varphi_7\}$ with cost 6. □

Analysis. BOGK-EM is an anytime algorithm that can return the current Chase π upon request, and can continue to refine π if needed, with decreasing incremental cost to generate new solutions. It is also memory efficient, with the memory cost in $O(2b \cdot (N_d(C_m) + |P_m||C_m|) + |\Sigma|)$, where d is the diameter of P_m.

4.6 Experimental Study

We use real graphs and ontologies to evaluate: (1) the efficiency of entity matching using OGKs; (2) the effectiveness of entity matching (with ground truth), and the trade-off between effectiveness and efficiency (using injected redundancy); and (3) case studies for real entity matching.

Experimental Setup. We implement all our algorithms in Java v8, and ran our experiments on a Linux machine with AMD 2.7 GHz CPU with 256 GB of memory.

Datasets. We use two real benchmark datasets containing the ground truth to evaluate the efficiency and effectiveness of our techniques. We also develop a data generator to inject duplicates into one of the benchmark datasets to evaluate scalability. Table 4.2 summarizes the data characteristics.

DBpedia-Yago[2]: this benchmark graph (used in [111]) contains $50,248$ verified ground truth entity pairs with aligned properties between DBpedia and Yago. These equivalent pairs cover 10 types of entities. To create a "hybrid" ontology, we added "is A" relations between two types for each ground truth entity pair, one from the DBpedia ontology, and the other from the Yago ontology.

DBpedia-IMDb[3]: this second benchmark graph (used in [43]) contains 33,437 entities covering 10 types, totaling $9,515$ redundant pairs between DBpedia and IMDb. We create the corresponding ontology that links entities between DBpedia and IMDb following a similar process for DBpedia-Yago.

DBpYago-Dup*:* To evaluate the trade-off between efficiency and effectiveness, we create a data generator that injects a controlled number of duplicate entity pairs, facts, and

[2]`https://github.com/lgalarra/vickey`

[3]`https://www.csd.uoc.gr/~vefthym/minoanER/datasets.html`

Dataset	#entities	#triples	#labels	#duplicates
DBpedia-Yago	592K	4.5M	10	50248
DBpedia-IMDb	33K	200K	10	9515
DBpYago-Dup	4.6M	29M	935	65248

Table 4.2: Data characteristics.

labels into DBpedia-Yago, and its corresponding ontology. The duplicate entries are duplicated from the ground truth entities with varied labels from the original ontology. The generator injects pairs of "seed" equivalent entity pairs (v, v'), and duplicates v and v' in DBpedia and Yago respectively, along with their neighborhood up to 3 hops. The generator then disturbs the entity types of these duplicates to a concept label in the ontology, and randomly updates 50% of the literals of duplicated entities. We then identify a set of true examples Γ^+ (containing equivalent entity pairs), and false examples Γ^- (that do not refer to the same entity as a result of injecting noise into the labels and literals).

We use the two benchmark graphs to evaluate the comparative accuracy of OGK based techniques against the baselines, and the controllable DBpYago-Dup to verify efficiency.

OGKs Generation. We extend the key mining algorithm, KeyMiner [16] to generate OGKs. Given ground truth, it discovers OGKs via level-wise graph pattern mining to identify maximal entity patterns and literals that preserve bisimilar matches for entity equivalent pairs. We only extracted the OGKs with at least 80% support of finding the entities and 90% confidence over the matched entities. We used hyponym edges to identify entity types, and equivalence and descriptive edges to identify concept labels, to be assigned to the OGK patterns. We extracted 10, 8, and 250 OGKs covering at least 40K, 7.6K and 52K of the ground truth in DBpedia-Yago, DBpedia-IMDb and DBpYago-Dup, respectively. We retain the OGKs with high support and confidence conditioned by proper literals. The OGKs for DBpYago-Dup are generated over the examples Γ^+ and Γ^-.

Algorithms. We implemented the following methods.

OGK-EM: our exact entity matching algorithm (Section 4.4).

EnumEM: a variant of OGK-EM without the top-down phase that follows a bottom-up strategy to perform pairwise verification of bisimilar matches without hypergraph refinement.

GK-EM: a variant of EnumEM that simulates graph key-based entity matching [48], by enforcing type equality without literals, and merges nodes instead of entity identifiers.

BOGK-EM: our weighted entity matching algorithm. We compare BOGK-EM against BEnumEM and BGK-EM, which are the budgeted versions that uses pairwise verification without hypergraph refinement, and enforces type equality, respectively. For pattern matching, we implement an algorithm that extends VF2 with ontology similarity [38].

Vickey [111] detects conditional keys via breadth-first search using logical rules with strict label equality. It considers neither topological nor ontological similarity.

Holistic [94] performs graph alignment by personalized page-rank from seed entities, and expands subgraphs (modeled as "pair graphs") to capture the impact of correlated entities.

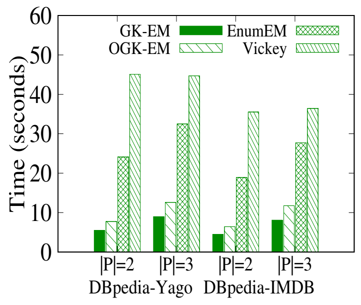
We set a support threshold 2% for Vickey, and a similarity threshold as 0.85 for Holistic to assert equivalence. We choose these settings to favor both methods under which they achieve the highest average precision and recall. All our source codes and test cases are available online[4].

4.6.1 Efficiency of Entity Matching

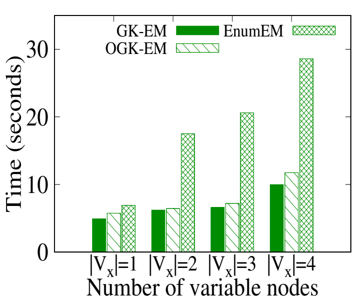
We evaluate the performance efficiency of OGK-EM and BOGK-EM against EnumEM, GK-EM, and BEnumEM, BGK-EM, respectively. Our objective is to evaluate the effectiveness of our optimizations on runtime, and the overhead of having ontology-enriched key semantics.

Exp-1: Efficiency of OGK-EM. Using DBpedia-Yago, we define 10 OGKs. We set lsim(l), and θ to include similar labels that are within 2 hops of l in the corresponding ontologies. The center nodes have on average 12000 matches. Figure 4.9a reports the time of entity matching and the impact of pattern size (number of edges). (1) It

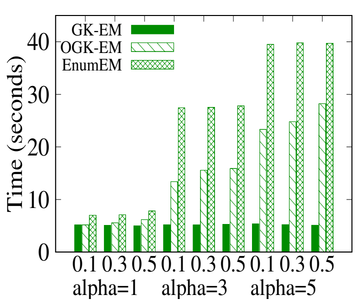
[4] `https://github.com/HanchaoMaWSU/OGKEM.git`



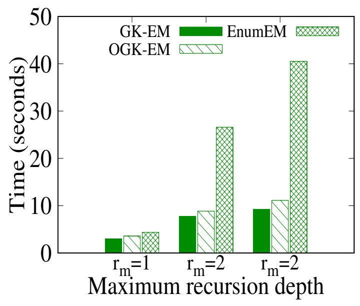
(a) Entity Matching Efficiency (Benchmarks)



(b) Varying $|V_x|$ (DBpYago-Dup)



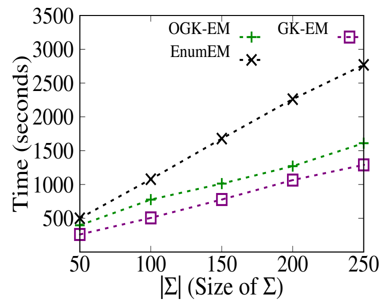
(c) Varying (α, θ) (DBpYago-Dup)



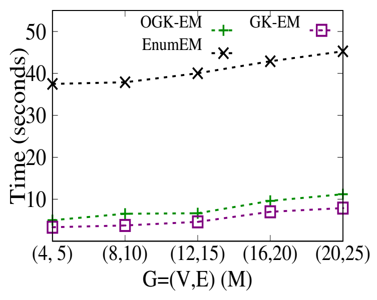
(d) Varying r_m (DBpYago-Dup)

Figure 4.9: Efficiency of Entity Matching using OGKs.

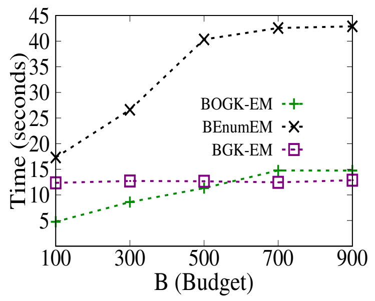
is quite feasible to identify equivalent entities over real-world graphs using OGK-EM. For example, it takes on average 10 seconds to enforce OGKs Σ over DBpedia-Yago, and 9.1 seconds over DBpedia-IMDb. (2) OGK-EM outperforms EnumEM by 2.4 times, and has comparable performance with GK-EM. This is notable since GK-EM enforces only label equivalence, thereby inspecting fewer entities than OGK-EM. (3) OGK-EM outperforms Vickey by 4.2 times on average. While the major bottleneck for both methods are pairwise comparison of entities, OGK-EM performs less comparisons due to an enriched OGK semantics and pruning strategy. Vickey identifies relatively more entity pairs that must be compared.



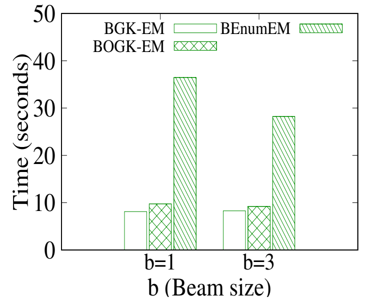
(a) Varying $|\Sigma|$ (DBpYago-Dup)



(b) Varying $|G|$ (DBpYago-Dup)



(c) Varying B (DBpYago-Dup)



(d) Varying b (DBpYago-Dup)

Figure 4.10: Efficiency of Entity Matching using OGKs.

We also found that Holistic cannot run to completion after 350 seconds. The major bottleneck is the construction of auxiliary structures, such as pair graphs.

Exp-2: Scalability. We evaluate the scalability of our algorithms by varying: the number of variable nodes ($|V_x|$), thresholds (α, θ), recursion depth (r_m), the number of OGKs ($|\Sigma|$), graph size ($|G|$=($|V|, |E|$)), budget (B), and beam size (b). We set $|\Sigma| = 10$, $|V_x| = 2$, $|G| = (16M, 20M)$, $r_m = 2$, and $(\alpha, \theta) = (2, 0.5)$ by default, unless otherwise specified.

Varying variable size $|V_x|$. Figure 4.9b reports the impact of the number of variable nodes $|V_x|$ (varied from 1 to 4 by selecting corresponding OGKs groups) to the performance of OGK-EM. While all algorithms take more time for OGKs with larger variable nodes, OGK-EM is less sensitive compared to EnumEM and GK-EM, due to its aggressive pruning that reduce the pairwise comparison costs. More equivalence classes can also be refined during the "bottom-up" phase over larger $|V_x|$, further reducing verification. On average, OGK-EM outperforms EnumEM by 2 times. *Varying* (α, θ). Figure 4.9c shows that OGK-EM scales well as OGKs relax strict equality matching to allow approximate entity matching using similar concept labels by tuning θ and α. OGK-EM must verify more matched entities, leading to longer running times. GK-EM lacks such tuning flexibility due to its strict enforcement of label equality and is quite insensitive to the change of α. We vary α and θ from 0.1 to 0.5. Increasing *alpha* by 0.1 will expand the search space by 1-hop in ontology graph. Given a pair of equivalent entities from the positive examples (benchmark), we calculate the distance between their labels in the corresponding ontology. We then induce the ranges of θ and α for our tests. Specifically, we varied θ and α from 0.1 to 0.5 as an applicable and reasonable configuration.

Varying recursion r_m. We vary the OGK recursion depth in Σ (*i.e.,* the maximum node rank r_m in dependency graph G_d) from 1 to 3. This occurs when new references are identified among the entities in Σ. Figure 4.9d show increased runtimes for all algorithms as deeper recursion r_m is enforced. However, OGK-EM terminates earlier for larger r_m, due to greater refinement of equivalence classes.

Varying $|\Sigma|$ *and* $|G|$. Figs 4.10a and 4.10b verify that all methods take longer time for larger $|\Sigma|$ and $|G|$. Specifically, OGK-EM outperforms EnumEM by 1.7 times and 4.8 times when $|\Sigma| = 250$ and $|G|$ is varied to $(20M, 25M)$, respectively. OGK-EM has a reduced number of comparisons, and comparable runtime with GK-EM, which only enforces label equality.

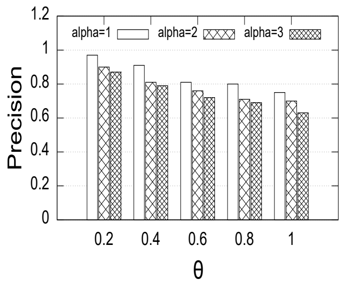
We now evaluate the anytime BOGK-EM for budgeted entity matching, and compare it with BEnumEM and BGK-EM. For a fair comparison, we report the runtime for each algorithm to converge to the best Chase for a given budget B.

Varying budget B. Figure 4.10c shows that all three budgeted algorithms take more time as we vary B from 100 - 900, due to additional verification from more merge operations. BOGK-EM outperforms BEnumEM by 5 times on average. ngive time over dataset; more about BGK-EM.

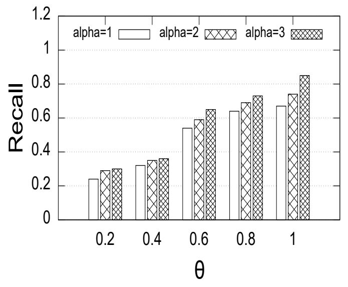
Varying beam size b. Fixing $|G| = (16M, 20M)$, Figure 4.10d shows that larger beam sizes allow BOGK-EM to prioritize Chase and select more promising OGKs first. Aggressive pruning can also occur during the bottom-up phase due to more OGKs at the leaf level. BGK-EM is not affected by b since it does not execute a beam search.

4.6.2 Effectiveness of Entity Matching

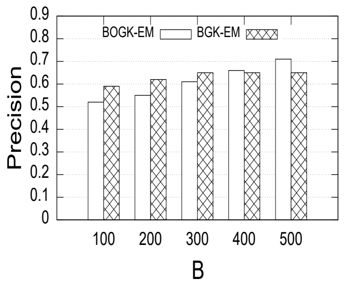
Exp-3: Impact of Parameters to Accuracy. We investigate the impact of the thresholds (α, θ) and the budget B using DBpYago-Dup. We injected 15K true and



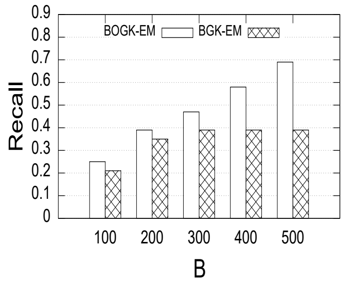
(a) precision vs.(α, θ) (DBpYago-Dup)



(b) rec vs. (α, θ) (DBpYago-Dup)



(c) precision vs. B (DBpYago-Dup)



(d) rec vs. B (DBpYago-Dup)

Figure 4.11: Effectiveness of Entity Matching using OGKs.

false examples each to DBpYago-Dup ($|\Gamma^+| = |\Gamma^-| = 15$K). By default, we set $\alpha = 3$, $\theta = 0.8$, and $B = 500$.

Varying threshold pair (α, θ). Recall that larger values of α and θ relax the strict equality conditions for label and ontological matching, respectively, e.g., label equality is enforced at $\alpha = 0$. By allowing this flexible similarity matching, we achieve precision gains over existing techniques. For example, OGK-EM achieves an 18% gain in precision on average over GK-EM by capturing more correct entities via ontological matching. Figure 4.11a shows that as (α, θ) increase, precision decreases, as more weakly related entities (false positive examples) are captured due to approximate

Entity Type	OGK-EM	Vickey	Holistic
(# positive)	P/R/F	P/R/F	P/R/F
Book(13.2K)	**0.97/0.85/0.9**	0.96/0.36/0.5	0.8/0.58/0.67
Actor(5K)	**1/0.66/0.79**	0.95/0.12/0.2	0.6/0.36/0.46
Museum(3.2K)	0.99/**0.42/0.58**	**1**/0.1/0.18	0.82/0.14/0.2
Scientist(16.1K)	**0.99/0.67/0.8**	**0.99**/0.2/0.33	0.72/0.56/0.6
University(12.6K)	**0.96/0.5/0.66**	0.93/0.11/0.2	0.79/0.3/0.43
Movie(9.5K)	**0.95/0.74/0.8**	0.95/0.1/0.18	0.65/0.1/0.17

Table 4.3: Comparative accuracy against baselines

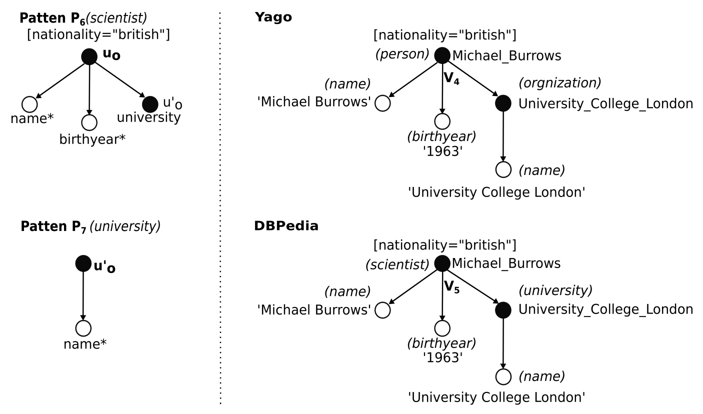
matching. We also verify this trend as fpr increases for larger (α, θ) values. In contrast, Figure 4.11b shows that increasing α and θ improve recall. We observe a 53% increase in recall and 25% increase in F_1 score (not shown) when (α, θ) is varied from (1,0.4) to (3,1), as more true positive entities are captured by OGKs via ontological matching.

Comparing the above analysis with their efficiency counterparts in Exp-2 (Figures 4.9c and 4.10c), we verify that OGK-based techniques support a flexible trade-off between matching efficiency and accuracy by tuning (α, β) and budget B.

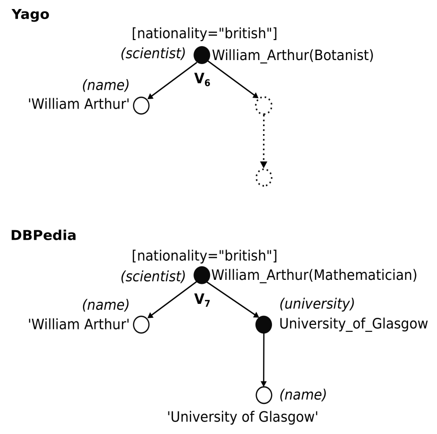
Exp-4. Case Analysis. We study the accuracy of OGK-EM against the baselines, and report a case analysis over 6 common entity types in Table 4.3, in terms of precision (P), recall (R) and F-measure (F). All entities are from DBpedia-Yago except for *Movie* entities, which are from DBpedia-IMDb. For each entity type, we consider 40% of the dataset as a training dataset and mine the keys over the training dataset. Then we validate the keys over 10% of the dataset and if the validation meets the criteria of OGKs, we run each algorithm over the rest 50% of the dataset by using the mined keys. As mentioned before, we consider 80% support and 90% confidence to mine OGKs over the training dataset and use the same thresholds for the validation. We set α=2, θ=0.8, and B=500. These settings over real benchmark graphs are guided by cases that have high accuracy using DBpYago-Dup. When no ground truth is available, a configuration can be initialized by duplicating entities with equal labels and examining cases with reasonable accuracy that cover these ground truth subsets.

We observe the following. (1) OGK-EM achieves the best precision and recall in most cases using OGKs: the joint topological and value constraints improve precision, while the ontological matching mitigate loss of recall. (2) Vickey achieves comparable precision at a cost of low recall due to enforcing label equality. For example, Vickey fails to identify *Michael_Burrows(scientist)* and *Michael_Burrows(person)*. (3) Holistic achieves higher recall but lower precision compared with Vickey. Indeed, not all "similar" entities are equivalent. For example, a pair of scientists *William_Arthur(botanist)* and *William_Arthur(mathematician)* are matched by Holistic due to having the same name, work place and nationality. However, this case is correctly distinguished by OGKs due to a high ontological matching cost.

We manually verified OGKs and equivalent entities. Two such keys are shown in Fig. 4.12a from DBpedia-Yago. (1) φ_1=(P_6(scientist), scientist.nationality='british') states that if two scientists with the nationality equals to 'british' share the same name, birth year and university, then they refer to the same scientist. (2) φ_2 with pattern P_7 identifies universities with their names, referred by φ_1. φ_1 identifies a pair of scientists (V_4 and V_5), which are equivalent in DBpedia-Yago, via ontological matching where *person* is 'superClassOf' *scientist* and *organization* is 'superClassOf'



(a) Real-world OGKs



(b) DBpedia-Yago Data Subgraphs

Figure 4.12: Effectiveness of Entity Matching using OGKs.

university. Vickey can apply conditional keys to detect duplicate entity pairs. However, it has limitations due to ignoring ontological similarities. For an instance, two matches V_4 and V_5 in Fig. 4.12a represent two scientist entities in Yago-DBpedia. When Vickey applying conditional keys of scientist, it regards only V_5 as a scientist entity and ignoring V_4 since the ontological type of $Michael\_Burrows$ in V_4 is *person*.

This verifies that OGKs benefits entity matching with finer grained conditions and tunable tolerance to label similarity.

Consider an alternative OGK φ_3=(P_8(scientist), scientist. nationality='british') which requires that two scientist refer to the same scientist if they share the same name and graduate from the same university. Fig. 4.12b shows two scientist entities (V_6 and V_7) in DBpedia-Yago. Given the pair of seed entities, Holistic initializes string similarity base on common literals which are linked by common edges. Then, it propagates the similarity to obtain the similarity score between the entity pair. V_6

and V_7 share one edge and the only common literal $'William\ Arthur'$. By applying Holistic, V_6 and V_7 inaccurately refer to the same scientist due to the high similarity score. Nevertheless, by applying φ_3, V_6 and V_7 are not regarded as the same scientist as V_6 does not have edges linking to university entities.

This verifies that OGKs benefits entity matching with enforcing rigorous topological constraints.

4.7 Related work

Ontological Dependencies. Existing work has coupled ontologies with functional dependencies (FDs), and equality-generating dependencies (EGDs) over RDF triples, along with an axiomatization, and inference algorithms, for both types of dependencies [13, 69]. An extension to OWL ontologies with integrity constraints is proposed to validate ontology completeness, by defining inclusion dependencies and domain constraints to check for missing and valid domain values [86]. These constraints, however, are limited to value bindings with no topological restrictions. Recent work has proposed ontological FDs (OFDs), a tighter integration of FDs and ontologies in relational data. OFDs relaxing the strict equality conditions in FDs to include synonym and inheritance (is-a) notions of semantic equivalence [27]. OFDs have shown to significantly reduce false positive errors, improve recall in entity resolution, and data cleaning [121, 14, 122].

This work extends keys for graph to include both graph patterns and ontologies to capture topological constraints and semantic equivalence. OFDs restrict ontological extensions to only the consequent attribute of the dependency. Furthermore, OFDs cannot be directly applied to characterize OGKs with topology and value constraints. Our work also benefits from pattern matching with ontologies [118].

Graph Dependencies. Recent work has proposed variants of graph functional dependencies (GFDs) that define value relationships over entities satisfying topology constraints, such as trees [119], and subgraph isomorphism [57, 53]. Keys for graphs are a special case of GFDs, but differ in that they consider *value equality* (based on value bindings of properties), and *node identity* to identify an entity [48]. In addition, keys for graphs may be *recursively defined*, allowing entity identification to be dependent on sub-entities. This recursive dependency makes keys for graphs inherently more complex than GFDs. In OGKs, this recursive complexity manifests in entity matching to efficiently resolve all dependent sub-entities, and to accurately propagate semantic similarity across the matched nodes. Graph entity dependencies (GEDs) unifies GFDs and keys [53].

Closer to our work is conditional keys for RDFs [111]. These constraints are defined by a conjunctive condition over attribute properties, and enforce attribute identity. A notion called *key graphs*, similar to entity patterns is proposed. Nevertheless, conditional keys are not characterized by patterns and topological constraints, but by conjunctive conditions. A special case of OGKs with label equality falls into GEDs that enforce identifier equivalence. These GED bindings, however, are fixed and inflexible to exploit external ontologies to reconcile semantically equivalent entities.

Graph Matching and Entity Categorization.

Probabilistic graph matching seeks a mapping that induces similar subgraphs from a pair of graph instances, and their node features [120]. Entity matching differs from graph matching as two entities that are "similar" in graph matching may not necessarily refer to the same entity. By using graph patterns and ontologies, we can precisely define the context of entity equivalence and also enable feasible entity matching for big graphs. Unlike entity categorization [65], OGKs incorporate ontological similarity during the matching process, and dynamically propagate this similarity to neighboring nodes to identify entity pairs that are semantically equivalent.

4.8 Conclusion

We proposed a class of ontological graph keys (OGKs), which are a variant of graph keys by relaxing node identity to entity identifier equivalence, and type equality to ontological matching. We extended Chase to characterize entity matching with OGKs, and show Chase preserves the Church Rosser property with fixed scope. We developed efficient algorithms with early termination, for both exact matching and budgeted matching. OGKs are a first class of approximate graph key constraints with tunable tolerance on type mismatching. As next steps, we intend to study the application of OGKs to knowledge fusion (e.g., compared with graph embeddings), and the parallel discovery of OGKs in distributed graphs.

Chapter 5

Conclusion

In this thesis, we addressed the challenges associated with data quality management over graphs in two dimensions, data consistency and data deduplication. For data consistency, we defined a new integrity constraint, called temporal graph functional dependencies (*i.e.,* TGFDs) for evolving temporal graphs and presented an efficient algorithm to detect inconsistencies *w.r.t.* TGFDs. For data deduplication, we proposed two properties to define a key and then an algorithm to mine all the keys for a given entity type that satisfy the properties. Our experimental evaluation shows that our algorithm, GKMiner, runs up to 6 times faster than the existing technique SAKey, and gains up to 61% gain in F_1-score. For the next research problem, we exploited external ontologies to enhance existing keys for graphs. We defined ontological graph keys (*i.e.,* OGKs) and presented an efficient algorithm to perform ontological entity matching over the attributed graphs. The evaluation shows that our proposed algorithm (OGK-EM) achieves the best precision and recall in most cases using OGKs compared to Vickey [111] and Holistic [94].

Temporal Graph Functional Dependencies. TGFDs generalize functional dependencies to temporal graphs as a sequence of graph snapshots that are induced by time intervals, and enforce both topological constraints and attribute value dependencies that must be satisfied by these snapshots. We establish the complexity results for the satisfiability and implication problems of TGFDs. We propose a sound and complete axiomatization system for TGFDs. We also present efficient parallel algorithms to detect inconsistencies in temporal graphs as violations of TGFDs. The algorithm exploits data and temporal locality induced by time intervals, and uses incremental pattern matching and load balancing strategies to enable feasible error detection in large temporal graphs. Using real datasets, we experimentally verify that our algorithms achieve lower runtimes compared to existing baselines, while improving the accuracy over error detection using existing graph data constraints, *e.g.,* GFDs and GTARs with 55% and 74% gain in F_1-score, respectively.

GKMiner. Keys for graphs uses the topology and value constraints needed to uniquely identify entities in a graph database for deduplication. We present our algorithm, GKMiner, to mine keys over graphs. GKMiner discovers keys in a graph via frequent subgraph expansion. We present two properties that define a key, including *minimality* and *support.* Lastly, using real-world graphs, we experimentally verify the efficiency of our algorithm on real world graphs.

Ontological Graph Keys. We propose a new class of key constraints, *Ontological Graph Keys* (OGKs) that extend conventional graph keys by ontological subgraph matching between entity labels and an external ontology. We study the entity matching problem with OGKs, and a practical variant with a budget on the matching cost. We develop efficient algorithms to perform entity matching based on a (budgeted) Chase procedure. Using real-world graphs, we experimentally verify the efficiency and accuracy of OGK-based entity matching.

5.1 Future Work

There is more work to do in both areas. Firstly, there needs to be more attention for qualitative approaches in the graph data quality management. While contextual data cleaning, incorporating semantics into the data cleaning process via external information, has been well studied in data quality over relational data [14, 27, 122], it has been less so for graph data quality management. More specifically, existing data consistency rules, such as TGFDs [18], GFDs [57] and GEDs [53] can be extended to support external ontologies, while we address this for graph keys [83] in this thesis. Secondly, as real world graphs are large and change frequently, there needs to be parallel algorithms to discover graph dependencies, including parallel discovery of TGFDs, GKeys, and OGKs. This would help to make practical use of these constrains in the graph data quality management.

5.1.1 TGFDs

As next steps, we intend to investigate following:

1. Explore non-parametrized methods to extend our workload rebalancing scheme to dynamically adapt to workload burstiness.

2. Given the utility of TGFDs, and the identified errors, graph data cleaning with respect to TGFDs would be an interesting next step.

3. Exploit ontologies to extend TGFDs to serve as a means to perform contextual data cleaning over temporal graphs. OFDs have been defined to extend traditional FDs with external ontologies for relational data and OGKs have been defined for graph keys. The use of ontologies could be explored for both GFDs and TGFDs.

4. Explore the use of TGFDs in existing graph databases *e.g.,* Neo4j [99] and Wikibase [40].

5. Similar to GEDs [53], we can extend the predicate in the dependency $X \to Y$ to go beyond equality. We intend to explore this extension to other predicates such as $\neq$, $<$, $>$, $\leq$, and $\geq$. With such new predicates, TGFDs can express denial constraints [21] and include a richer temporal semantics. Moreover, the time interval can be tied to the predicate as attribute values might have bigger changes over a large time window compared to smaller time windows.

6. We intend to explore the use of conditional graph patterns [46], where one can

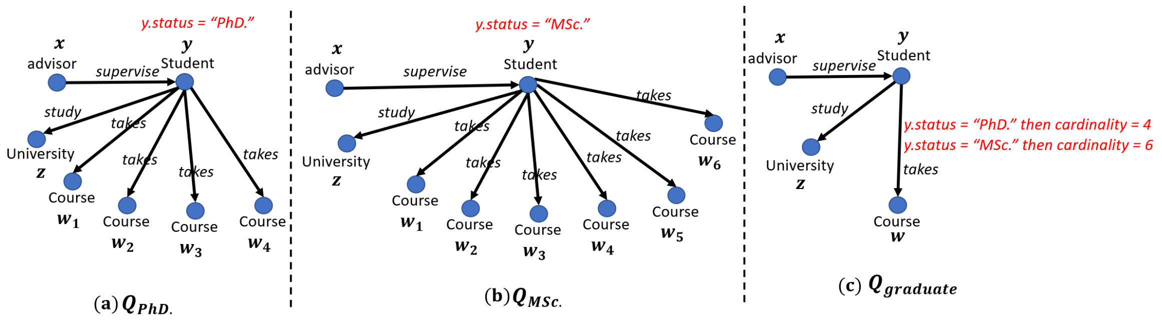


Figure 5.1: Example of the conditional graph pattern with cardinality on edges.

specify a simple condition on each edge such that the edge exists if and only if the condition is satisfied. Conditional graph patterns increase the expressiveness of TGFDs, and provide a succinct representation of graph patterns. Conditional graph patterns can be extended to contain edge cardinality as well. The cardinality of an edge equals to one by default, and one can express the cardinality via a natural number $[1, n]$. If the cardinality of an edge $n > 1$, then we expect n match occurrences of that edge. For example, Figure 5.1(a) shows a graph pattern to capture *PhD.* students that need to take four courses to graduate. Similarly, Figure 5.1(b) captures *MSc.* students that need to take six courses for graduation. However, one can extend and merge these two patterns into a conditional graph pattern with edge cardinality and define Figure 5.1(c), where the edge between *student* and *course* has the cardinality of four (resp. six) if the student is doing *PhD.* (resp. *MSc.*) at the university. This example shows that the use of conditional graph patterns increases the expressiveness of TGFDs. Moreover, the matching algorithm could benefit from this high level representation and re-use the matches of the simpler patterns, whenever necessary.

5.1.2 GKMiner

As next steps, we intend to extend GKMiner to mine conditional GKeys and study the the application of conditional GKeys to data linking. Recent work proposed extending graph patterns with conditions [46]. Such conditions can be modeled as a constant literal over the attributes of the nodes, or the edges in the pattern. Using conditional patterns, we can extend the definition of GKeys and then mine for such keys in the graphs. Moreover, the parallel discovery of GKeys in a distributed graphs is another avenue to extend GKMiner. We also intend to extend GKMiner to mine OGKs.

5.1.3 OGKs

As next steps, we intend to study the application of OGKs to knowledge fusion compared with graph embeddings and graph learning techniques that transform nodes, edges, and their attributes into vector space and then train a model for knowledge fusion. We intend to extend the OGK-EM to a parallel scalable algorithm to perform entity matching in a distributed setting. A parallel discovery algorithm of the OGKs would be another line of research for the next step. Moreover, similar to TGFDs, we would like to explore the use of OGKs in existing graph databases like Neo4j [99] and Wikibase [40]. However, these databases are based on RDF triples and support SPARQL query language [99, 82], while our techniques consider general graph patterns and general property graphs. SPARQL leverages the schema of the RDF and the semantics of the query for optimization purposes, while general graph pattern subsumes RDF and SPARQL. In order to use OGKs in such databases, we need to implement a general entity matching module based on general graph patterns that is far more intriguing than conventional subgraph isomorphism. Lastly, we intend to explore other ontological relationships to be used for OGKs, *e.g.,* (1) *inheritance* or *IS-A* relationship, where values form a taxonomy in the ontology and the distance between values are computed via the number of hops between the values and their least common ancestor in the taxonomy tree [27]; (2) *owl:disjointWith* , which states that the extensions of the two class descriptions have no individuals in common, therefore a distance equals ∞ would be the natural distance between the two values [9]; (3) *owl:differentFrom* relationship, that shows two properties are different from each other and are not equal [9].

Bibliography

[1] Full version. `http://www.cas.mcmaster.ca/~alipoum/full.pdf`.

[2] Crowdflower 2016 data science report. `https://www.kdnuggets.com/2016/04/crowdflower-2016-data-science-repost.html`, 2016.

[3] FDA approves first treatment for COVID-19. `https://www.fda.gov/news-events/press-announcements/fda-approves-first-treatment-covid-19`, 2020.

[4] Canada's COVID-19 economic response plan. `https://www.canada.ca/en/department-finance/economic-response-plan.html#businesses`, 2021.

[5] IMDB dataset. `ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/`, 2021.

[6] TGFD code and data repository. `https://github.com/TGFD-Project/TGFD/`, 2021.

[7] Institute for safe medication practices, FDA advise-ERR: Reported

medication errors with veklury (remdesivir) emergency use authorization. *https://www.ismp.org/resources/fda-advise-err-reported-medication-errors-veklury-remdesivir-emergency-use-authorization, 2022.*

[8] *JGraphT: A java library of graph theory data structures and algorithms.* `https://jgrapht.org/`, *2022.*

[9] *TWL semantics.* `https://www.w3.org/TR/owl-ref/#Semantics`, *2022.*

[10] *Yago Knowledge Base.* `https://yago-knowledge.org/downloads/yago-3`, *2022.*

[11] *Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning.* Proceedings of the VLDB Endowment, *9(4):336–347, 2015.*

[12] *S. Abiteboul, R. Hull, and V. Vianu.* Foundations of databases, *volume 8. Addison-Wesley Reading, 1995.*

[13] *W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in rdf. In* International Workshop on Semantics in Data and Knowledge Bases, *pages 23–39. Springer, 2010.*

[14] *M. Alipour-Langouri, Z. Zheng, F. Chiang, L. Golab, and J. Szlichta. Contextual data cleaning. In* 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), *pages 21–24. IEEE, 2018.*

[15] *M. Alipourlangouri. Temporal dependencies for graphs. In* Proceedings of the 2021 International Conference on Management of Data, *pages 2881–2883, 2021.*

[16] *M. Alipourlangouri and F. Chiang. Keyminer: Discovering keys for graphs. In* VLDB workshop TD-LSG, *2018.*

[17] *M. Alipourlangouri and F. Chiang. Discovery of keys for graphs [extended version].* arXiv preprint arXiv:2205.15547, *2022.*

[18] *M. Alipourlangouri, A. Mansfield, F. Chiang, and Y. Wu. Temporal graph functional dependencies–technical report.* arXiv preprint arXiv:2108.08719, *2021.*

[19] *R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens, et al. Pg-keys: Keys for property graphs. In* Proceedings of the 2021 International Conference on Management of Data, *pages 2423–2436, 2021.*

[20] *R. Angles and C. Gutierrez. Survey of graph database models.* ACM Computing Surveys (CSUR), *40(1):1, 2008.*

[21] *M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In* Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, *pages 68–79, 1999.*

[22] *M. Arenas and L. Libkin. A normal form for xml documents.* ACM Transactions on Database Systems (TODS), *29(1):195–232, 2004.*

[23] *M. Atencia, M. Chein, M. Croitoru, J. David, M. Leclère, N. Pernelle, F. Saïs, F. Scharffe, and D. Symeonidou. Defining key semantics for the rdf datasets. In* ICCS, *pages 65–78, 2014.*

[24] *M. Atencia, J. David, and F. Scharffe. Keys and pseudo-keys detection for web*

datasets cleansing and interlinking. In International Conference on Knowledge Engineering and Knowledge Management*, pages 144–153. Springer, 2012.*

[25] P. Augustyniak and G. Slusarczyk. Graph-based representation of behavior in detection and prediction of daily living activities. Computers in Biology and Medicine*, 95:261–270, 2018.*

[26] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. Proceedings of the VLDB Endowment*, 9(13):1457–1460, 2016.*

[27] S. Baskaran, A. Keller, F. Chiang, L. Golab, and J. Szlichta. Efficient discovery of ontology functional dependencies. In CIKM*, 2017.*

[28] C. Batini and M. Scannapieco. Data and Information Quality: Dimensions, Principles, and Techniques*. Springer-Verlag, Inc., 2016.*

[29] M. Bawa, B. F. Cooper, A. Crespo, N. Daswani, P. Ganesan, H. Garcia-Molina, S. Kamvar, S. Marti, M. Schlosser, Q. Sun, et al. Peer-to-peer research at stanford. Technical report, Stanford InfoLab, 2003.

[30] J. Birnick, T. Bläsius, T. Friedrich, F. Naumann, T. Papenbrock, and M. Schirneck. Hitting set enumeration with partial information for unique column combination discovery. Proceedings of the VLDB Endowment*, 13(12):2270–2283, 2020.*

[31] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The traveling salesman problem in bounded degree graphs. ACM Trans. Algorithms*, 8(2), 2012.*

[32] *T. Bleifuß, L. Bornemann, T. Johnson, D. V. Kalashnikov, F. Naumann, and D. Srivastava. Exploring change: A new dimension of data analytics.* Proc. VLDB Endow., *12(2):85–98, Oct. 2018.*

[33] *K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther. Pattern mining in frequent dynamic subgraphs. In* Sixth International Conference on Data Mining (ICDM'06), *pages 818–822. IEEE, 2006.*

[34] *P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for xml.* Computer networks, *39(5):473–487, 2002.*

[35] *A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In* ICDT, *2009.*

[36] *Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro. Aiding the detection of fake accounts in large scale social online services. In* Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, *pages 15–15. USENIX Association, 2012.*

[37] *A. Chatterjee and A. Segev. Data manipulation in heterogeneous databases.* ACM SIGMOD Record, *20(4):64–68, 1991.*

[38] *L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In* 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition, *pages 149–159, 2001.*

[39] *T. Deng, L. Hou, and Z. Han. Keys as features for graph entity matching. In* 2020 IEEE 36th International Conference on Data Engineering (ICDE), *pages 1974–1977. IEEE, 2020.*

[40] *D. Diefenbach, M. D. Wilde, and S. Alipio. Wikibase as an infrastructure for knowledge graphs: The eu knowledge graph. In* International Semantic Web Conference, *pages 631–647. Springer, 2021.*

[41] *X. Dong, X. He, A. Kan, X. Li, Y. Liang, J. Ma, Y. Xu, C. Zhang, T. Zhao, G. B. Saldana, S. Deshpande, A. M. Manduca, J. Ren, S. P. Singh, F. Xiao, H.-S. Chang, G. Karamanolakis, Y. Mao, Y. Wang, C. Faloutsos, A. McCallum, and J. Han. Autoknow: Self-driving knowledge collection for products of thousands of types.* Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, *2020.*

[42] *X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion.* VLDB, *7(10):881–892, 2014.*

[43] *V. Efthymiou, K. Stefanidis, and V. Christophides. Benchmarking blocking algorithms for web entities.* IEEE Transactions on Big Data, *2016.*

[44] *A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey.* IEEE Transactions on knowledge and data engineering, *19(1):1–16, 2006.*

[45] *G. Even, N. Garg, J. Könemann, R. Ravi, and A. Sinha. Min–max tree covers of graphs.* Operations Research Letters, *32(4):309–315, 2004.*

[46] *G. Fan, W. Fan, Y. Li, P. Lu, C. Tian, and J. Zhou. Extending graph patterns with conditions. In* Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, *pages 715–729, 2020.*

[47] *W. Fan. Data quality: From theory to practice.* Acm Sigmod Record*, 44(3):7–18, 2015.*

[48] *W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs.* Proceedings of the VLDB Endowment*, 8(12):1590–1601, 2015.*

[49] *W. Fan and F. Geerts.* Foundations of data quality management*, volume 4. Morgan & Claypool Publishers, 2012.*

[50] *W. Fan, C. Hu, X. Liu, and P. Lu. Discovering graph functional dependencies. In* SIGMOD*, 2018.*

[51] *W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In* Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, page 925–936, 2011.*

[52] *W. Fan, X. Liu, P. Lu, and C. Tian. Catching numeric inconsistencies in graphs. In* Proceedings of the 2018 International Conference on Management of Data*, pages 381–393, 2018.*

[53] *W. Fan and P. Lu. Dependencies for graphs.* ACM Transactions on Database Systems (TODS)*, 44(2):1–40, 2019.*

[54] *W. Fan, P. Lu, C. Tian, and J. Zhou. Deducing certain fixes to graphs.* Proceedings of the VLDB Endowment*, 12(7):752–765, 2019.*

[55] *W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching.* ACM Transactions on Database Systems (TODS)*, 38(3):1–47, 2013.*

[56] *W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns.* Proceedings of the VLDB Endowment, *8(12):1502–1513, 2015.*

[57] *W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In* SIGMOD International Conference on Management of Data, *pages 1843–1857, 2016.*

[58] *J. Ferlez, C. Faloutsos, J. Leskovec, D. Mladenic, and M. Grobelnik. Monitoring network evolution using mdl. In* 2008 IEEE 24th International Conference on Data Engineering, *pages 1328–1330. IEEE, 2008.*

[59] *P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In* Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, *pages 188–199, 2001.*

[60] *L. Galárraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient rdf processing. In* Proceedings of the 23rd International Conference on World Wide Web, *pages 267–268, 2014.*

[61] *M. Gardner and T. M. Mitchell. Efficient and expressive knowledge base completion using subgraph feature extraction. In* EMNLP 2015, *pages 1488–1498, 2015.*

[62] *M. R. Garey and D. S. Johnson.* Computers and Intractability; A Guide to the Theory of NP-Completeness. *W. H. Freeman and Co., 1990.*

[63] *S. Gurukar, S. Ranu, and B. Ravindran. Commit: A scalable approach to mining communication motifs from dynamic networks. In* Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, *pages 475–489, 2015.*

[64] *M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. Dream: distributed rdf engine with adaptive query planner and minimal communication.* Proceedings of the VLDB Endowment, *8(6):654–665, 2015.*

[65] *S. Hao, N. Tang, G. Li, and J. Feng. Discovering mis-categorized entities. In* ICDE, *2018.*

[66] *S. Hao, N. Tang, G. Li, and J. Li. Cleaning relations using knowledge bases. In* Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, *pages 933–944, 2017.*

[67] *A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations.* Proceedings of the VLDB Endowment, *7(4):301–312, 2013.*

[68] *J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional constraints on patterns with an application to the rdf data model. In* Foundations of Information and Knowledge Systems, *pages 250–269. Springer, 2014.*

[69] *J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional and constant constraints.* Annals of Mathematics and Artificial Intelligence, *76(3-4):251–279, 2016.*

[70] *J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia.* Artificial Intelligence, *194:28–61, 2013.*

[71] *J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs.* Proceedings of the VLDB Endowment, *4(11):1123–1134, 2011.*

[72] *Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies.* The computer journal, *42(2):100–111, 1999.*

[73] *I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication.* Foundations and Trends in Databases, *5(4):281–393, 2015.*

[74] *N. Ismail. The cost of bad data.* information-age, *2017.*

[75] *C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Extending existing dependency theory to temporal databases.* TKDE, *8(4):563–582, 1996.*

[76] *R. Knappe, H. Bulskov, and T. Andreasen. Perspectives on ontology-based querying.* International Journal of Intelligent Systems, *22(7):739–761, 2007.*

[77] *N. Lao, T. M. Mitchell, and W. W. Cohen. Random walk inference and learning in A large scale knowledge base. In* EMNLP, *pages 529–539, 2011.*

[78] *W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In* 2012 IEEE 28th International Conference on Data Engineering, *pages 666–677. IEEE, 2012.*

[79] *J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia.* Semantic Web, *2015.*

[80] *G. Levchuk, J. Roberts, and J. Freeman. Learning and detecting patterns in multi-attributed network data. In* AAAI Fall Symposium: Social Networks and Social Contagion, *2012.*

[81] *P. Lin, Q. Song, J. Shen, and Y. Wu. Discovering graph patterns for fact checking in knowledge graphs. In* International Conference on Database Systems for Advanced Applications, *pages 783–801. Springer, 2018.*

[82] *S. Link. Neo4j keys. In* International Conference on Conceptual Modeling, *pages 19–33. Springer, 2020.*

[83] *H. Ma, M. Alipourlangouri, Y. Wu, F. Chiang, and J. Pi. Ontology-based entity matching in attributed graphs.* Proceedings of the VLDB Endowment, *12(10):1195–1207, 2019.*

[84] *F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In* CIDR, *2014.*

[85] *D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results.* Proceedings of the VLDB Endowment, *3(1-2):208–219, 2010.*

[86] *B. Motik, I. Horrocks, and U. Sattler. Adding Integrity Constraints to OWL. In* OWL: Experiences and Directions (OWLED), *2007.*

[87] *M. H. Namaki, Y. Wu, Q. Song, P. Lin, and T. Ge. Discovering graph temporal association rules. In* CIKM, *pages 1697–1706. ACM, 2017.*

[88] *M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs.* Proceedings of the IEEE, *2016.*

[89] L. Noronha and F. Chiang. Discovery of temporal graph functional dependencies. In ACM International Conference on Information and Knowledge Management, Virtual Event, pages 3348–3352, 2021.

[90] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In Proceedings of the tenth ACM international conference on web search and data mining, pages 601–610, 2017.

[91] H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. Semantic web, 8(3):489–508, 2017.

[92] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pages 228–238, 2005.

[93] N. Pernelle, F. Saïs, and D. Symeonidou. An automatic key discovery approach for data linking. Web Semantics, 23:16–30, 2013.

[94] M. Pershina, M. Yakout, and K. Chakrabarti. Holistic entity matching across knowledge graphs. In 2015 IEEE International Conference on Big Data (Big Data), pages 1585–1590. IEEE, 2015.

[95] J. L. Peterson and A. Silberschatz. Operating system concepts. Addison-Wesley Longman Publishing Co., Inc., 1985.

[96] J. Pujara, H. Miao, L. Getoor, and W. Cohen. Knowledge graph identification. In International Semantic Web Conference, pages 542–557. Springer, 2013.

[97] M. R. Quillan. Semantic memory. Technical report, BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA, 1966.

[98] *R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In* Proceedings of Workshop on GRAph Data management Experiences and Systems, *pages 1–6, 2014.*

[99] *I. Robinson, J. Webber, and E. Eifrem.* Graph databases: new opportunities for connected data. *" O'Reilly Media, Inc.", 2015.*

[100] *N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In* Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, *pages 1055–1064. ACM, 2015.*

[101] *E. P. Shironoshita, M. T. Ryan, and M. R. Kabuka. Cardinality estimation for the optimization of queries on ontologies.* SIGMOD Rec., *36(2):13–18, 2007.*

[102] *D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem.* Mathematical programming, *62(1):461–474, 1993.*

[103] *Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In* Proceedings of the 32nd international conference on Very large data bases, *pages 691–702, 2006.*

[104] *P. Skavantzos, K. Zhao, and S. Link. Uniqueness constraints on property graphs. In* International Conference on Advanced Information Systems Engineering, *pages 280–295. Springer, 2021.*

[105] *Q. Song, P. Lin, H. Ma, and Y. Wu. Explaining missing data in graphs: A*

constraint-based approach. In 2021 IEEE 37th International Conference on Data Engineering (ICDE), *pages 1476–1487. IEEE, 2021.*

[106] T. Soru, E. Marx, and A.-C. Ngonga Ngomo. Rocker: A refinement operator for key discovery. In Proceedings of the 24th International Conference on World Wide Web, *pages 1025–1033, 2015.*

[107] F. M. Suchanek, M. Sozio, and G. Weikum. Sofie: a self-organizing framework for information extraction. In Proceedings of the 18th international conference on World wide web, *pages 631–640. ACM, 2009.*

[108] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, *pages 687–696, 2007.*

[109] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. arXiv preprint arXiv:1205.6691, *2012.*

[110] D. Symeonidou, V. Armant, N. Pernelle, and F. Saïs. Sakey: Scalable almost key discovery in rdf data. In International Semantic Web Conference, *pages 33–49. Springer, 2014.*

[111] D. Symeonidou, L. Galárraga, N. Pernelle, F. Saïs, and F. Suchanek. Vickey: Mining conditional keys on knowledge bases. In ISWC, pages 661–677, 2017.

[112] D. Vrandečić and M. Krötzsch. Wikidata: a free collaborative knowledgebase. Communications of the ACM, *57(10):78–85, 2014.*

[113] *X. Wang, Y. Xu, and H. Zhan. Extending association rules with graph patterns.* Expert Systems with Applications, *141:112897, 2020.*

[114] *S. Wasserman and K. Faust. Social network analysis: Methods and applications. 1994.*

[115] *Z. Wei, U. Leck, and S. Link. Discovery and ranking of embedded uniqueness constraints.* Proceedings of the VLDB Endowment, *12(13):2339–2352, 2019.*

[116] *J. Wijsen. Temporal dependencies. In* Encyclopedia of Database Systems, *2009.*

[117] *J. Wijsen, J. Vandenbulcke, and H. Olivie. Functional dependencies generalized for temporal databases that include object-identity. In* International Conference on the Entity-Relationship Approach, *volume 823 of* Lecture Notes in Computer Science, *pages 99–109, 1993.*

[118] *Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In* ICDE, *pages 697–708, 2013.*

[119] *Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In* ISWC, *2011.*

[120] *R. Zass and A. Shashua. Probabilistic graph and hypergraph matching. In* CVPR, *2008.*

[121] *Z. Zheng, M. Alipour, Z. Qu, I. Currie, F. Chiang, L. Golab, and J. Szlichta. Fastofd: Contextual data cleaning with ontology functional dependencies. In* EDBT, *pages 694–697, 2018.*

[122] Z. Zheng, L. Zheng, M. Alipourlangouri, F. Chiang, L. Golab, J. Szlichta, and S. Baskaran. Contextual data cleaning with ontology functional dependencies. ACM Journal of Data and Information Quality (JDIQ), *2022.*