Discovering Graph Functional Dependencies

WENFEI FAN, University of Edinburgh, Shenzhen Institute of Computing Sciences, and BDBC, Beihang Univ.
CHUNMING HU, Beihang University
XUELI LIU, College of Intelligence and Computing, Tianjin University
PING LU, BDBC, Beihang University

This article studies discovery of Graph Functional Dependencies (GFDs), a class of functional dependencies defined on graphs. We investigate the fixed-parameter tractability of three fundamental problems related to GFD discovery. We show that the implication and satisfiability problems are fixed-parameter tractable, but the validation problem is co-W[1]-hard in general. We introduce notions of reduced GFDs and their topological support, and formalize the discovery problem for GFDs. We develop algorithms for discovering GFDs and computing their covers. Moreover, we show that GFD discovery is feasible over large-scale graphs, by providing parallel scalable algorithms that guarantee to reduce running time when more processors are used. Using real-life and synthetic data, we experimentally verify the effectiveness and scalability of the algorithms.

CCS Concepts: • **Information systems** → **Inconsistent data**;

Additional Key Words and Phrases: Functional dependencies, graphs, discovery, validation, implication

ACM Reference format:
Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2020. Discovering Graph Functional Dependencies. *ACM Trans. Database Syst.* 45, 3, Article 15 (September 2020), 42 pages.
https://doi.org/10.1145/3397198

1 INTRODUCTION

Functional dependencies have recently been studied for property graphs [25, 27], referred to as *graph functional dependencies* (GFDs). As opposed to relational databases, real-life graphs often do not come with a schema. On such graphs, GFDs provide a primitive form of integrity constraints

Fan is supported in part by ERC Grant No. 652976, Royal Society Wolfson Research Merit Award No. WRM/R1/180014, EPSRC Grant No. EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Lu is supported in part by NSFC Grant No. 61602023. Liu is supported in part by NSFC Grant No. 61902274.

Authors' addresses: W. Fan, Univ. of Edinburgh, 10 Crichton Street, Edinburgh, UK, EH8 9AB, Shenzhen Institute of Computing Sciences, Shenzhen, China, BDBC, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191; email: wenfei@inf.ed.ac.uk; C. Hu, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191; email: hucm@buaa.edu.cn; X. Liu (corresponding author), College of Intelligence and Computing, Tianjin University, 135 Yaguan Road, Jinnan District, Tianjin, China, 300350; email: xueli@tju.edu.cn; P. Lu, BDBC, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191; email: luping@buaa.edu.cn.

0362-5915/2020/09-ART15 $15.00
https://doi.org/10.1145/3397198

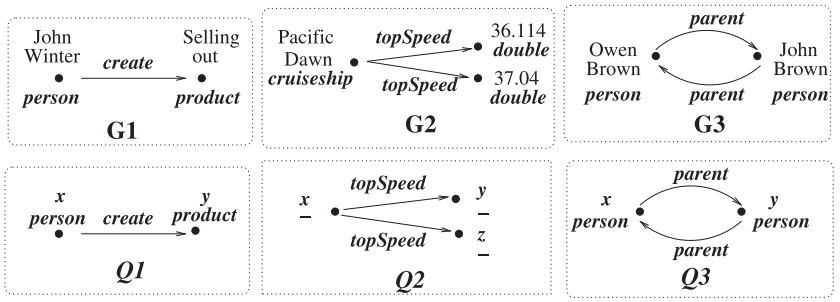


Fig. 1. Graphs and graph patterns.

to specify a fundamental part of the semantics of the schemaless graph-structured data. The need for GFDs is evident in specifying the integrity of graph entities, detecting spam in social networks, optimizing graph queries, and, in particular, consistency checking.

Example 1.1. Consistency checking is a major challenge to knowledge acquisition and knowledge base enrichment. Errors are common in real-world knowledge bases, e.g., those depicted in Figure 1.

(a) YAGO3 [48]: A person John Winter is given credit for producing film Selling Out, as shown in graph G_1 of Figure 1, although the film was actually created by producer Jack Winter, while John is a high jumper, not a movie producer.
(b) DBpedia [2]: The top speed of cruise ship "Pacific Dawn" is recorded as both 36.114 km/h and 37.04 km/h, which is impossible, since a ship can only have one top speed.
(c) DBpedia: John Brown and Owen Brown are claimed to be a parent of each other (graph G_3 in Figure 1).

Graph functional dependencies (GFDs) of References [25, 27] are able to catch these inconsistencies.

(1) Consider GFD $\varphi_1 = Q_1[x, y](y.\text{type} = \text{"film"} \rightarrow x.\text{type} = \text{"producer"})$. Here Q_1 is shown in Figure 1, and x and y are variables denoting two nodes in Q_1, each carrying an attribute type (not shown). On a graph G, φ_1 states that in any subgraph of G that matches Q_1 via isomorphism, if product y has type film, then the type of person x is producer. It catches the error in G_1.

(2) Consider GFD $\varphi_2 = Q_2[x, y, z](\emptyset \rightarrow y.\text{val} = z.\text{val})$, where pattern Q_2 is shown in Figure 1, $\emptyset$ denotes an empty set of literals, and val is an attribute of nodes y and z. It ensures that the value of the top speed of x must be unique. It catches the error in G_2. Note that nodes x, y and z are labeled with wildcard "_," which can match, e.g., cruiseship and double.

(3) Consider GFD $\varphi_3 = Q_3[x, y](\emptyset \rightarrow \text{false})$, where Q_3 is depicted in Figure 1, and false is a Boolean constant. It states that there exist no person entities x and y who are parent of each other, i.e., graph pattern Q_3 specifies an "illegal" structure. It catches the inconsistency in graph G_3.

To make practical use of GFDs, however, we need effective algorithms to discover meaningful GFDs from real-life graphs. This is challenging. A GFD $Q[\bar{x}](X \rightarrow Y)$ is a combination of a graph pattern Q and a functional dependency (FD) $X \rightarrow Y$, *positive* (specifying Y "entailed" by Q and X, e.g., φ_1 and φ_2), or *negative* (specifying "illegal" cases with false, e.g., φ_3). GFD discovery is much harder than discovering relational FDs [35, 57], as GFDs additionally require topological constraints Q. It is more challenging than graph pattern mining [18, 26, 34, 36, 47, 53], since it has to discover both positive and negative GFDs (e.g., φ_3). Worse yet, validation and implication

of GFDs are coNP-complete and NP-complete, respectively [27], which are embedded in GFD discovery.

Contributions. This article tackles these challenges, settles fundamental problems associated with GFD discovery, and develops parallel discovery algorithms with performance guarantees.

(1) We investigate three fundamental problems related to GFD discovery (Section 4). The satisfiability problem is to determine whether GFDs discovered are consistent, i.e., the GFDs have a model; implication is to decide whether a GFD discovered is "redundant," i.e., implied by a set of GFDs already known; and validation is to ensure that GFDs discovered from a graph G are satisfied by G.

We show that while the implication and satisfiability problems are fixed-parameter tractable [28], the validation problem is co-W[1]-hard [16]. However, we show that for GFDs with patterns of a bounded size, all these problems become tractable. These results are not only of theoretical interest but also help us formulate the discovery problem and develop practical discovery algorithms.

(2) We formalize the discovery problem for GFDs (Section 5). We introduce a notion of support for *positive and negative* GFDs in graphs to find "frequent" GFDs, and define reduced GFDs and GFD covers to exclude "redundant" GFDs. We show that the GFD support is anti-monotonic, which is more intriguing for graph-structured data than its counterpart in conventional data mining. Based on these, we formalize the discovery problem for GFDs, to strike a balance between the complexity of GFD discovery and the enhanced expressiveness of GFDs.

(3) We develop sequential algorithms for discovering GFDs of Reference [27] and for computing their covers (Section 6). In contrast to prior discovery algorithms, we combine pattern mining and FD discovery in a single process. Moreover, we provide effective pruning strategies. We also develop an algorithm for computing a cover of the set Σ of discovered GFDs, i.e., a minimal set of "non-redundant" GFDs that is equivalent to Σ. This algorithm involves the implication analysis of GFDs.

(4) We parallelize these algorithms for discovering GFDs in fragmented graphs, to cope with large-scale graphs (Section 7). We employ distributed incremental joins to balance the workload. We show that the algorithm is parallel scalable [45] relative to the sequential algorithm of (3), i.e., it guarantees to reduce response time with the increase of processors. Thus, it is feasible to discover GFDs from (possibly big) real-life graphs by adding processors when needed. We also develop a parallel scalable algorithm for computing a cover of discovered GFDs (Section 8).

(5) Using real-life and synthetic graphs, we experimentally evaluate the algorithms (Section 9). We find the following. (a) GFD discovery is parallel scalable. It is on average 3.78 times faster on real-life graphs when the number of processors n increases from 4 to 20. (b) GFD discovery is feasible in practice. On YAGO2, with 7.64M entities and edges, the sequential mining algorithm takes 1.3 h to discover 4-bounded GFDs (i.e., GFDs with graph patterns of at most four nodes; we picked patterns with four nodes, since real-life graph patterns are typically small [10, 30, 52]; see also Section 4). The performance is substantially improved by parallelization. When $n = 20$, it takes 913 s on average on real-life graphs (314 s on YAGO2), and 30 min on synthetic graphs with 30M nodes and 60M edges. (c) Computing GFD cover is parallel scalable. It is on average 1.75 times faster when n varies from 4 to 20. (d) Our algorithms find useful GFDs, positive and negative.

These algorithms yield a promising tool for exploring interesting dependencies in real-life graphs, including "axioms" for knowledge bases. We provide new techniques for parallel discovery

and verification of graph dependencies, e.g., vertical and horizontal spawning of GFD generations, handling of negative GFDs, distributed joins for workload balancing, and levelwise GFD checking to reduce isomorphism tests. We also settle fundamental problems for GFD discovery.

Related work. This work is an extension of Reference [7] by including the following: (1) a new section (Section 3.2) that shows how GFDs uniformly express axioms for enforcing integrity on knowledge bases; (2) the proofs of fundamental results, including fixed-parameter tractability (Theorem 4.2), the complexity of k-bounded GFDs (Proposition 4.3), the anti-monotonicity of support (Theorem 5.2), and the parallel scalability of GFD discovery (Theorem 7.1); (3) a new result extending Theorem 4.2 to separate the complexity of the validation problem for GFDs with connected patterns and with disconnected patterns; (4) details of label upgrading (Section 6.1) and load balancing (Section 7.2); (5) an approximation algorithm for load balancing in parallel GFD implication (Proposition 8.3); and (6) new experiments to evaluate the performance of our algorithms (Section 9).

We categorize other related work as follows.

FDs *for graphs*. In RDF, a predicate p is a functional property if for any entity x, there exists at most one y such that x is connected to y via predicate p [43, 59]. Based on the type of y, functional properties can be categorized as *value-based* (i.e., when y is a value), and *entity-based* (i.e., when y is an entity). For relational databases, FDs capture value-based functional properties.

Functional dependencies (FDs) have been studied for RDF [5, 11, 12, 15, 29, 32, 33, 46, 58]. Based on whether to support value equality or entity equality, these dependencies can also be categorized as value-based (i.e., enforcing value equality) and entity-based (i.e., enforcing entity equality). (1) Dependencies in References [46, 58] are entity-based. Using clustered values, Reference [58] defines FDs with path patterns; keys and foreign keys for RDF were studied in Reference [46] to identify entities (vertices). (2) Dependencies in References [11, 32] are value-based. Reference [58] is extended to support CFDs (conditional functional dependencies [21]) for RDF [32]. FDs are also defined in Reference [11], by mapping relations to RDF, using tree patterns. (3) Dependencies in References [5, 12, 15, 29, 33] are both value-based and entity-based. Based on triple patterns with variables, References [5, 15] define FDs with homomorphism. The implication and satisfiability problems for the FDs are shown decidable [5], but their complexity bounds are open; axiom systems are provided [15, 33] via relational encoding of RDF. AMIE [12, 29] extends association rules with conjunctive horn clauses for knowledge graph enhancement.

This work adopts the GFDs defined in Reference [27] for the following reasons. (a) GFDs extend relational FDs to capture value-based functional properties in graphs. They are defined for general property graphs, not limited to RDF. (b) GFDs support (cyclic) graph patterns with variables, e.g., φ_3 in Example 1.1, as opposed to References [11, 32, 58]. (c) GFDs support bindings of semantically related values like CFDs in relational database [21], e.g., φ_1, and a negative form with false like forbidding constraints in RDFS [15], e.g., φ_3, which cannot be expressed as the FDs of References [5, 11, 12, 15, 29, 32]. The need for supporting these is evident in consistency checking, as indicated by axioms for knowledge bases, and by the experience of cleaning relational data [21]. In contrast, functional properties in References [43, 59] cannot associate entities with constants, and AMIE does not support graph pattern matching via subgraph isomorphism, constant-value binding, negative rules or rules with wildcard.

An extension of the GFDs of Reference [27] was defined in Reference [25], referred to as GEDs. GEDs aim to support both value-based functional dependencies, e.g., GFDs of Reference [27], and entity-based functional dependencies, e.g., keys for graphs [20], in a uniform format under the semantics of graph homomorphism for pattern matching. For GEDs, the satisfiability, implication

and validation problems remain coNP-complete, NP-complete and coNP-complete, respectively; and a sound and complete axiom system is developed for finite implication analysis of GEDs [25]. To simplify the discussion, we focus on the original form of GFDs of Reference [27] in this article; nonetheless, the techniques developed in this work can be extended to discover GEDs of Reference [25]. Note that neither Reference [27] nor Reference [25] considers GFD discovery.

Dependency discovery. Discovery algorithms have been well studied for relational dependencies, e.g., FDs [35, 57], CFDs [13, 22], and denial constraints [14]. As remarked earlier, GFD discovery is much harder. Closer to this work are algorithms for discovering FDs over graphs [32, 58]. The method of Reference [58] first pre-clusters property values; it then adapts the levelwise process of TANE [35] to discover FDs defined with path patterns over RDF. It is extended in Reference [32], which first enumerates frequent graph structures, and then adopts CFDMiner [22] to mine CFDs in each subgraph found.

To the best of our knowledge, no prior work has studied (a) discovery of dependencies with (possibly cyclic) patterns, which involves enumeration of isomorphic subgraph mappings and is inherently intractable, as opposed to path patterns [32, 58], (b) negative GFDs, which demand a support quite different from the conventional notion for graph patterns, but are particularly useful for consistency checking in knowledge bases [43], (c) dependencies whose validation is intractable, (d) topological support and reduced dependencies, and (e) parallel discovery algorithms, not to mention parallel scalability. GFD discovery is unique in these aspects.

Following the practice of conventional relational FD mining, we discover GFD candidates from (possibly dirty) graphs. Such candidate GFDs are subject to inspection and selection by domain experts, before the GFDs are used as, e.g., data quality rules. We aim to find meaningful GFDs that are non-redundant and frequent by defining reduced GFDs and their topological support.

Graph pattern mining. Related to GFD discovery is graph pattern mining from graph databases [34, 36, 39]. Apriori [36] and pattern-growth [34] methods expand a frequent pattern by adding new nodes and edges. The method of Reference [39] connects two graphs in a graph database via Pearson correlation. Multiobjective subgraph mining [53] optimizes subgraphs via skyline processing. As observed in Reference [38], pattern mining over graph databases does not help GFD discovery, since the anti-monotonicity of the support of References [34, 36, 39] no longer holds over a single graph.

Closer to this work are also mining techniques for a single graph [18, 26, 47, 55]. GRAMI [18] considers graph patterns without edge labels, and models isomorphic subgraph enumeration in terms of constraint satisfaction. The method of Reference [47] mines frequent subgraphs via a two-step filter-and-refinement process, in MapReduce. Arabesque [55] uses "pattern-centric" MapReduce programming to parallelize pattern mining. The method of Reference [26] discovers top-k diversified association rules of the form $Q \Rightarrow p$ defined with a graph pattern Q and a single edge (predicate) p.

GFD discovery differs from the prior work in the following. (a) It requires both graph pattern mining and FD mining, not just pattern mining. We develop new data structures and techniques to combine the two into a single process; indeed, we find that separating the two processes does not allow us to scale with large-scale graphs. (b) No prior work has studied "negative patterns" coupled with FDs. For such GFDs, the computation of the support is more intriguing than a direct isomorphic counting of References [18, 26]. (c) To find a cover of discovered GFDs, we have to check GFD implication, an intractable problem, which is not an issue for graph pattern mining. (d) We offer parallel scalability, a performance guarantee not found in the prior algorithms except

[26, 27]. Our algorithms differ from References [26, 27] in problem statements and methods. We balance the workload via distributed and incremental joins, while [26, 27] require special treatments for skewed graphs.

Organization. The remainder of the article is organized as follows. Section 2 reviews basic notations of graphs and graph patterns. Section 3 presents the GFDs of Reference [27], and shows how GFDs express axioms for consistency checking in knowledge bases. Section 4 investigates the fixed-parameter tractability of the three fundamental problems for GFDs. Section 5 introduces the support of GFDs, proves its anti-monotonicity, and formalizes the discovery problem for GFDs. Section 6 presents sequential algorithms for discovering GFDs and GFD implication. Sections 7 and 8 develop parallel scalable algorithms for discovering GFDs and deducing GFD implication, respectively. An experimental study is reported in Section 9, followed by topics for future work in Section 10.

2 GRAPHS AND GRAPH PATTERN MATCHING

We first review basic notations that we will use to define GFDs.

Assume three countably infinite sets Θ, Υ, and U of labels, attributes and constants, respectively. We consider directed graphs $G = (V, E, L, F_A)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set in which (v, v') is an edge from node v to v'; (3) each node $v \in V$ (respectively, edge $e \in E$) is labeled $L(v) \in \Theta$ (respectively, $L(e) \in \Theta$); and (4) $F_A()$ is function such that for each node v, $F_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where a_i is a constant in U, A_i is an *attribute* of v with $A_i \in \Upsilon$, written as $v.A_i = a_i$; and $A_i \neq A_j$ if $i \neq j$. The attributes carry the content of the node as in property graphs, social networks and knowledge bases.

We will use two notions of subgraphs, presented as follows.

- A graph $G' = (V', E', L', F'_A)$ is a *subgraph of* $G = (V, E, L, F_A)$ if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$ (respectively, edge $e \in E'$), $L'(v) = L(v)$ (respectively, $L'(e) = L(e)$), and $F'_A(v) = F_A(v)$.
- A subgraph G' of G is *induced by* a set V' of nodes if E' consists of all the edges e in G such that the endpoints of e are both in V'.

Example 2.1. Three graphs are shown in Figure 1. Graph G_1 depicts that a high jumper named "John Winter" creates a film named "Selling out." It consists of (1) two nodes v_1 and v_2, which are labeled person and product, respectively; and (2) one edge (v_1, v_2) labeled create. Moreover, both v_1 and v_2 have two attributes name and type (not shown in the figure), and the values of these attributes are defined as follows: $F_A(v_1)$.name = "John Winter," $F_A(v_1)$.type = "high jumper," $F_A(v_2)$.name = "Selling out," and $F_A(v_2)$.type = "film." Here $F_A(v_1)$.name = "John winter" and $F_A(v_1)$.type = "high jumper" mean that the person represented by v_1 is a high jumper, and his name is John Winter; similar for attributes of v_2. Graphs G_2 and G_3 can be interpreted similarly.

Graph patterns. A *graph pattern* is a nonempty directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (respectively, E_Q) is a set of pattern nodes (respectively, edges); (2) L_Q is a function assigning a label $L_Q(u)$ (respectively, $L_Q(e)$) to each node $u \in V_Q$ (respectively, edge $e \in E_Q$); we allow u and e to be labeled with wildcard "_," i.e., labels $L_Q(u)$ and $L_Q(e)$ can be "_"; (3) $\bar{x}$ is a set of variables such that its arity $|\bar{x}|$ is equal to the number $|V_Q|$ of nodes, where $\bar{x}$ is used to represent the nodes in V_Q; and (4) μ is a bijective mapping from $\bar{x}$ to V_Q that assigns a distinct variable to each node v in V_Q. For $x \in \bar{x}$, we use $\mu(x)$ and x interchangeably when it is clear in the context. We use variables $\bar{x}$ to denote pattern nodes to separate them from nodes in graphs, for the ease of defining matches as valuations of variables.

Example 2.2. Figure 1 shows three graph patterns: (1) Q_1 depicts a person entity connected to a product entity with an edge labeled create; here μ maps x to person and y to product; (2) Q_2 specifies that the top speed of an entity x is recorded as both y and z, while all of x, y and z are labeled with wildcard "_"; and (3) Q_3 is a pattern of person entities; it is cyclic.

Pattern matching via subgraph isomorphism. For labels ℓ and ℓ', we write $\ell \prec \ell'$ if $\ell \in \Theta$ and ℓ' is "_." For instance, country $\prec$ _. We write $\ell \preceq \ell'$ if $\ell \prec \ell'$ or $\ell = \ell'$.

A *match* of pattern Q in graph G is a subgraph $G' = (V', E', L', F'_A)$ of G that is "isomorphic" to Q. That is, there exists a *bijective function* h from the set V_Q of nodes nodes in G to the set V' of nodes in G such that (1) for each node $u \in V_Q$, $L'(h(u)) \preceq L_Q(u)$; and (2) $e = (u, u')$ is an edge in Q if and only if (written as *iff*) $e' = (h(u), h(u'))$ is an edge in G' and moreover, $L'(e') \preceq L_Q(e)$.

Intuitively, a wildcard "_" indicates generic entities or properties, and hence may map to any label in the alphabet Θ. We also denote the match as a set $h(\bar{x})$ in the sequel, consisting of $h(x)$ (i.e., $h(\mu(x))$) for all variables $x \in \bar{x}$.

Example 2.3. There exists a match of pattern Q_2 in graph G_2 of Figure 1, via a mapping h_2 defined as follows: $x \mapsto$ Pacific Dawn, $y \mapsto 36.114$, and $z \mapsto 37.04$. The match contains no extra edge besides those induced by h_2; similarly for patterns Q_1 in G_1 and Q_3 in G_3.

Note that the wildcards "_" in pattern Q_2 map to cruiseship and double in G_2. That is, distinct wildcards (on different nodes or edges) may have different label valuations.

3 GRAPH FUNCTIONAL DEPENDENCIES

We next review the GFDs introduced in Reference [27], and give a case study of GFDs.

3.1 Functional Dependencies for Graphs

A *graph functional dependency* (GFD) is defined as $\varphi = Q[\bar{x}](X \rightarrow Y)$ [27], where

- $Q[\bar{x}]$ is a graph pattern, referred to as the *pattern* of φ; and
- X and Y are two (possibly empty) sets of literals of $\bar{x}$.

Here a *literal* of $\bar{x}$ has the form of either $x.A = c$ or $x.A = y.B$, where $x, y \in \bar{x}$, A and B denote attributes, and c is a constant in U. Intuitively, φ is a combination of two constraints:

- a *topological constraint* imposed by pattern Q, and
- *attribute dependency* specified by $X \rightarrow Y$ (recall that attributes are not specified in Q).

Here Q specifies the scope of φ such that the dependency $X \rightarrow Y$ is imposed only on matches of Q. Literals $x.A = c$ enforce bindings of semantically related constants, along the same lines as relational CFDs [21]. As syntactic sugar, we allow Y to be Boolean false, as it can be expressed as, e.g., $y.A = c \wedge y.A = d$ for distinct constants c and d, for any variable $y \in \bar{x}$ and attribute A of y.

For instance, Example 1.1 shows three GFDs φ_1, φ_2 and φ_3.

Semantics. Consider a match $h(\bar{x})$ of pattern Q in a graph G, and a literal $x.A = c$. We say that $h(\bar{x})$ *satisfies* the literal if *there exists* an attribute A at the node $v = h(\mu(x))$ such that $v.A = c$; similarly for $x.A = y.B$. We write $h(\bar{x}) \models X$ if $h(\bar{x})$ satisfies *all* the literals in a set X of literals.

We write $h(\bar{x}) \models X \rightarrow Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$, i.e., if $h(\bar{x}) \models X$, then $h(\bar{x}) \models Y$.

A graph G *satisfies* GFD φ, denoted by $G \models \varphi$, if *for all* matches $h(\bar{x})$ of Q in G, $h(\bar{x}) \models X \rightarrow Y$. Graph G *satisfies* a set Σ of GFDs, denoted by $G \models \Sigma$, if for all $\varphi \in \Sigma$, $G \models \varphi$.

To check whether $G \models \varphi$, we need to examine all matches of Q in G. Moreover, we consider schemaless graphs and hence, have to accommodate the semi-structured nature of such graphs:

(1) For $x.A = c$ in X, if $h(x)$ has *no* attribute A, then $h(\bar{x})$ satisfies $X \rightarrow Y$. Indeed, node $h(x)$ is not required to have attribute A, since graphs have no schema.
(2) In contrast, if $x.A{=}c$ is in Y and $h(\bar{x}) \models Y$, then $h(x)$ must have attribute A by the definition of satisfaction; similarly for $x.A{=}y.B$. In this way, GFDs can enforce the existence of attributes.
(3) If X is $\emptyset$, then $h(\bar{x}){\models}X$ for any match $h(\bar{x})$ of Q. When $Y = \emptyset$, Y is constantly true, and φ is trivial.

Intuitively, if a match $h(\bar{x})$ of Q in G *violates* $X \rightarrow Y$, i.e., $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$, then the subgraph induced by $h(\bar{x})$ is inconsistent, i.e., its entities have inconsistencies.

Negative GFDs. A GFD is called *negative* if it has the form $Q[\bar{x}](X \rightarrow \text{false})$. It is *positive* otherwise.

There are two cases of negative GFDs φ.

(a) When $X = \emptyset$, i.e., φ has the form $Q[\bar{x}](\emptyset \rightarrow \text{false})$; it says that in a graph G, there exists *no* match of Q, i.e., Q specifies an "illegal" structure, e.g., Q_3 of Figure 1.
(b) When $X \neq \emptyset$, it states that the combination of pattern Q and condition X is "inconsistent." Note that when X is conflicting (e.g., when X includes $x.A = c$ and $x.A = d$ for distinct c and d), the corresponding GFD is trivial. Thus, in the sequel, we consider non-conflicting X.

Example 3.1. In Figure 1, $G_2 \not\models \varphi_2$. Indeed, a match of Q_2 in G_2 is h_2: $x \mapsto$ Pacific Dawn, $y \mapsto$ 36.114, and $z \mapsto$ 37.04, as we have seen in Example 2.3. Here X in φ_2 is trivially true ($\emptyset$) but $y.\text{val} \neq z.\text{val}$ (36.114 vs. 37.04). Hence, φ_2 finds G_2 inconsistent. Similarly, $G_1 \not\models \varphi_1$ and $G_3 \not\models \varphi_3$. GFD φ_3 is negative, while φ_1 and φ_2 given in Example 1.1 are positive.

Normal form. We consider *w.l.o.g.* positive GFDs of the form $\varphi = Q[\bar{x}](X \rightarrow l)$, where l is a literal, i.e., Y in φ has a single literal l. To see this normal form does not lose generality, observe that a positive GFD$Q[\bar{x}](X \rightarrow Y)$ is equivalent to a set of GFDs $Q[\bar{x}](X \rightarrow l)$ for each $l \in Y$. More specifically, this can be verified by using the following notations.

A set Σ of GFDs *implies* another GFD φ, denoted by $\Sigma{\models}\varphi$, if for all graphs G, $G{\models}\Sigma$ implies $G \models \varphi$.

A set Σ of GFDs is *equivalent to* a set Σ', denoted by $\Sigma{\equiv}\Sigma'$, if $\Sigma{\models}\varphi'$ for all $\varphi'{\in}\Sigma'$ and vice versa.

In the sequel, we consider positive GFDs in the normal form.

Remark. One might want to extend GFDs by supporting richer graph patterns, e.g., regular expressions as edge labels [23]. This would, however, increase the complexity for reasoning about the constraints. For instance, the implication problem (Section 4) would become PSPACE-hard.

The notations and the sections where their definitions are introduced are summarized in Table 1.

3.2 Expressing Axioms for Knowledge Enrichment

GFDs can express axioms proposed for consistency checking in knowledge bases. Hence, we can reason about the axioms in the uniform framework of GFDs. Recall RDF triples (s, p, o), as an edge from s to o labeled p, where s is a *subject*, p is a *predicate*, and o is an *object*.

Disjoint. The axiom states that two disjoint classes cannot have a common instance [8, 43, 59]. It is to capture inconsistencies such as (Pride_Library, type, place), (Pride_Library, type, agent), (agent, disjointWith, place) found in DBPedia. It has been incorporated into DBPedia and YAGO by using a disjointWith statement of OWL 2. It can be expressed as the following GFD:

$$\varphi_4 = Q_4[x, y, z](\emptyset \rightarrow \text{false}),$$

Table 1. Notations

symbols	notations	Section
G	graph (V, E, L, F_A)	2
$Q[\bar{x}]$	graph pattern (V_Q, E_Q, L_Q, μ)	2
φ, Σ	GFD $\varphi = (Q[\bar{x}], X \rightarrow Y)$, Σ is a set of GFDs	3.1
negative GFDs	$(Q[\bar{x}], X \rightarrow \text{false})$ when X is satisfiable	3.1
$h(\bar{x}) \models X \rightarrow Y$	a match $h(\bar{x})$ of Q satisfies $X \rightarrow Y$	3.1
$G \models \Sigma$	graph G satisfies a set Σ of GFDs	3.1
$\Sigma \models \varphi$	Σ implies another GFD φ	3.1
$Q[\bar{x}] \ll Q'[\bar{x}']$	pattern Q reduces another pattern Q'	5.1
$\text{supp}(\varphi, G)$	the support of GFD φ in graph G	5.2
$t(\lvert G\rvert, k, \sigma)$	sequential complexity for GFD discovery	6.4
$T(\lvert G\rvert, n, k, \sigma)$	parallel complexity with n processors	7.1

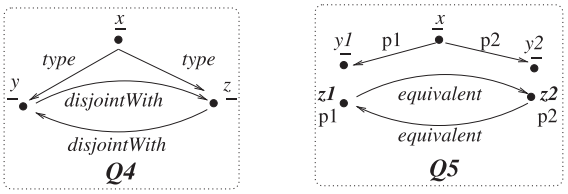


Fig. 2. GFDs for knowledge enrichment axioms.

where pattern Q_4 is given in Figure 2. It ensures that for any x, if x has disjoint types y and z, then Q_4 should not find a match in any graph. It is generic with wildcard as labels.

A related axiom states that an entity cannot connect to an object with two disjoint properties [43]. This axiom can also be enforced by a GFD similar to φ_4 above.

Asymmetric. Property p is *asymmetric* if (x, p, y) cannot co-exist with (y, p, x) [43]. This property can be expressed as GFDs similar to φ_3, to catch inconsistencies such as (Walter_Maule, child, Henry_Maule) and (Henry_Maule, child, Walter_Maule) found in DBpedia.

A related axiom is for irreflexive properties [43, 59]: If p is irreflexive, then for any x, (x, p, x) is not allowed, i.e., x cannot be connected to itself via an edge labeled p. This axiom can be expressed as a GFD, catching errors such as (George_McGovern, child, George_McGovern) in DBpedia.

Equivalence. It states that if two predicates p_1 and p_2 are marked equivalent, then for any entity x, there cannot exist y_1 and y_2 such that (x, p_1, y_1) and (x, p_2, y_2) coexist while y_1 and y_2 have distinct values. This is expressed as a GFD defined as follows:

$$\varphi_5 = Q_5[x, y_1, y_2, z_1, z_2](\emptyset \rightarrow y_1.\text{val} = y_2.\text{val}),$$

with disconnected pattern Q_5 shown in Figure 2. Similarly, synonym properties can be specified.

Inheritance. As shown in Reference [27], general properties of subclass and is_a inheritance relation can be readily enforced by GFDs with wildcard "_." In addition, as shown in Reference [27], GFDs subsume relational FDs and CFDs as special cases, when tuples are represented as vertices.

Functional. The axiom states that if a predicate p is a functional property, then for any entity x, there exists at most one y such that x is connected to y via predicate p. It is implemented by DBPedia and YAGO [43, 59]. Value-based functional properties defined in RDF can be enforced by

GFDs of the form φ_2 given in Example 1.1 (see Section 1). Here, φ_2 enforces two nodes to have the same value, but does not merge them, since the nodes may have different values for, e.g., range attributes. Value-based functional properties are common in knowledge bases. For example, 48 functional properties are specified in YAGO2, and among them, 41 are value-based.

Putting these together, we can see that the variety of axioms proposed separately for knowledge bases can be expressed as GFDs, and be reasoned about in the uniform framework of GFDs.

4 FUNDAMENTAL PROBLEMS FOR GFDS

We next revisit four fundamental problems for GFDs, which are related to GFD discovery.

(1) A set Σ of GFDs is *weakly satisfiable* if there exists a non-empty graph G such that $G \models \Sigma$.
The *weak satisfiability problem* is to decide whether a given set Σ of GFDs can be satisfied.

(2) A set Σ of GFDs is *satisfiable* if there exists a graph G such that (a) $G \models \Sigma$, and (b) when not all GFDs in Σ are negative, there exists a positive GFD $Q[\bar{x}](X \to l)$ in Σ such that Q has a match in G.

Different from the weak satisfiability problem, the satisfiability problem is not only to decide whether there exist graphs satisfying Σ but also to check whether there exists some GFD φ in Σ such that φ can be applied to graphs without conflicts. Note that when there exist graphs satisfying Σ, it is still possible that no GFD can be applied without conflicts, as shown by the example below.

Example 4.1. Consider $\Sigma = \{\varphi_1, \varphi_2\}$, where $\varphi_1 = Q[x](\emptyset \to x.A = 1)$ and $\varphi_2 = Q[x](\emptyset \to x.A = 2)$. Here Q consists of a single pattern node x labeled "o." It is easy to see that these two GFDs conflict with each other, since they attempt to assign two distinct constants to the same attribute of the same node. Therefore, no GFD from Σ can be applied without conflicts. Observe the following.

(a) This set Σ of GFDs is weakly satisfiable, since there exist non-empty graphs satisfying Σ. To see this, consider a graph G, which consists of a single node u labeled "τ" with $o \neq \tau$. It is easy to verify that there exists no match of pattern Q in graph G. So $G \models \Sigma$.

(b) In contrast, Σ is not satisfiable. Assume by contradiction that there exists a graph G such that $G \models \Sigma$ and Q has a match in G. Since Q has a match in G, there exists a node u labeled "o" in G. Because $G \models \Sigma$, we have that $G \models \varphi_1$ and $G \models \varphi_2$. From $G \models \varphi_1$, we know that $u.A = 1$. But from $G \models \varphi_2$, we also have that $u.A = 2$, a contradiction. Hence, Σ is not satisfiable.

(3) The *implication problem* for GFDs is to determine, given a set Σ of GFDs and another GFD φ, whether $\Sigma \models \varphi$, i.e., whether φ is a logical consequence of Σ.

(4) The *validation problem* is to decide, given a set Σ of GFDs and a graph G, whether $G \models \Sigma$, i.e., there exists no violation of the GFDs in graph G.

We remark the following about these fundamental problems.

- The weak satisfiability analysis is to check whether the discovered GFDs can be satisfied. The satisfiability problem is to check whether some discovered GFDs can be applied to a graph without conflicts. More specifically, its condition (b) is to ensure that when there are positive GFDs in Σ, at least one positive GFD in Σ can be applied to nonempty graphs. Note that negative GFDs do not necessarily find a match in a graph, since their patterns may specify abnormal cases and such GFDs are used to catch abnormal situations
- The implication analysis is needed for computing a cover of discovered GFDs (see Section 8). It also helps us eliminate redundant GFDs and reduce the cost of applying the GFDs.

- In parallel GFD discovery, the validation analysis is a must, since we have to ensure that GFDs discovered from a fragment of a distributed graph G are satisfied by the entire G (see Section 7). This also helps us catch inconsistencies in G by using Σ [27].

Fixed-parameter tractability. It was shown that the satisfiability, implication, and validation problems for GFDs are coNP-complete, NP-complete, and coNP-complete, respectively [27].

We next study their fixed-parameter tractability. An instance of a *parameterized problem* P is a pair (x, k), where x is an input as in the classical complexity theory, and k is a parameter that characterizes the structure of x. It is called *fixed-parameter tractable*, denoted by FPT, if there exist a computable function f and an algorithm for P such that for any instance (x, k) of P, it takes $O(f(k) \cdot |x|^c)$ time to find the solution, where c is a constant (see, e.g., Reference [16] for details). Intuitively, if k is small, then it is feasible to solve the problem efficiently, although $f(k)$ could be, e.g., 2^k.

For a set Σ of GFDs, we use k to denote $\max(|\bar{x}|)$ for all $Q[\bar{x}](X \to l)$ in Σ, i.e., the number of vertices in Q. We parameterize the implication problem by k as follows:

- Input: A set Σ of GFDs and a GFD φ.
- Parameter: $k = \max\{|\bar{x}| \mid Q[\bar{x}](X \to l) \in \Sigma \cup \{\varphi\}\}$.
- Question: Does $\Sigma \models \varphi$?

Similarly, we can parameterize the satisfiability and validation problems by k.

One might think that these problems are in FPT. Unfortunately, this is not the case.

Theorem 4.2. *For* GFDs, *(1) the weak satisfiability problem is in* PTIME*; (2) the implication and satisfiability problems are in* FPT *with parameter* k*. However, (3) the validation problem is co-*W[1]*-hard with parameter* k*; but (4) when* GFDs *are defined with connected patterns and when the maximum degree of the nodes in graph* G *is bounded by another parameter* d*, the validation problem is in* FPT *with parameters* k *and* d*.*

Here W[1] is the class of all parameterized problems that are FPT-reducible to a certain weighted satisfiability problem (see Reference [16]). It is conjectured that W[1]-hard problems are not fixed-parameter tractable. This tells us that the validation analysis remains nontrivial for GFDs defined with possibly disconnected patterns even when k is small. Here a pattern $Q[\bar{x}]$ is *connected* if for every pair of nodes in $Q[\bar{x}]$ there exists an undirected path between them. For example, patterns Q_1, Q_2, Q_3 of Figure 1 and Q_4 of Figure 2 are connected, while Q_5 of Figure 2 is disconnected. For the validation problem to be FPT, it additionally requires the maximum degree d of graphs as a parameter.

We remark that maximum degree d is not too large a parameter in practice. For instance, world road network [6] and routing networks [9] have a maximum degree of 9 and 153, respectively. GFDs can be used to detect abnormal patterns in world road network and routing networks [41, 51]. For example, we can use graph patterns to express traffic patterns during DoS (denial-of-service) attacks, and use GFDs of the form φ_3 (Example 1.1) to detect new DoS attacks [41]. In light of these, FPT problems parameterized with k and d still find practical applications. For the same reason, several other problems take k and d as parameters and become FPT [37, 42, 49, 50].

Proof. (1) Weak satisfiability. Below, we first provide a characterization for the weak satisfiability problem. We then show that the characterization can be verified in PTIME.

Characterization. The idea of the characterization is as follows. Given a set Σ of GFDs, we first define a graph G. We then compute a set Σ_G of GFDs embedded in G and a set enforced(Σ_G) of

literals enforced by Σ_G. Finally, we show that enforced(Σ_G) is not conflicting if and only if Σ is weakly satisfiable. More specifically, we define Σ_G, enforced(Σ_G) and conflicting sets as follows.

(a) We first give the construction of G. Let τ be a label that does not appear in Σ. Then graph G consists of only one isolated node v, which is labeled with τ. Since the weak satisfiability problem only requires non-empty graphs, it suffices to consider graphs with only one node.

(b) We borrow the notions of Σ_G and enforced(Σ_G) from Reference [27].

(•) For the constructed graph G, the set of GFDs *embedded in* G, denoted by Σ_G, consists of GFDs of the form $\varphi_1 = Q_1[\bar{x}_1](f(X_1) \rightarrow f(l_1))$, such that

(i) there exists a GFD $\varphi_1' = Q_1[\bar{x}_1](X_1 \rightarrow l_1)$ in Σ, and
(ii) f is a match from Q_1 into G.

Here $f(X_1)$ is obtained by replacing x by $f(x)$ for every node x appearing in X_1; similarly for $f(l_1)$. Note that such a match may exist when Q_1 carries wildcard "_."

Intuitively, Σ_G consists of GFDs φ_1 in Σ such that Q_1 can be mapped to G and hence, φ_1 has to be enforced on every match of Q in G that satisfies Σ.

(•) The set enforced(Σ_G) consists of literals that have to be enforced on each match of Q in graph G if $G \models \Sigma$. More specifically, it is inductively defined as follows.

A literal l_1 is in enforced(Σ_G) if one of the following conditions is satisfied:

(i) either $Q_1[\bar{x}_1](\emptyset \rightarrow l_1)$ is in Σ_G; or
(ii) $Q_1[\bar{x}_1](X_1 \rightarrow l_1)$ is in Σ_G, and all literals in X_1 can be deduced from enforced(Σ_G) by the transitivity of equality literals; that is, for each literal $x.A = y.B$ in X_1, there exists a chain of equality literals $x.A = x_1.A_1, x_1.A_1 = x_2.A_2, \ldots, x_n.A_n = y.B$ such that both $x_i.A_i = x_{i+1}.A_{i+1}$ $(0 \leq i \leq n-1)$ and $x_n.A_n = y.B$ are in enforced(Σ_G); similarly for each literal $x.A = c$.

Taking a literal l as a pair $(x.A, c)$ or $(x.A, y.B)$, enforced(Σ_G) is an equivalence relation on attributes and constants, such that if l is $x.A = y.B$ (respectively, $x.A = c$), then $x.A$ and $y.B$ (respectively, $x.A$ and c) are in the same equivalence class; i.e., enforced(Σ_G) is reflexive, symmetric, and transitive.

We say that enforced(Σ_G) is *conflicting* if there exist an attribute $x.A$ and two distinct constants c and d such that both $x.A = c$ and $x.A = d$ are in the equivalence relation enforced(Σ_G).

(c) We now show that enforced(Σ_G) is not conflicting if and only if Σ is weakly satisfiable [27].

($\Rightarrow$) Assume that enforced(Σ_G) is not conflicting. We next show that Σ is weakly satisfiable. That is, we enforce literals in enforced(Σ_G) on graph G, and show that $G \models \Sigma$.

We first enforce the literals in enforced(Σ_G). Observe that G has only one node v. Then, we can define the attributes of G as follows. For each attribute $v.A$ in enforced(Σ_G), (i) if there exists a constant c such that $v.A = c$ can be deduced from enforced(Σ_G) by the transitivity of equality literals (see the computation of enforced(Σ_G)), then set $v.A = c$ in G; (ii) otherwise, let $v.A_1, \ldots, v.A_n$ be all attributes in the equivalence class of $v.A$, then set $v.A = v.A_1 = \cdots = v.A_n = \#$ for a new distinct constant # that is not in enforced(Σ_G), and add the equality $v.A = \#$ to enforced(Σ_G).

We next show that $G \models \Sigma$. Assume by contradiction that $G \not\models \Sigma$. Then there exist a GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ in Σ and a match h of Q in G such that $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models l$. If so, then the literal $h(l)$ would have been added to enforced(Σ_G) during the computation of enforced(Σ_G), and been enforced on G during the population process above, i.e., $h(\bar{x}) \models l$ should hold in G, a contradiction.

($\Leftarrow$) Conversely, assume that Σ is weakly satisfiable, and let G_1 be a non-empty graph such that $G_1 \models \Sigma$. We prove by contradiction that enforced(Σ_G) is not conflicting. More specifically, we show that if enforced(Σ_G) is conflicting, then there exist two constants c and d such that both $u.A = c$ and $u.A = d$ hold for some node u in G_1, i.e., G_1 is not well defined, a contradiction.

Suppose that enforced(Σ_G) is conflicting. We can deduce the contradiction as follows. (i) Since G consists of a node v labeled with a label τ that does not appear in Σ, for any GFD $Q[\bar{x}](X \rightarrow Y)$ in Σ_G, we have that Q consists of a single node x labeled "_," since otherwise Q cannot have a match in G; (ii) then for any node u in G_1, all GFDs in Σ_G can also be applied on u, i.e., all patterns in Σ_G can be embedded in u; (iii) by induction on the computation steps of enforced(Σ_G), we can show that for each literal $v.A = c$ or $v.A = v.B$ in enforced(Σ_G), $u.A = c$ or $u.A = u.B$ also holds in G_1; and (iv) since there exist two constants c and d such that both $v.A = c$ and $v.A = d$ are in enforced(Σ_G), both $u.A = c$ and $u.A = d$ are in G_1, i.e., G_1 is not well defined, a contradiction.

Complexity. It remains to show that the characterization can be verified in PTIME. Note that since G consists of only one node, patterns in Σ may have only one match in G. Then, we can enumerate all matches of patterns of Σ in PTIME, compute enforced(Σ_G) in PTIME, and then verify the characterization in PTIME. Therefore, the weak satisfiability problem is in PTIME.

(2) Satisfiability and implication. We will give an algorithm for the satisfiability analysis of GFDs in the proof of Proposition 4.3, which is in $O(|\Sigma|^2 \cdot k^k)$ time. Similarly, we will develop an algorithm for checking GFD implication, in $O((|\Sigma| + |\varphi|) \cdot k^k)$ time. Thus, by the definition of fixed-parameter tractability, these problems are in FPT with parameter k.

(3) Validation. We first provide an FPT algorithm for GFDs defined with connected patterns. We then give a co-W[1]-hardness proof for GFDs defined with general (possibly disconnected) patterns.

Connected patterns. We will give an algorithm for the validation problem for GFDs with connected patterns in the proof of Proposition 4.3, which is in $O(d^{k^2}|\Sigma||G|)$ time. Thus, by the definition of fixed-parameter tractability, this problem is in FPT with parameters k and d for such GFDs.

General patterns. We show that the validation problem for GFDs with general patterns is co-W[1]-hard, by reduction from the complement of k-clique problem, which is known to be W[1]-complete [17]. The k-clique problem is to decide, given an undirected graph $G = (V, E)$ and a natural number k, whether there exists a clique of size k in G.

Given an undirected graph $G = (V, E)$ and a natural number k, we construct a natural number k', a graph G_1, and a set Σ of GFDs such that Σ is k'-bounded (i.e., each pattern in Σ has at most k' nodes; the notation will be explained shortly), and G has a clique of size k iff $G_1 \not\models \Sigma$. More specifically, we define $k' = k \times k + 1$. We construct graph G_1 and the set Σ of GFDs as follows.

(a) Graph G_1 is to encode the structure of G using several connected components. More specifically, G_1 is constructed in the following two steps. (i) It first includes a single node x labeled with a distinct label a_0 that does not appear in V. We set $x.A = 0$. Intuitively, we use attribute A to store the id of each node, such that nodes v with the same $v.A$ denote the same node in G. (ii) Then for each edge (u, v) in G, G_1 contains a connected component, which is constructed as follows: construct two nodes $x^u_{(u,v)}$ and $x^v_{(u,v)}$ labeled with u and v, respectively; set $x^u_{(u,v)}.A = u$ and $x^v_{(u,v)}.A = v$; and add two edges $(x^u_{(u,v)}, x^v_{(u,v)})$ and $(x^v_{(u,v)}, x^u_{(u,v)})$ labeled 0. Intuitively, we use two directed edges to represent one undirected edge in G. Two directed edges are identified as (u, v) by the A-attribute values of their endpoints. These encode the structure of G.

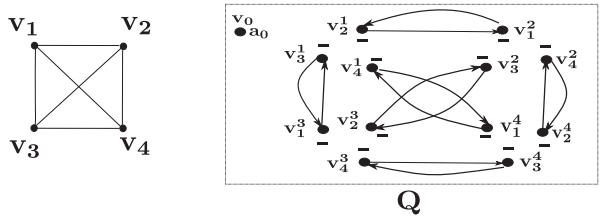


Fig. 3. The pattern of the GFD φ.

Note that the maximum degree of nodes in G_1 is 2. Hence, the hardness of validation is quite robust, since even when we restrict the maximum degree of graphs, the problem remains to be W[1]-hard. This motivates the use of connected patterns for the FPT results above.

(b) The set Σ consists of a single GFD $\varphi = Q[\bar{x}](X \to x_0.A = 1)$, to encode a k-clique. The pattern Q consists of $2 \cdot \binom{k}{k-1}$ isolated edges. To recover the clique, nodes with the same A-attribute value denote the same node in the clique. Moreover, Q has an isolated node v_0 to ensure that if there exists a k-clique in G, then Σ is not satisfiable. More specifically,

- $\bar{x}$ consists of variables $x_0, x_2^1, \ldots, x_k^1, x_1^2, x_3^2, \ldots, x_{k-1}^k$; intuitively, x_0 denotes the node in G_1 where we will deduce conflict. We use a group $x_1^i, \ldots, x_k^i$ of nodes (excluding x_i^i) to denote a node v_i in the k-clique, for $i \in [1, k]$; note that since the maximum degree of G_1 is 2, to make sure that the GFD φ can be applied to G_1, we also make the maximum degree of Q to be 2, and use a group of nodes to denote a node in the k-clique; and
- the literals in X are $(X_1 \wedge \ldots \wedge X_k)$, where X_1 is $(x_2^1.A = x_3^1.A) \wedge \ldots \wedge (x_k^1.A = x_2^1.A), \ldots,$ and X_k is $(x_1^k.A = x_2^k.A) \wedge \ldots \wedge (x_{k-1}^k.A = x_1^k.A)$. These ensure that all the nodes in each group share the same value of attribute A, representing the same node in the clique.

The pattern $Q[\bar{x}]$ in the GFD φ is defined as (V, E, L, μ), where

- $V = \{v_0, v_2^1, \ldots, v_{k-1}^k\}$;
- $E = \{(v_{i_2}^{i_1}, v_{i_1}^{i_2}), (v_{i_1}^{i_2}, v_{i_2}^{i_1}) \mid (i_1, i_2 \in [1, k]) \wedge (i_1 \neq i_2)\}$;
- for each node $u \in V \setminus \{v_0\}$, label $L(u) =$"_," and $L(v_0) =$"a_0"; for each $e \in E$, $L(e) =$"0"; and
- $\mu(v_0) = x_0$ and $\mu(x_j^i) = v_j^i$.

Intuitively, pattern Q is to check whether there exists a k-clique in G. The clique is constructed by connecting nodes in different groups. For example, edge (v_1, v_k) in the clique is represented by two edges (x_k^1, x_1^k) and (x_1^k, x_k^1). This GFD ensures that if there exists a k-clique in G, then for any node v labeled "a_0," $v.A = 1$. In contrast, by the construction of G_1, we know that for such node v, $v.A$ must be 0, which is a contradiction. Hence, G cannot contain a k-clique.

An encoding of a 4-clique by pattern Q is depicted in Figure 3. For instance, we use three nodes v_2^1, v_3^1 and v_4^1 to encode node v_1 in the 4-clique. Intuitively, these nodes represent edges (v_1, v_2), (v_1, v_3) and (v_1, v_4), respectively. In addition, we use two directed edges (v_1, v_2) and (v_2, v_1) to represent an undirected edge between nodes v_1 and v_2 in the 4-clique.

By the definition of Σ, we can see that Σ is k'-bounded. We next verify that G has a k-clique if and only if $G_1 \not\models \Sigma$.

($\Rightarrow$) Suppose that G has a k-clique. We show that $G_1 \not\models \Sigma$ by proving that $x.A = 1$ is enforced by Σ. Then by the definition of G_1, $x.A = 0$, a contradiction. Hence, $G_1 \not\models \Sigma$.

To see that $x.A = 1$ is enforced by Σ, suppose that the k-clique in G consists of nodes $v_1, \ldots, v_k$, and there exist nodes $x_{(v_1, v_2)}^{v_1}, \ldots, x_{(v_{k-1}, v_k)}^{v_k}$ in G_1 by the construction of G_1. Then, we can define a

match h of Q in G_1: $h(x_j^i) = x_{(v_i,v_j)}^{v_i}$ for $i, j \in [1,k]$ with $i \neq j$, and $h(x_0) = x$. To simplify the presentation, here we assume that $x_{(v_i,v_j)}^{v_i}$ and $x_{(v_j,v_i)}^{v_i}$ are the same variable, since (v_i, v_j) and (v_j, v_i) are the same edge in the undirected graph G. One can verify that h is a valid match of Q in G_1 such that $h(\bar{x}) \models X$, where X is in $\varphi = Q[\bar{x}](X \rightarrow x_0.A = 1)$. By the definition of φ, $x.A = 1$ is enforced by Σ.

($\Leftarrow$) Suppose that $G_1 \not\models \Sigma$. By the definition of Σ, this is possible only when the nodes labeled "a_0" have two distinct values for attribute A. Hence, there exists a match h of $\varphi = Q[\bar{x}](X \rightarrow x_0.A = 1)$ in G_1 such that $h(\bar{x}) \models X$. Based on these, we next show that there exists a k-clique in G.

The k-clique is defined as follows. Let $v_1 = h(v_2^1).A$, $v_2 = h(v_3^2).A, \ldots, v_{k-1} = h(v_k^{k-1}).A$, and $v_k = h(v_1^k).A$. It suffices to show that (i) $v_1, v_2, \ldots$, and v_k are the nodes in G, and (ii) $v_1, v_2, \ldots$, and v_k form a k-clique in G. For (i), observe that $h(v_2^1), h(v_3^2), \ldots, h(v_k^{k-1})$ and $h(v_1^k)$ are not the node labeled "a_0." Indeed, the node labeled "a_0" is an isolated node in G_1, while $h(v_2^1), h(v_3^2), \ldots, h(v_k^{k-1})$ and $h(v_1^k)$ are nodes having an adjacent edge. Hence, $v_1, v_2, \ldots$, and v_k are in G.

Next, we show that there exists an edge between any two of these nodes, i.e., for any $i, j \in [1,k]$ with $i \neq j$, there exist two nodes x_i and x_j in G_1 such that edge (x_i, x_j) is in G_1, and $x_i.A$ (respectively, $x_j.A$) is v_i (respectively, v_j). For if this holds, then these nodes make a k-clique. Given any $i, j \in [1,k]$ with $i \neq j$, by the definition of GFD φ given above, there exists an edge (v_j^i, v_i^j) in Q. Define $x_i = h(v_j^i)$ and $x_j = h(v_i^j)$. Since $h(\bar{x}) \models X$, we know that $x_i.A = h(v_j^i).A = h(v_{n_i}^i).A = v_i$ and $x_j.A = h(v_i^j).A = h(v_{n_j}^j).A = v_j$, where $n_i = ((i+1) \mod k)$ and $n_j = ((j+1) \mod k)$. Hence, there indeed exists an edge (x_i, x_j) in G_1, and therefore, there exists an edge (v_i, v_j) in G. □

Size-bounded GFDs. There are practical cases when the fundamental problems are tractable. Consider size-bounded GFDs defined as follows. For a constant k, we say that a pattern $Q[\bar{x}]$ is *k-bounded* if $|\bar{x}| \leq k$, i.e., the number of its vertices is at most k. We say that a set Σ of GFDs is *k-bounded* if for each GFD $Q[\bar{x}](X \rightarrow l)$ in Σ, $Q[\bar{x}]$ is k-bounded.

It often suffices to consider k-bounded GFDs in practice with a fairly small k [10, 30, 52]. Indeed, 66.41% of patterns in DBPedia and 97.25% in SWDF consist of a single triple, i.e., $k \leq 2$; 98% of real-life SPARQL queries have radius 1 and no more than four nodes and five edges [30]. This is also witnessed by GFDs expressing axioms in knowledge bases (Section 3.2). As pattern (query) verification is typically a crucial step in rule mining [29], the size of practical queries indicates a reasonable k.

Better still, the three intractable problems above become tractable for k-bounded GFDs. That is, when the parameter is small, these problems are within reach in practice.

Proposition 4.3. *When k is a predefined constant, the satisfiability, implication and validation problems are in* PTIME *for k-bounded* GFDs.

The proof makes use of characterizations of the satisfiability and implication problems given in Reference [27]. Before giving the proof, below we first review the characterizations of Reference [27].

Characterizing satisfiability. Assume that Σ is a given set of GFDs. For each GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ in Σ, we compute a set enforced(Σ_Q) of literals when we treat Q as a graph (see the proof of Theorem 4.2 for the definition of enforced(Σ_Q)). As shown in Reference [27], the condition below is a characterization of GFD satisfiability: *a set Σ of* GFDs *is satisfiable if and only if there exists a* GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ *in Σ such that* enforced(Σ_Q) *is not conflicting.*

Characterizing implication. Consider a set Σ of GFDs and another GFD $\varphi = Q[\bar{x}](X \to l)$. Extending enforced$(\Sigma_Q)$, we define a set closure(Σ_Q, X) as follows:

(a) all literals in X are in closure(Σ_Q, X); and
(b) for every GFD $Q_1[\bar{x}_1](X_1 \to l_1)$ in Σ_Q, l_1 is in closure(Σ_Q, X) if either $X_1 = \emptyset$, or $X_1 \neq \emptyset$ and all literals in X_1 can be deduced from closure(Σ_Q, X) via the transitivity of equality literals.

Similar to conflicting enforced(Σ_Q) given above, we define *conflicting* closure(Σ_Q, X).

Note that (1) given a set Σ of GFDs and a GFD φ, the computation of closure(Σ_Q, X) may take exponential time; and (2) when all GFDs are k-bounded, the construction of Σ_Q is in PTIME, and so is the computation of closure(Σ_Q, X).

The implication of GFDs is characterized as follows [27]. *For any set* Σ *of* GFDs *and* GFD $\varphi = Q[\bar{x}](X \to l)$, $\Sigma \models \varphi$ *if and only if either* closure(Σ_Q, X) *is conflicting or* l *is in* closure(Σ_Q, X).

Intuitively, it states that either (a) Σ and (Q, X) are inconsistent and hence, l is a "logical consequence" of Σ and (Q, X); or (b) closure(Σ_Q, X) is consistent and l is enforced by Σ and (Q, X).

Based on the characterizations, we now give a proof of Proposition 4.3.

Proof. For constant k, we give PTIME algorithms for the three problems one by one.

(a) Satisfiability. We check the satisfiability of Σ as follows:

(1) compute the set Σ_Q of GFDs embedded in Q for each GFD $Q[\bar{x}](X \to l)$ in Σ, and compute the associated equivalence relation enforced(Σ_Q);
(2) if enforced(Σ_Q) is conflicting for all GFDs $Q[\bar{x}](X \to l)$ in Σ, then return false; otherwise, return true.

The correctness of the algorithm follows from the characterization of the satisfiability problem in Reference [27] as described above. For the complexity, observe the following. (I) We can compute Σ_Q as follows. For each GFD $\varphi' = Q'[\bar{x}](X' \to l') \in \Sigma$, enumerate all isomorphic mappings from Q' to subgraphs of Q; for each mapping f, add $Q'[\bar{x}](f(X') \to f(l'))$ to Σ_Q. Here an *isomorphic mapping* f from Q' into Q is a match of Q' in Q when we treat Q as a graph. Since pattern Q' is k-bounded (i.e., Q' has at most k nodes), there exist at most k^k candidate isomorphic mappings of Q' to subgraphs of Q. Hence, this can be done in $O(|\Sigma| \times k^k)$ time. Since we do this for all GFDs in Σ, step (1) in is $O(|\Sigma|^2 \times k^k)$ time. In the same process, enforced(Σ_Q) can be computed, by deducing the equality of attributes via the transitivity, similar to the computation of the closure of relational FDs [3]. (II) One can verify that $|\text{enforced}(\Sigma_Q)| \leq |\Sigma| \times k$ when represented as an equivalence relation, and that checking whether enforced(Σ_Q) is conflicting can be done in linear time. Thus, step (2) takes $O(|\Sigma|^2 \times k)$ time. Hence, the algorithm is in $O(|\Sigma|^2 \times k^k)$ time, which shows that the satisfiability problem is in PTIME when k is a constant.

(b) Implication. Given a GFD $\varphi = Q[\bar{x}](X \to l)$ and a set Σ of GFDs, we check $\Sigma \models \varphi$ as follows:

(1) compute the set Σ_Q of GFDs embedded in Q, and closure(Σ_Q, X) accordingly;
(2) if closure(Σ_Q, X) is conflicting, or if l can be deduced from closure(Σ_Q, X) via equality transitivity, then return true; otherwise, return false.

The correctness follows from the characterization of GFD implication of Reference [27] reviewed above. For the complexity, (I) step (1) computes Σ_Q in the same way as step (1) in satisfiability checking, except that we handle Q instead of patterns in Σ. It takes $O((|\varphi|+|\Sigma|)\times k^k)$ time. (II) Step (2) takes $O((|\varphi|+|\Sigma|)\times k)$ time, along the same lines as computing the closures of

relational FDs. So the algorithm is in $O((|\varphi|+|\Sigma|)\times k^k)$ time, and the implication problem is in PTIME when k is a constant.

(c) Validation. Given a set Σ of GFDs and a graph G, we check whether $G \models \Sigma$ as follows:

(1) for all GFDs $\varphi = Q[\bar{x}](X \rightarrow l)$ in Σ and all matches h of Q in G, check whether $h(\bar{x}) \models (X \rightarrow l)$;
(2) if so, return true; otherwise, return false.

The correctness follows from the definition of satisfaction of GFDs. For the complexity, we show that: (†) for any connected pattern Q', there exist at most $d^{k^2}|G|$ matches of Q' in G, where d is the maximum degree of the nodes in G. If it holds, given any GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ in Σ, since Q has at most k connected components, then we know that Q has at most $(d^{k^2}|G|)^k = d^{k^3}|G|^k$ matches in G. Hence, the algorithm is in $O(|\Sigma|d^{k^3}|G|^k) = O(|\Sigma||G|^{k^3+k})$ time, which is in PTIME when k is a constant. Note that when all patterns in Σ are connected, the algorithm is in $O(|\Sigma|d^{k^2}|G|) = O(|\Sigma||G|^{k^2+1})$ time.

It remains to show statement (†) above. Observe that (I) for any node x_0 in Q', it can be mapped to at most $|G|$ many vertices in G; (II) for each such match v_0 of x_0, the matches of Q' relative to v_0 can involve at most d^k many vertices, since we adopt the subgraph isomorphism semantics and Q' is a connected pattern; (III) there exist at most d^{k^2} many matches of Q' relative to v_0 for all these d^k many vertices, and then there exist at most $d^{k^2}|G|$ many matches of Q' in the graph G. □

5 THE DISCOVERY PROBLEM

We next formalize the discovery problem for GFDs. Given a graph G, GFD discovery aims to find a set Σ of GFDs such that $G \models \Sigma$. Clearly it is not desirable to return all such GFDs, since Σ contains unnecessarily large amount of trivial and redundant GFDs. Instead, we prefer (1) GFDs that are not redundant and nontrivial, and (2) *frequent* GFDs that capture regularities and constraints.

To simplify the discussion, we focus on discovery of GFDs with connected patterns in this article. The handling of GFDs with disconnected patterns is more costly and is deferred to future research.

5.1 Reduced GFDs and GFD Cover

We first formulate nontrivial and reduced GFDs, following the practice of mining relational dependencies (e.g., conditional functional dependencies on relations [13, 22]).

Nontrivial GFDs. A GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ is *trivial* if either (a) X is conflicting, such that φ is satisfied by any graph; or (b) l can be derived from X via the transitivity of equality, i.e., $X \rightarrow l$ trivially holds. Obviously, we are only interested in discovering nontrivial GFDs.

Reduced GFDs. We formulate reduced GFDs with an ordering on graph patterns and GFDs.

(1) Given two graph patterns $Q[\bar{x}](V_Q, E_Q, L_Q, \mu)$ and $Q'[\bar{x}'](V'_Q, E'_Q, L'_Q, \mu')$, we say that Q *reduces* Q', denoted by $Q[\bar{x}] \ll Q'[\bar{x}']$, if Q either removes nodes or edges from Q', or changes some labels in Q' to wildcard. That is, Q is a topological constraint less restrictive than Q', and can be mapped to more subgraphs. For instance, $Q_2 \ll G_2$ if we treat graph G_2 given in Figure 1 as a pattern. Note that since Q can be obtained by removing nodes or edges from Q', or changing some labels in Q' to wildcard, there exists an isomorphic mapping f from pattern Q into a subgraph of pattern Q'.

(2) In a graph pattern $Q[\bar{x}]$, we designate a variable $z \in \bar{x}$ and refer to it as *the pivot of* Q. Intuitively, we use pivot to explore the data locality of subgraph isomorphism: for any v in graph G, if there exists a match h of Q in G such that $h(z) = v$, then $h(\bar{x})$ consists of only those nodes in the d_Q-neighbor of v. Here d_Q is the *radius* of Q at z, i.e., the longest shortest path from z to any node in Q. The d_Q-neighbor of v includes all nodes and edges within d_Q hops of v.

In practice, a pivot indicates user-specified interest. Ideally, we pick a pivot that is selective, e.g., it bears an "uncommon" label. For instance, for patterns Q_1-Q_3 of Example 1.1, we may pick x as their pivot and rewrite the corresponding GFDs as $Q_1[\underline{x}, y](y.\text{type} = \text{"film"} \rightarrow x.\text{type} = \text{"producer"})$, $Q_2[\underline{x}, y, z](\emptyset \rightarrow y.\text{val} = z.\text{val})$ and $Q_3[\underline{x}, y](\emptyset \rightarrow \text{false})$, respectively.

(3) Consider positive GFDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \rightarrow l_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \rightarrow l_2)$. We say that φ_1 *reduces* φ_2, denoted by $\varphi_1 \ll \varphi_2$, if there exists an isomorphic mapping f from pattern Q_1 into a subgraph of pattern Q_2 such that (a) $f(z_1) = z_2$, where z_i denotes the pivot of Q_i ($i \in [1, 2]$), i.e., f preserves pivots; (b) f maps variables in X_1 and l_1 to those in X_2 and l_2, respectively, such that $f(X_1) \subseteq X_2$ and $f(l_1) = l_2$, where $f(X)$ substitutes $f(x)$ for each variable x (i.e., node $\mu(x)$; recall that we use $\mu(x)$ and x interchangeably) in X; and (c) either $Q_1 \ll Q_2$ via mapping f or $f(X_1) \subsetneq X_2$. Intuitively, Q_1 reduces Q_2 and X_1 reduces X_2.

Example 5.1. Recall GFD $\varphi_1 = Q_1[x, y](X_1 \rightarrow l)$ from Example 1.1, where $X_1 = \{y.\text{type} = \text{"film"}\}$, and l is $x.\text{type} = \text{"producer"}$. Let x be the pivot. (1) Consider GFD $\varphi_1^1 = Q_1^1[x, y, z](X^1 \rightarrow l)$ with pivot x, where (a) $Q_1^1[x, y, z]$ is obtained by adding an edge (y, z) to Q_1, z is labeled award, and (b) X_1^1 is $X_1 \cup \{y.\text{name} = \text{"Selling out"}\}$. Then $\varphi_1 \ll \varphi_1^1$. (2) Consider GFD $\varphi_1^2 = Q_1^1[x, y, z](X_1^2 \rightarrow l)$, where $X_1^2 = \{y.\text{name} = \text{"Selling out"}\}$. Then $\varphi_1 \not\ll \varphi_1^2$, since $X_1 \not\subseteq X_1^2$.

Based on GFD ordering, we define reduced GFDs.

Reduced positive GFDs. We say that a positive GFD φ is *reduced* in graph G if $G \models \varphi$ but $G \not\models \varphi'$ for any GFD $\varphi' \ll \varphi$. It is *minimum in* G if it is both nontrivial and reduced. Intuitively, when φ is reduced in G, for any GFD $\varphi' = Q[\bar{x}](X \rightarrow Y)$ such that $\varphi' \ll \varphi$, there exists a match $h(\bar{x})$ of φ' in G such that $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$, i.e., $G \not\models \varphi'$. Conversely, since φ is more restrictive than φ', and $h(\bar{x})$ may not be a match of φ, it is possible that $G \models \varphi$ but $G \not\models \varphi'$.

A reduced positive φ guarantees the following: (a) its attribute dependency is *left-reduced* [13, 22, 35], i.e., $G \not\models Q[\bar{x}](X' \rightarrow l)$ for any proper subset $X' \subsetneq X$, and hence X does not include redundant literals; and (b) its pattern is *pattern-reduced*, i.e., $G \not\models Q'[\bar{x}'](X \rightarrow l)$ for any $Q'[\bar{x}']$ with $Q'[\bar{x}'] \ll Q[\bar{x}]$, where $X \rightarrow l$ is defined on $Q'[\bar{x}']$ (with variable renaming).

Reduced negative GFDs. We say that a negative GFD φ is *minimum* if it is extended from a positive minimum GFD $\psi = Q[\bar{x}](X \rightarrow l)$ by either (a) adding an edge to pattern Q and obtaining GFD $\varphi = Q'[\bar{x}](\emptyset \rightarrow \text{false})$, or (b) adding a literal to X and getting $\varphi = Q[\bar{x}](X' \rightarrow \text{false})$. That is, it is triggered by minimum change to ψ on either pattern Q or literal set X.

Cover of GFDs. Consider a set Σ of GFDs such that $G \models \Sigma$. We say that Σ is *minimal* if for all GFDs $\varphi \in \Sigma$, $\Sigma \not\equiv \Sigma \setminus \{\varphi\}$, i.e., Σ includes no redundant GFDs.

A *cover* Σ_c of Σ on graph G is a subset of Σ such that

(a) $G \models \Sigma_c$,
(b) $\Sigma_c \equiv \Sigma$,
(c) all GFDs in Σ_c are minimum, and
(d) Σ_c is minimal itself.

That is, Σ_c contains no redundant or non-interesting GFDs (see Reference [3] for more about covers).

5.2 Frequent GFDs

We want to find "frequent" GFDs φ on a graph G, indicating how often φ can be applied and thus whether φ captures regularity and is "interesting." This is typically measured in terms of support.

The notion of support is, however, nontrivial to define for GFDs. To see this, consider a GFD $\varphi = Q[\bar{x}](X \to l)$. Following the conventional notion, the support of φ would be defined as the number of matches of Q in G that satisfy $X \to l$. However, as observed in References [18, 38], this definition is *not* anti-monotonic. For example, consider pattern $Q[x]$ with a single node labeled person x, and $Q'[x, y]$ with a single edge from person x to person y labeled hasChild. In real-life graphs G, we often find that the support of Q' is larger than that of Q although Q is a sub-pattern of Q', since a person may have multiple children; similarly for GFDs defined with Q and Q'.

We next propose a notion of support for GFDs, defined in terms of the support of their associated pattern and the correlation of its attributes, as follows.

Pattern support. Consider a graph G, and a positive GFD φ with pattern $Q[\bar{x}]$, where Q has pivot z. Denote by $Q(G, z)$ the set of nodes $h(z)$ for all matches h of Q in G.

We define the *support* of graph pattern Q as follows:

$$\mathsf{supp}(Q, G) = |Q(G, z)|.$$

Intuitively, this notion quantifies the frequency of the entities in G that satisfy the topological constraint that is posed by pattern Q and is "pivoted" at z.

It is to ensure the anti-monotonicity of support of GFDs that we employ pivots in the definition above. As verified by the practice of conventional data mining, the anti-monotonicity is crucial to scaling with large datasets and speeding up the discovery process. To simplify the discussion, we do not include pivots in $\mathsf{supp}(Q, G)$ when it is clear from the context.

Correlation. To quantify attribute dependencies in $Q[\bar{x}]$, we define the *correlation* of GFD φ as

$$\rho(\varphi, G) = \frac{|Q(G, Xl, z)|}{|Q(G, z)|}.$$

Here $Q(G, Xl, z)$ denotes the subset of the set $Q(G, z)$ of nodes with mapping h such that both $h(\bar{x}) \models X$ and $h(\bar{x}) \models l$ (recall that GFD $\varphi = Q[\bar{x}](X \to l)$).

Intuitively, $\rho(\varphi, G)$ characterizes the dependency of literal l on X in terms of "true" implication of l from X, i.e., l holds when X holds, excluding the cases when either X is not satisfied by $h(\bar{x})$, or both X and l are not satisfied by $h(\bar{x})$. One can verify that if we include these two cases, then $Q(G, X \to l, z) = Q(G, z)$ as long as $G \models \varphi$, where $Q(G, X \to l, z)$ is the subset of $Q(G, z)$ of nodes with mappings h such that $h(\bar{x}) \models X \to l$. That is, it does not accurately measure the correlation of X and l. Note that $Q(G, Xl, z)$ and $Q(G, X \to l, z)$ are different, and $Q(G, Xl, z)$ can be a proper subset of $Q(G, X \to l, z)$. Compared with $Q(G, Xl, z)$, the set $Q(G, X \to l, z)$ also includes nodes in $Q(G, z)$ with mappings h such that $h(\bar{x}) \not\models X$, i.e., $h(\bar{x}) \models X \to l$ holds.

Support of positive GFD φ. The *support of φ in G* is defined as

$$\mathsf{supp}(\varphi, G) = \mathsf{supp}(Q, G) * \rho(\varphi, G).$$

Given the definitions of $\mathsf{supp}(Q, G)$ and $\rho(\varphi, G)$ defined above, we can further simplify the computation of $\mathsf{supp}(\varphi, G)$ as follows: $\mathsf{supp}(\varphi, G) = \mathsf{supp}(Q, G) * \rho(\varphi, G) = |Q(G, Xl, z)|$.

Anti-monotonicity. We next justify that the support is well defined in terms of anti-monotonicity and GFD ordering. The result below shows that the support of GFDs is anti-monotonic. The proof will be given shortly, after we extend the notion of support to negative GFDs.

Theorem 5.2. *For any graph G and any nontrivial* GFDs φ_1 *and* φ_2, *if* $\varphi_1 \ll \varphi_2$ *then* $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$.

For instance, for φ_1 and φ_1^1 of Example 5.1, we have that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_1^1, G)$, since $\varphi_1 \ll \varphi_1^1$. Indeed, every producer x induced from the match $Q_1^1(G, X_1^1 l, x)$ is a producer in $Q_1(G, X_1 l, x)$.

Support of negative GFDs. As remarked in Section 3, a negative GFD $\varphi = Q[\bar{x}](X \to \text{false})$ has two forms: (a) $X = \emptyset$, i.e., Q is an "illegal" pattern and hence $Q(G, z) = \emptyset$ at any pivot z in a "consistent" graph G; and (b) $X \neq \emptyset$ and it is satisfiable, i.e., the combination of Q and X is "illegal". Putting φ in the normal form $Q[\bar{x}](X \to l)$ by taking false as a literal l, in both cases, $Q(G, Xl, z) = \emptyset$. Thus, we cannot discover negative GFDs by computing $Q(G, Xl, z)$ and $Q(G, z)$.

As remarked earlier, we are interested in negative GFD φ that is obtained from a positive $Q'[\bar{x}'](X' \to l')$ with minimal extensions: either "vertical extension" of Q' with a single edge (possible with new nodes), or "horizontal extension" of X' with a single literal. Thus, we define

$$\mathsf{supp}(\varphi, G) = \mathsf{max}_{\varphi' \in \Phi'}(\mathsf{supp}(\varphi', G)),$$

where (1) if $X = \emptyset$, the set Φ' consists of patterns $Q'[\bar{x}']$ with the same pivot z such that $\mathsf{supp}(Q', G) > 0$, and Q' is obtained from Q by removing one edge along with its nodes of degree 0; and (2) if $X \neq \emptyset$, Φ' is the set of positive GFDs $\varphi' = Q[\bar{x}](X' \to l')$ with the same pivot z such that $G \models \varphi'$, and there exists a literal l'' in X with $X = X' \uplus \{l''\}$, where $\uplus$ is the disjoint union operator. In case (1) (respectively, case (2)), we refer to pattern $Q' \in \Phi'$ (respectively, positive GFD $\varphi' \in \Phi'$) with the maximum support in Φ' as a *base* of φ.

That is, $\mathsf{supp}(\varphi, G)$ of negative GFD φ is decided by its base pattern Q' or base positive GFD φ'. If Q' or φ' has a sufficiently large support, then φ suggests a meaningful negative GFD.

Given graph G, a GFD φ and a support threshold σ, φ is *frequent in* G w.r.t.σ if $\mathsf{supp}(\varphi, G) \geq \sigma$.

Proof of Theorem 5.2. We next prove Theorem 5.2. Theorem 5.2 holds on generic GFDs, *positive or negative*. This is the first anti-monotonicity result for mining functional dependencies with graph patterns. It ensures the feasibility of GFD discovery (see Section 7 for details).

Consider two nontrivial GFDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \to l_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to l_2)$ such that $\varphi_1 \ll \varphi_2$. Then by the definition of $\varphi_1 \ll \varphi_2$, there exists a bijective mapping h from Q_1 to a subgraph of Q_2 such that (a) $Q_1 \ll Q_2$ via h, (b) $h(z_1) = z_2$, where z_i denotes the pivot of Q_i for $i \in [1, 2]$, and (c) $h(X_1) \subseteq X_2$ and $h(l_1) = l_2$. We distinguish the following cases.

(1) Both φ_1 and φ_2 are positive. Denote by $Q_1(G, X_1 l_1, z_1)$ (respectively, $Q_2(G, X_2 l_2, z_2)$) the set of nodes that match pivot z_1 (respectively, z_2) induced by $h(\bar{z})$ for all matches h of pattern Q_1 (respectively, Q_2) in G such that $h(\bar{x}) \models X_1$ and $h(\bar{x}) \models l_1$ (respectively, $h(\bar{x}) \models X_2$ and $h(\bar{x}) \models l_2$). By the definition of support of positive GFDs, it suffices to show that $Q_2(G, X_2 l_2, z_2) \subseteq Q_1(G, X_1 l_1, z_1)$. That is, for any node v in G, if there exists a match h_2 of Q_2 in G such that $h_2(z_2) = v$, $h_2(\bar{x}_2) \models X_2$ and $h_2(\bar{x}_2) \models l_2$, then there must exist a match h_1 of Q_1 in G such that $h_1(z_1) = v$, $h_1(\bar{x}_1) \models X_1$ and $h_1(\bar{x}_1) \models l_1$.

Given h_2, we define mapping $h_1 = h_2 \circ h$, i.e., the composition of h_2 and h. Then, we can show that $h_1(z_1) = v$. Moreover, $h_1(\bar{x}_1) \models X_1$ and $h_1(\bar{x}_1) \models l_1$. Indeed, since $h_2(\bar{x}_2) \models X_2$ and $h(X_1) \subseteq X_2$, we have that $h_2(\bar{x}_2) \models h(X_1)$. Therefore, $h_2 \circ h(\bar{x}_1) \models X_1$. That is, $h_1(\bar{x}_1) \models X_1$; similarly, we show that $h_1(\bar{x}_1) \models l_1$. Hence, $v \in Q_1(G, X_1 l_1, z_1)$, and thus we can conclude that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$.

(2) Only one of φ_1 and φ_2 is positive. This case cannot happen. Indeed, if GFD φ_1 is positive, then by $h(l_1) = l_2$, literal l_1 must be false and hence φ_1 is negative, a contradiction. Similarly, we can prove that GFD φ_2 cannot be positive while φ_1 is negative.

(3) Both φ_1 and φ_2 are negative. Recall that if $\varphi = Q(\emptyset \rightarrow \text{false})$, then the negative GFD φ is obtained from a positive GFD by adding an edge (possibly with new nodes). If $\varphi = Q(X \rightarrow \text{false})$ and $X \neq \emptyset$, then φ is obtained by adding a literal. Indeed, if both $X \neq \emptyset$ and $\text{supp}(Q, G) = 0$, then any GFD defined with Q is already negative. Consider the three cases below.

(a) When $\varphi_1 = Q_1(\emptyset \rightarrow \text{false})$ and $\varphi_2 = Q_2(\emptyset \rightarrow \text{false})$. Since $\varphi_1 \ll \varphi_2$, assume *w.l.o.g.* that Q_i' with pivot z_i is the base of φ_i for $i \in [1, 2]$ such that $Q_1' \ll Q_2'$. Suppose that $\text{supp}(\varphi_2, G) > 0$. To show that $\text{supp}(\varphi_1, G) \geq \text{supp}(\varphi_2, G)$, we prove that $\text{supp}(Q_1', G) \geq \text{supp}(Q_2', G)$. This suffices, since if it holds, then by the definitions of $\text{supp}(\varphi_1, G)$ and $\text{supp}(\varphi_2, G)$, we can conclude that $\text{supp}(\varphi_1, G) = \text{supp}(Q_1', G) \geq \text{supp}(Q_2', G) = \text{supp}(\varphi_2, G)$.

Denote by $Q_1(G, z_1)$ (respectively, $Q_2(G, z_2)$) the set of nodes that match pivot z_1 (respectively, z_2) of Q_1 (respectively, Q_2) in G. By the definition of pattern support, it suffices to show that $Q_2'(G, z_2) \subseteq Q_1'(G, z_1)$. That is, for any node v in G, if there exists a match h_2 of Q_2' in G such that $h_2(z_2) = v$, then there must exist a match h_1 of Q_1' in G such that $h_1(z_1) = v$. Given h_2, we define h_1 as follows. Since $Q_1' \ll Q_2'$, there exists a bijective mapping h from Q_1' to a subgraph of Q_2' such that $h(z_1) = z_2$. Define $h_1 = h_2 \circ h$. Then, $h_1(z_1) = v$. Hence, $v \in Q_1'(G, z_1)$, and thus $\text{supp}(\varphi_1, G) \geq \text{supp}(\varphi_2, G)$.

(b) When one of φ_1 and φ_2 has the form $Q[\bar{x}](\emptyset \rightarrow \text{false})$ while the other is $Q'[\bar{x}'](X \rightarrow \text{false})$ with $X \neq \emptyset$. The two are incomparable. Intuitively, the two have different types of bases: one measures the maximum "legal" pattern, and the other assesses the maximum "consistent" literals for entities identified by Q'. Neither can reduce the other. As an indicator, $\text{supp}(Q, G) = 0$ and $\text{supp}(Q', G) \neq 0$.

(c) When $\varphi_1 = Q_1(X_1 \rightarrow \text{false})$ and $\varphi_2 = Q_2(X_2 \rightarrow \text{false})$, where $X_1 \neq \emptyset$ and $X_2 \neq \emptyset$. We show that $\text{supp}(\varphi_1, G) \geq \text{supp}(\varphi_2, G)$. Since $\varphi_1 \ll \varphi_2$, suppose that φ_i' is the base of φ_i for $i \in [1, 2]$ such that $\varphi_1' \ll \varphi_2'$. Because φ_1' and φ_2' are positive GFDs, from the proof of case (1) it follows that $\text{supp}(\varphi_1', G) \geq \text{supp}(\varphi_2', G)$. By the definitions of $\text{supp}(\varphi_1, G)$ and $\text{supp}(\varphi_2, G)$, we have that $\text{supp}(\varphi_1, G) = \text{supp}(\varphi_1', G)$ and $\text{supp}(\varphi_2, G) = \text{supp}(\varphi_2', G)$. Hence, $\text{supp}(\varphi_1, G) \geq \text{supp}(\varphi_2, G)$.

This completes the proof of Theorem 5.2. □

Open World Assumption (OWA). Recall that the OWA states that absent data cannot be used as counterexamples in knowledge bases [26, 29]. We remark that the support of GFDs defined above is consistent with the OWA: (1) for a positive GFD φ, its support quantifies the entities that exist and conform to φ; and (2) for negative φ, its support is determined by the support of positive GFD φ' justified in (1); that is, negative GFDs characterize "non-existence" cases in the observed world; the unknown data does not have impact on the discovery of negative GFDs.

5.3 The Discovery Problem

We are now ready to state the discovery problem for GFDs.

- Input: A graph G, a natural number $k \geq 2$, and a support threshold $\sigma > 0$.
- Output: A cover Σ_c of all k-bounded minimum GFDs φ that are σ-frequent, i.e., $\text{supp}(\varphi, G) \geq \sigma$.

Observe that the validation and implication problems are embedded in GFD discovery, for checking $G \models \varphi$ and computing a cover Σ_c of k-bounded GFDs φ discovered, respectively.

We take the number k of nodes in graph patterns as a parameter to balance the complexity of discovery and the interpretablility of GFDs. Indeed, (a) GFDs with too large patterns are less likely to be frequent, and are hard to interpret for end users (Section 4), and (b) by Proposition 4.3, the implication and validation problems for GFDs are in PTIME when k is fixed. While the

mining takes "pay-as-you-go" cost with larger k, we find that k-bounded GFDs suffice to cover meaningful rules to detect errors with high accuracy when k is fairly small (see Section 9).

Remarks. (1) To reduce excessive literals, we often select a set Γ of *active attributes* from G that are of users' interest or are attributes with high confidence to be "clean." We only discover GFDs with literals composed of attributes in Γ. (2) The selection of σ is domain-specific. We set σ empirically based on the support of active attributes and pattern support, to make sure the discovered GFDs have sufficient pattern support. One can adopt a small σ to find non-frequent GFDs, just like References [12, 29]. (3) Our techniques apply to the discovery of general GFDs without these restrictions.

6 SEQUENTIAL GFD DISCOVERY

In the next three sections, we develop algorithms for discovering GFDs.

We start with a sequential discovery algorithm in this section, denoted as SeqDisGFD. It consists of two algorithms: (1) SeqDis that, given G, k and σ, discovers the set Σ of all k-bounded minimum σ-frequent GFDs, and (2) SeqCover that, given Σ, computes a cover Σ_c of Σ.

Below, we first present algorithm SeqDis and show how it discovers positive GFDs (Section 6.1). We then extend SeqDis to discover negative GFDs (Section 6.2). We present algorithm SeqCover in Section 6.3. Finally, we give an analysis of these algorithms in Section 6.4.

6.1 Sequential GFD Mining

A brute-force algorithm first enumerates all frequent patterns Q in G following conventional graph pattern mining (e.g., References [18, 55]), and then generates GFDs with Q by adding literals. However, enumerating all k-bounded GFDs is costly when G is large. To reduce the cost, algorithm SeqDis integrates the two processes into one, to eliminate non-interesting GFDs as early as possible.

Overview. Given as Algorithm 1, algorithm SeqDis runs in k^2 iterations. At each iteration i, it discovers and stores all the minimum σ-frequent GFDs of size i (with i edges) in a set Σ_i, by using a GFD *generation tree*. In the first iteration, it "cold-starts" GFD discovery by initializing the GFD generation tree T with frequent GFDs that carry a single-node pattern (lines 1 and 2). The tree T is then expanded by interleaving two levelwise spawning processes: *vertical spawning* VSpawn to extend graph patterns Q (line 4) , and *horizontal spawning* HSpawn to generate dependencies $X \rightarrow l$ (line 10). At each iteration i ($1 \leq i \leq k^2$), SeqDis generates and verifies candidate GFDs (lines 7 and 13), and fills the level-i part of tree T (line 14). It interleaves the following two steps.

(1) Pattern verification. Algorithm SeqDis first performs *vertical spawning* VSpawn, which generates new graph patterns at level i of tree T (to be discussed shortly). Each pattern Q' expands a level $i-1$ pattern Q by adding a new edge e (possibly with new nodes). It then performs pattern matching to find matches for all the patterns at level i. Vertical spawning grows tree T levelwise vertically.

(2) GFD Validation. Algorithm SeqDis then performs *horizontal spawning* HSpawn, which associates a set of dependencies $X \rightarrow l$ with each newly verified pattern Q at level i of tree T to generate a set of candidate GFDs $Q[\bar{x}](X \rightarrow l)$. For each batch of candidate GFDs, it performs GFD *validation* to find GFDs in Σ_i, i.e., those candidates at level i such that they are satisfied by G, and are frequent and minimum. The validation process terminates when all such candidate GFDs pertaining to the graph patterns at level i are validated. This step grows tree T levelwise horizontally.

These two steps interleave and iterate until either no new GFDs can be further spawned, or all the k-bounded GFDs on graph G have been checked (i.e., when $i = k^2$).

ALGORITHM 1: Algorithm SeqDis

Input: graph G, integer k, support threshold σ.
Output: a set Σ of all k-bounded minimum σ-frequent GFDs φ.

```
1  set Σ := ∅; GFD tree T := ∅; integer i := 1; flag_V := true;
2  SpawnGFD(T); /* initialize generation tree T with single-node GFDs */ ;
3  while i ≤ k^2 and flag_V /* superstep i */ do
4      VSpawn(i); Σ_i := ∅;
5      flag_V := false if no new pattern is spawned;
6      if flag_V then
7          pattern verification;
8          j := 1; flag_H := true;
9          while j ≤ J and flag_H do
10             set Σ_{C_ij} := HSpawn(i, j);
11             flag_H := false if no new GFD candidates are spawned;
12             if flag_H then
13                 GFD validation for Σ_{C_ij};
14                 Σ := Σ ∪ Σ_{C_ij}; j := j + 1;
15             end
16         end
17     end
18     Σ := Σ ∪ Σ_i i := i + 1;
19 end
20 return Σ;
```

We next present the details of *vertical spawning* and *horizontal spawning*. Underlying the processes is the maintenance of a GFD generation tree, which stores candidate GFDs.

Generation tree. The generation of candidate GFDs is controlled by a GFD *generation tree* $T = (V_T, E_T)$. More specifically, (1) each node $v \in V_T$ at level i of tree T stores a pair $(Q[\bar{x}], \mathsf{lvec})$, where (a) $v.Q[\bar{x}]$ is a graph pattern with i edges, and (b) $v.\mathsf{lvec}$ is a vector, in which each entry $\mathsf{lvec}[l]$ records a literal tree rooted at a literal l. Here, l is $x.A = c$ or $x.A = y.B$, for $x, y \in \bar{x}$, A and B are active attributes in Γ, and c is a constant in G. Each node at level j of entry $\mathsf{lvec}[l]$ is a literal set X such that $Q[\bar{x}](X \to l)$ is a candidate GFD. There is an edge (X_1, X_2) in $v.\mathsf{lvec}[l]$ if there exists a literal l' such that $l' \notin X_2$ and $X_1 = X_2 \cup \{l'\}$. (2) Each node $v(Q[\bar{x}], \mathsf{lvec})$ has an edge $(v, v') \in E_T$ to another $v'(Q'[\bar{x}], \mathsf{lvec}')$ if pattern Q' extends Q by adding a single edge.

Example 6.1. A fraction of a GFD generation tree T is shown in Figure 4. It contains a node $v(Q_1[\bar{x}], \mathsf{lvec})$ at level 1, and $v'(Q_1'[\bar{x}_1], \mathsf{lvec}')$ at level 2. Node $v(Q_1[\bar{x}], \mathsf{lvec})$ stores graph pattern $Q_1[\bar{x}]$ of Figure 1, and its literal tree $v.\mathsf{lvec}[l]$ rooted at literal $l = (x.\mathsf{type} = \text{“producer”})$. At node $v'(Q_1'[\bar{x}_1], \mathsf{lvec}')$, a literal tree is rooted at a different literal $l' = (y.\mathsf{type} = \text{“film”})$. There exists an edge from X' to X'' in literal tree $v'.\mathsf{lvec}'[l']$, since $X'' = X' \cup \{z.\mathsf{name} = \text{“Academy best picture”}\}$. There exists an edge from $v(Q_1[\bar{x}], \mathsf{lvec})$ to $v'(Q_1'[\bar{x}_1], \mathsf{lvec}')$ in T, since Q_1' is obtained by adding a single edge (y, z) to pattern Q_1. The literal at node X in literal tree $v.\mathsf{lvec}[l]$ encodes the GFD φ_1 of Example 1.1. Similarly, X'' in literal tree $v'.\mathsf{lvec}'[l']$ encodes GFD $\varphi_4 = Q_1'[x, y, z](\{x.\mathsf{type} = \text{“producer”}, z.\mathsf{name} = \text{“Academy best picture”}\} \to y.\mathsf{type} = \text{“film”})$.

Observe that for each candidate GFD $\varphi = Q[\bar{x}](X \to l)$ at level i of the tree T, the length $|X|$ is at most $J = 2i^2|\Gamma|(|\Gamma| + 1)$, where Γ consists of active attributes in G.

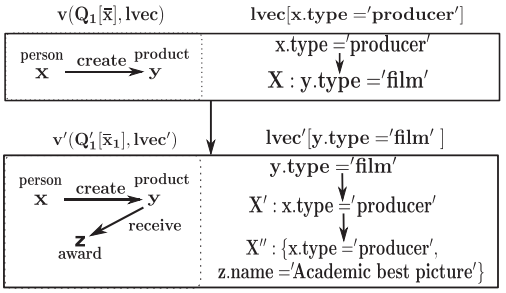


Fig. 4. A GFD generation tree.

GFD Spawning. Employing a generation tree T, algorithm SeqDis spawns new candidate GFDs by performing the following two "atomic" operations.

Vertical spawning. Operation VSpawn(i) creates new nodes $v'.Q'$ at level i by adding one edge e to patterns $v.Q$ at level $i-1$. It adds an edge (v, v') to T, growing T levelwise vertically.

Intuitively, VSpawn(i) adds new patterns to T, for $1 \le i \le k^2$. For each GFD $\varphi = Q[\bar{x}](X \to l)$ at level $i-1$, it generates pattern Q' by adding one edge to Q. It finds matches $h(\bar{x})$ of Q'. It also associates Q' with (a) "frequent" edges e' that connect to nodes in $h(\bar{x})$, and (b) literals $h(x).A = c$ or $h(x).A = h(y).B$ for $x, y \in \bar{x}$, taking A, B from Γ and constants c from G. VSpawn expands patterns with e'.

Moreover, for each pattern Q_1 at level i, VSpawn maintains a set $P(Q_1)$ of edges from the *parents* of Q_1 at level $i-1$ (we will see its use in Section 7). If Q_1' is added to iso(Q_1), then $P(Q_1')$ is merged into $P(Q_1)$. Here, iso(Q_1) is the set of patterns at level i that are isomorphic to Q_1.

Example 6.2. Consider the generation tree T depicted in Figure 4. A graph pattern Q_1' is spawned by VSpawn(2) from Q_1, by adding a single edge $e = (y, z)$.

Horizontal spawning. HSpawn generates literals with the attributes and constants. More specifically, HSpawn(i, j) executes at level j of the literal trees of all level-i patterns in T. It generates a set of candidate GFDs $\varphi = Q[\bar{x}](X \to l)$, where Q ranges over all level-i patterns, $|X| = j$, and literals in X and l take attributes in Γ and constants in G collected by VSpawn (see details shortly).

That is, when $j = 0$, HSpawn(i, j) adds a candidate GFD $Q[\bar{x}](\emptyset \to l)$ with a literal l of G. For $j > 0$, it generates level-j GFDs $\varphi' = Q[\bar{x}](X \cup \{l'\} \to l)$ from a GFD $\varphi = Q[\bar{x}](X \to l)$ at existing level-i nodes by adding a literal l' to X. It grows tree T levelwise horizontally, but adds no new graph patterns. We denote the set of candidate GFDs generated by HSpawn(i, j) as $\Sigma_{C_{ij}}$.

Example 6.3. Continuing Example 6.2, HSpawn(2, j) is performed on the newly added patterns at level 2 of tree T. For pattern Q_1' and literal tree $v'.\text{lvec}'$ rooted at literal $l' = (y.\text{type} = \text{"film"})$, HSpawn(2, 2) extends X' at level $j = 1$ to X'' at $j = 2$, by adding a literal $z.\text{name} = \text{"Academy best picture."}$ This yields a new candidate GFD$\varphi_4 = Q_1'[x, y, z](X'' \to l)$ (see Figure 4).

These steps iterate until either no new GFDs can be spawned, or when j reaches $J = 2i^2|\Gamma|(|\Gamma| + 1)$, the maximum length of literal set X. By the levelwise generation of candidates and Lemma 6.5 to be presented shortly, the GFDs validated are guaranteed to be minimum.

Upgrade. SeqDis also "upgrades" GFDs at level i. For each GFD $\varphi = Q[\bar{x}](X \to l)$ discovered and validated, it collects all variants of φ into a set, i.e., those GFDs $\varphi' = Q'[\bar{x}](X' \to l')$ such that pattern Q' has an isomorphic structure as Q except different node or edge labels, and X' (respectively, l') is the same as X (respectively, l) subject to variable renaming based on the

isomorphism. It checks in the set whether a label $L_Q(v)$ (respectively, $L_Q(e)$) ranges over all node (respectively, edge) labels in Θ; if so it upgrades φ to φ^- by replacing $L_Q(v)$ (respectively, $L_Q(e)$) with "_," and substitutes φ^- for all the corresponding variants. The set $P(Q)$ of parents is also extended by merging all $P(Q')$ of corresponding variants.

Example 6.4. Consider GFD $\varphi_4 = Q_1'[x, y, z](X'' \to l)$ (Figure 4). Suppose that the active domain Θ_x of $L_Q(x)$ in G is {"staff," "crew," "person"}. After validation, SeqDis finds another two GFDs $\varphi_7 = Q_7[x, y, z](X'' \to l)$ and $\varphi_8 = Q_8[x, y, z](X'' \to l)$, where $Q_7[x, y, z]$ (respectively, $Q_8[x, y, z]$) is a variant of Q_1' by changing label $L_Q(x)$ from "person" to "staff" (respectively, "crew"). Since φ_4, φ_7, and φ_8 are verified minimum positive GFDs, and because $L_Q(x)$ ranges over Θ_x, they are all upgraded to a single GFD $\varphi_4^-= Q_9[x, y, z](X'' \to l)$, by replacing $L_Q(x)$ in Q_1' with "_." The GFD φ_4^- is added to Σ, instead of φ_4, φ_7 and φ_8. It then extends the parent set $P(Q_9)$ in the GFD tree T as $P(Q_1') \cup P(Q_7) \cup P(Q_8)$.

Pruning. By the following lemma, at pattern Q and literal l, (1) HSpawn stops expanding literals X as soon as it is verified that $G \models Q[\bar{x}](X \to l)$, and (2) VSpawn stops expanding pattern Q if $\text{supp}(Q, G) < \sigma$. These strategies ensure the feasibility of GFDs discovery in practice.

Lemma 6.5. *For a cover Σ_c of* GFDs *with support above threshold σ,*

(a) Σ_c includes no trivial GFDs *φ;*
(b) for any φ=$Q[\bar{x}](X \to l)$, if $G \models \varphi$ then Σ_c does not include $\varphi' = Q[\bar{x}](X' \to l)$ if $X \subsetneq X'$; and
(c) if a GFD *$\varphi = Q[\bar{x}](X \to l)$ has $\text{supp}(Q, G) < \sigma$, then Σ_c does not include $\varphi' = Q'[\bar{x}](X' \to l')$ if $Q \ll Q'$.*

Proof. (a) For a trivial GFD φ, we have that $\text{supp}(\varphi, G) = 0 < \sigma$. Hence, φ is not included in Σ_c.

(b) When $G \models \varphi$, we show that Σ_c cannot contain φ'. Indeed, if $\text{supp}(\varphi) \geq \sigma$, then φ is a candidate for Σ_c but φ' is not, since φ' is not reduced. If $\text{supp}(\varphi) < \sigma$, then by Theorem 5.2, $\text{supp}(\varphi') \leq \text{supp}(\varphi)$ and hence φ' cannot make a candidate for Σ_c either.

(c) If $Q \ll Q'$, then $\text{supp}(Q, G) \geq \text{supp}(Q', G)$ as verified in the proof of Theorem 5.2. Moreover, we can deduce that $\text{supp}(\varphi', G) \leq \text{supp}(Q', G)$ by the definition of $\text{supp}(\varphi', G)$. Since $\text{supp}(Q, G) < \sigma$, we have that $\text{supp}(\varphi', G) \leq \text{supp}(Q', G) \leq \text{supp}(Q, G) < \sigma$. That is, GFD φ' cannot be included in Σ_c. □

6.2 Discovering negative GFDs

Unlike conventional FD mining, SeqDis discovers both positive and negative GFDs *simultaneously*. It uses a set Σ_N to maintain negative GFDs. Recall that a negative GFD can be (a) $Q[\bar{x}](\emptyset \to \text{false})$, or (b) $Q[\bar{x}](X \to \text{false})$ if $X \neq \emptyset$. At each iteration i, SeqDis triggers (1) *negative vertical* NVSpawn that extends VSpawn to find negative GFDs of case (a), in the pattern matching step, and (2) *negative horizontal* NHSpawn that extends HSpawn to find GFDs of case (b), in the validation step.

Discover negative GFDs *in case (a).* In this case, pattern Q is expanded from a pattern Q' by adding a single edge, where $\text{supp}(Q', G) \geq \sigma$, since otherwise $\text{supp}(\varphi, G) = \max \text{supp}(Q', G) < \sigma$. Hence, NVSpawn($i$) is triggered by VSpawn(i) at iteration i, once the set $\boldsymbol{Q}_i$ of all the level-i patterns is generated and verified. It finds all patterns $Q \in \boldsymbol{Q}_i$ with $\text{supp}(Q, z) = 0$, and adds candidate GFD $\varphi = Q[\bar{x}](\emptyset \to \text{false})$ to Σ_N. It guarantees that $\text{supp}(\varphi, G) \geq \sigma$ by the existence of Q'.

Discover negative GFDs *in case (b).* In this case a negative GFD φ extends a positive minimum $\varphi' = Q[\bar{x}](X' \to l)$ by adding a single literal to X'. Moreover, $\text{supp}(\varphi', G) \geq \sigma$, since otherwise $\text{supp}(\varphi, G) = \max \text{supp}(\varphi', G) < \sigma$. Hence, NHSpawn($i, j$) extends HSpawn($i, j$) as follows. As soon

as HSpawn(i, j) verifies that $G \models \varphi'$ and $\text{supp}(\varphi', G) \geq \sigma$, it generates negative candidates $\varphi = Q[\bar{x}](X \rightarrow \text{false})$, where X extends X' with a single literal. It checks whether $Q(G, X, z) = 0$ and adds φ to Σ_N if so. It guarantees that $\text{supp}(\varphi, G) \geq \sigma$ due to the existence of φ'.

Example 6.6. Recall negative GFD φ_3 from Example 3.1. It is of case (a) above, and is discovered as follows. Algorithm SeqDis first finds a pattern Q as a single edge from person x to person y, and adds an edge with VSpawn(1) to get pattern Q_3 as shown in Figure 1. It triggers NVSpawn(1) to verify that $\text{supp}(Q_3, x) = 0$, and adds φ_3 to Σ_N as a negative GFD.

6.3 Sequential Cover Computation

Given a set Σ of GFDs computed by SeqDis, algorithm SeqCover computes a cover Σ_c of Σ. It is based on the characterization of GFD implication [27] reviewed in Section 4.

Algorithm SeqCover. Making use of the characterization and following the algorithms for computing cover of relational FDs (see, e.g., Reference [3]), SeqCover works as follows: for each $\varphi \in \Sigma$, it checks whether $\Sigma \backslash \{\varphi\} \models \varphi$ based on the characterization; if so, then it removes φ from Σ. It iterates until no more φ can be removed, and ends up with Σ_c. This is inherently sequential, inspecting φ one by one.

Algorithm SeqDis correctly generates and validates the set Σ of all k-bounded σ-frequent minimum GFDs, removing trivial and non-reduced GFDs by Lemma 6.5. Moreover, SeqCover correctly computes a cover of Σ. (1) For each $\varphi \in \Sigma$, it checks implication based on the characterization of GFD implication. (2) When it terminates, no more φ can be removed from Σ_c. Thus, Σ_c is a cover of Σ.

6.4 Analysis of Sequential GFD Discovery

We next analyze the complexity of SeqDisGFD, which consists of the following two parts.

*Mining cost (*SeqDis*).* Denote by $\mathsf{C}(k, G)$ the number of k-bounded GFD candidates in graph G. Algorithm SeqDis checks $\mathsf{C}(k, G)$ many candidates, and validates each. Validating a GFD involves subgraph isomorphism, which takes $O(|G|^k)$ time in the worst case. This is the best a sequential algorithm could do so far: "For subgraph isomorphism, nothing better than the naive exponential $|G_2|^{|G_1|}$ bound is known" [19]. This is due to the intractable nature of the problem, unless P = NP. Thus, SeqDis takes $O(\mathsf{C}(k, G) \cdot |G|^k)$ time in the worst case, denoted by $t_1(|G|, k, \sigma)$.

*Implication (*SeqCover*).* Let $\Sigma = \{\varphi_1, \ldots, \varphi_M\}$. Denote by $T(\Sigma, \varphi_i)$ the cost of checking $\Sigma \setminus \{\varphi_i\} \models \varphi_i$ by a "best" sequential algorithm $\mathcal{A}_c$. Denote by $t_2(\Sigma, k)$ the sum of $T(\Sigma, \varphi_i)$ for $i \in [1, M]$. It takes $O(t_2(\Sigma, k))$ time, since SeqCover removes redundant GFDs one by one.

Taken together, the overall cost of algorithm SeqDisGFD, denoted by $t(|G|, k, \sigma)$, is in $O(t_1(|G|, t, \sigma) + t_2(\Sigma, k))$ time. As argued above, this indicates the worst-case cost of any sequential algorithm for GFD discovery, which subsumes subgraph isomorphism.

7 PARALLEL DISCOVERY ALGORITHMS

Real-life graphs are often big, and we have seen that GFD discovery is costly. Nonetheless, we show that GFD discovery is feasible in large-scale graphs by providing a parallel scalable algorithm.

7.1 Parallel Scalability Revisited

To characterize the effectiveness of parallel GFD discovery in large-scale graphs, we revisit the notion of *parallel scalability* that was introduced in Reference [45] and has been widely adopted. Consider a yardstick sequential algorithm $\mathcal{A}$ that, given a graph G, a bound k and support σ,

finds a cover Σ_c of k-bounded minimum σ-frequent GFDs. Denote its worst-case running time as $t(|G|, k, \sigma)$.

We say that a parallel algorithm $\mathcal{A}_p$ for GFD discovery is *parallel scalable relative to* $\mathcal{A}$ if its running time by using n processors can be expressed as follows:

$$T(|G|, n, k, \sigma) = \tilde{O}\left(\frac{t(|G|, k, \sigma)}{n}\right),$$

where the notation $\tilde{O}$ hides $\log(n)$ factors (see, e.g., Reference [56]).

Intuitively, parallel scalability guarantees speedup of $\mathcal{A}_p$ *relative to* a "yardstick" sequential algorithm [45]. A parallel scalable algorithm $\mathcal{A}_p$ is able to "linearly" reduce the sequential cost of $\mathcal{A}$ when more processors are used. The main result of this section is as follows.

Theorem 7.1. *There exists an algorithm* DisGFD *for* GFD *discovery that is parallel scalable relative to* SeqDisGFD *and generates a set of* GFDs *that are equivalent to those discovered by* SeqDisGFD.

The main conclusion we can draw from Theorem 7.1 is that in theory, the more processors are used, the faster algorithm DisGFD is. Hence, algorithm DisGFD can scale with large graphs G by adding processors as needed. It makes GFD discovery feasible in real-life graphs. Note that parallel algorithm DisGFD and sequential algorithm SeqDisGFD may output *different but equivalent* sets of GFDs, since given a set Σ of GFDs, there may exist multiple equivalent covers Σ_c of Σ.

A proof sketch. We provide such a DisGFD as a proof of Theorem 7.1. It consists of two algorithms: (a) ParDis (Section 7.2) that "parallelizes" sequential algorithm SeqDis to discover the set Σ of all k-bounded minimum σ-frequent GFDs from G, and (b) ParCover (Section 8) that "parallelizes" SeqCover to compute a cover Σ_c of Σ. We will show that these algorithms are parallel scalable relative to their sequential counterparts SeqDis and SeqCover, respectively. We can also show that SeqDisGFD and DisGFD generate sets of GFDs that are equivalent to each other. □

Both algorithms work with a master S_c and n workers, on a graph G that is evenly partitioned into n fragments $(F_1, \ldots, F_n)$ via vertex cut [40], and distributed across n workers $(P_1, \ldots, P_n)$.

7.2 Parallel GFD Mining

We start with algorithm ParDis, shown as Algorithm 2. It works almost the same as SeqDis, except that it performs graph pattern matching and GFD validation in parallel (lines 7 and 13).

More specifically, the algorithm runs in supersteps. It uses Σ_i to store all minimum σ-frequent GFDs with i edges, at superstep i. ParDis first initializes Σ and tree T (lines 1 and 2), and then performs at most k^2 supersteps (lines 3–15). At each superstep i ($1 \leq i \leq k^2$), ParDis generates and verifies GFD candidates in parallel, by further "parallelizing" the core steps, i.e., pattern verification (vertical spawning) and GFD validation (horizontal spawning) steps of algorithm SeqDis. More specifically, it performs the following.

(1) Parallel pattern verification. Algorithm ParDis performs vertical spawning VSpawn(i) (Section 6.1) at master S_c to generate graph patterns at level i of T (line 4 of Algorithm 2). It conducts parallel pattern matching if there exist new patterns spawned (line 7; see details below), to find matches for all the patterns at level i that contribute to candidate GFDs.

(2) Parallel GFD *validation.* Algorithm ParDis then performs horizontal spawning HSpawn(i, j) (Section 6.1) at master S_c with the verified graph patterns to generate a set of candidate GFDs. Operation HSpawn(i, j) iterates for $j \in [1, J]$, where $J = 2i^2|\Gamma|(|\Gamma| + 1)$ (see Section 6.1), followed by parallel validation of the candidate GFDs (lines 9–14). Once each superstep i terminates, the set Σ is expanded by including all verified minimum frequent GFDs in Σ_i at this level (line 18).

ALGORITHM 2: Algorithm ParDis

Input: a fragmented graph G, integer k, support threshold σ.
Output: a set Σ of all k-bounded minimum σ-frequent GFDs φ.

```
1  set Σ := ∅; GFD tree T := ∅; integer i := 1; flag_V := true;
2  SpawnGFD(T); /* initialize generation tree T with single-node GFDs */ ;
3  while i ≤ k^2 and flag_V /* superstep i */ do
4      VSpawn(i); Σ_i := ∅;
5      flag_V := false if no new pattern is spawned;
6      if flag_V then
7          parallel graph pattern matching;
8          j := 1; flag_H := true;
9          while j ≤ J and flag_H do
10             set Σ_{C_ij} := HSpawn(i, j);
11             flag_H := false if no new GFD candidates are spawned;
12             if flag_H then
13                 parallel GFD validation for Σ_{C_ij};
14                 Σ_i := Σ_i ∪ Σ_{C_ij}; j := j + 1;
15             end
16         end
17     end
18     Σ := Σ ∪ Σ_i i := i + 1;
19 end
20 return Σ;
```

The two steps iterate until no new candidate GFDs can be spawned, or all the k-bounded GFDs are checked (i.e., $i = k^2$). When one of the conditions is satisfied, ParDis returns Σ (line 20).

We next present the details of *parallel verification* and *parallel validation*, which dominate the cost of GFD discovery. We will see that both steps are parallel scalable.

Parallel pattern matching. Denote the set of graph patterns generated by VSpawn(i) as Q'_i. Algorithm ParDis conducts *incremental* pattern matching in parallel as follows.

(1) At master S_c, for each pattern $Q' \in Q'_i$, algorithm ParDis constructs a *work unit* (Q, e) such that Q' is obtained by adding an edge e to a verified pattern Q in the tree T. The work unit is a request that "performs a *join*$Q(F_s) \bowtie e(F_t)$ to compute $Q'(F_s)$ locally at fragment F_s, for all $t \in [1, n]$," where $Q(F_s)$ denotes the set of all matches of Q in F_s, and $e(F_t)$ and $Q'(F_s)$ are defined similarly. Here edge e is treated as a single-edge pattern. It then distributes the work units to n workers to be computed in parallel, following a workload balancing strategy (see details below).

(2) Upon receiving a set of work units, each worker P_s *incrementally* compute $Q'(F_s)$, by (a) joining the locally verified matches $Q(F_s)$ with $e(F_t)$ for $t \in [1, n]$, where $e(F_t)$ is shipped from P_t to P_s if $s \neq t$; and (b) verifying matches $Q'(F_s)$ by isomorphism check. After this, worker P_i stores matches $Q'(F_s)$ for the next round. Once all the patterns are verified, it sends a flag Terminate to master S_c.

The correctness is ensured by $Q'(G) = \bigcup_{s \in [1,n]} Q'(F_s)$ and $Q'(F_s) = \bigcup_{t \in [1,n]} Q(F_s) \bowtie e(F_t)$.

Intuitively, $Q'(G)$ is a set of k-ary tuples, in which some attributes may not be instantiated yet. Each tuple $h(\bar{x})$ encodes a match of $Q'[\bar{x}]$ in G, for $|\bar{x}| \leq k$. Note that i is the number of edges in Q', while matches are node mappings. ParDis checks whether $h(\bar{x})$ makes an isomorphism of Q'.

Algorithm ParDis also associates Q' with (a) a set of "frequent" edges e' that connect to nodes in an $h(\bar{x})$ of $Q'(G)$, and (b) a set of literals $h(x).A = c$ for $x \in \bar{x}$, taking A from active domain Γ of

attributes and constants c from graph G. These sets are sent to master S_c, where VSpawn expands patterns with edges e' and HSpawn generates literals using the attributes and constants.

Example 7.2. Consider the generation tree T depicted Figure 4. When algorithm ParDis executes VSpawn(2), it spawns a new pattern Q_1' (among others) from Q_1, by adding an edge $e = (y, z)$. Its corresponding work unit $(Q_1, (y, z))$ is to compute $Q_1'(F_s) = Q_1(F_s) \bowtie e(F_t)$ for $t \in [1, n]$ at each worker P_s, in parallel. Here $Q_1(F_s)$ was computed by VSpawn(1), consisting of tuples $h_{Q_1}(x, y)$ that match edge create(x, y) in Q_1; and $e(F_t)$ includes $h_e(y_1, z)$ as matches of receive(y_1, z). The join of the two tables yields $h_{Q'}(x, y, z)$ when $y = y_1$. The isomorphic checking ensures that x, y, and z are distinct nodes, i.e., $h_{Q'}(x, y, z)$ is indeed an 1-to-1 mapping.

Load balancing. We initially partition the edges of graph G evenly across n workers via vertex cut. That is, the size $|e(F_t)|$ of edge set at each worker is bounded by $O(\frac{|E|}{n})$. This helps us cope with skewed graphs, in which a large number of low-degree nodes connect to dense and small groups. For $Q(F_s) \bowtie e(F_t)$, if no $Q(F_s)$ is "skewed," i.e., much larger than other $Q(F_t)$'s, we compute $Q(F_s) \bowtie e(F_t)$ locally at each worker P_s. Otherwise, we re-distribute $Q(F_s) \bowtie e(F_t)$ evenly across workers and compute it in parallel. That is, the algorithm balances (a) parallel validation workload, and (b) parallel matching work unit $Q'(F_s) \bowtie e'(F_t)$ in the next superstep.

More specifically, a work unit $Q(F_s) \bowtie e(F_t)(t \in [1, n])$ is *skewed* if it takes more cost than the others. ParDis quantifies the "skewness" of $Q(F_s)$ in terms of $\frac{|Q(F_s)|}{\text{Avg}_{t\in[1,n]}|Q(F_t)|}$. If the skewness of $Q(F_s)$ exceeds a threshold ϵ, then the work unit is skewed. Since pattern Q is already verified in superstep $i - 1$, the checking is performed locally at master S_c, and it adds little verification cost.

For each skewed $Q(F_s)$, worker P_s evenly distributes $Q(F_s)$ to all n workers, such that each worker processes $\frac{|Q(F_s)|}{n}$ tuples. For each tuple $h_s \in Q(F_s)$ sent to another worker P_t, P_t "merges" h_s with its local $Q(F_t)$, and computes $Q(F_s) \bowtie e(F_t)(t \in [1, n])$ as above, in parallel.

Example 7.3. Recall pattern Q_1' and its corresponding work unit (Q_1, e) from Example 7.2. Consider three workers P_1, P_2, and P_3, which store the following for (Q_1, e). Assume a threshold $\epsilon = 1.5$.

	P_1	P_2	P_3
$Q(F_s)$	$< x_j, y_1 >$ $(j \in [1, 300])$	$< x_2, y_2 >$	$< x_3, y_3 >$
$e(F_s)$	$< y_1, z_1 >$	$< y_3, z_3 >$	$< y_2, z_3 >$
$Q(F_s)$	$< x_i, y_1 >$ $(i \in [1, 100])$	$< x_j, y_1 >$ $(j \in [101, 200])$ $< x_2, y_2 >$	$< x_k, y_1 >$ $(k \in [201, 300])$ $< x_3, y_3 >$
$e(F_s)$	$< y_1, z_1 >$	$< y_1, z_1 >$ $< y_2, z_2 >$ $< y_3, z_3 >$	$< y_1, z_1 >$ $< y_2, z_2 >$ $< y_3, z_3 >$

The loads of workers P_1, P_2, and P_3 measured by $|Q(F_s)|$ are 300, 1 and 1, respectively. Algorithm ParDis finds $Q(F_1)$ skewed, which contains 300 matches, with skewness $\frac{300}{\frac{302}{3}} = 2.9$ above threshold $\epsilon = 1.5$. Worker P_1 thus redistributes tuples of $Q(F_1)$ evenly across P_1-P_3 (as shown in the table above). Each worker then requests $e(F_t)$ from all other workers, and computes their local join in parallel. The balanced loads are 101, 104, and 104 for P_1, P_2 and P_3, respectively.

Parallel validation. Once all workers complete parallel pattern matching and send Terminate to master S_c, algorithm ParDis performs HSpawn(i, j) to generate candidate GFDs $\Sigma_{C_{ij}}$ coordinated

at (i, j) of T. It then posts $\Sigma_{C_{ij}}$ to the n workers, to validate the GFDs in parallel. Workload balancing is performed when necessary following the same strategy as for parallel pattern matching.

(1) For each $\varphi = Q[\bar{x}](X \to l)$ in $\Sigma_{C_{ij}}$, each worker P_s computes in parallel (a) local supports $\mathsf{supp}(\varphi, F_s) = |Q(F_s, Xl, \bar{z})|$ and (b) a Boolean flag SAT^i_φ, which is set to true if $F_s \models \varphi$. It then sends $\mathsf{supp}(\varphi, F_s)$ and SAT^i_φ to S_c. To do these, it uses the matches $Q(F_s)$ computed by $\mathsf{VSpawn}(i)$.

(2) When all workers P_s's complete their local validation, for each candidate GFD $\varphi \in \Sigma_{C_{ij}}$, algorithm ParDis checks at master S_c whether $\mathsf{supp}(\varphi, G) = \sum_{s\in[1,n]} \mathsf{supp}(\varphi, F_s) \geq \sigma$, and $\bigwedge_{s\in[1,n]} \mathsf{SAT}^s_\varphi = \text{true}$. If so, then it adds φ to the set Σ_i as a verified frequent GFD.

Example 7.4. Continuing with Example 7.2, $\mathsf{HSpawn}(2, j)$ is performed on the newly generated patterns at level 2 of generation tree T. For pattern Q'_1 and literal tree $v'.\mathsf{lvec}'$ rooted at $l' = (y.\mathsf{type} = \text{“film”})$, $\mathsf{HSpawn}(2, 2)$ extends X' at level $j = 1$ to X'' at level $j = 2$, by adding a new literal $z.\mathsf{name} = \text{“Academy best picture.”}$ This yields a candidate GFD$\varphi_4 = Q_1[x, y, z](X'' \to l)$ (see Figure 4). By Lemma 6.5, ParDis generates φ_4 only if $G \not\models Q_1[x, y, z](X' \to l)$.

Just like algorithm SeqDis, algorithm ParDis also does the following: “upgrade” a node label $L_Q(v)$ in a GFD φ to wildcard “_” if $L_Q(v)$ ranges over all labels of Θ in the variants of φ, to get a more general pattern; similarly for edge $L_Q(e)$ (see Section 6.1 for more details).

Algorithm. Putting these together, we present the main driver of algorithm ParDis in Algorithm 2. It initializes the set Σ and tree T (line 2), and performs at most k^2 supersteps (lines 3–18). In each superstep i, (1) $\mathsf{VSpawn}(i)$ generates graph patterns at level i of T (line 4), and conducts parallel pattern matching if there exist new patterns spawned (line 7). (2) It then performs at most J rounds of $\mathsf{HSpawn}(i, j)$ with $j \in [1, J]$, where $J = 2i^2|\Gamma|(|\Gamma| + 1)$, followed by parallel validation of the candidate GFDs (lines 9–14). Once each superstep i terminates, Σ is expanded by including all verified minimum frequent GFDs Σ_i (line 18). It finally returns Σ (line 20).

Analysis. We next show the following: (1) ParDis and SeqDis generate the same set of GFDs; and (2) ParDis is parallel scalable relative to SeqDis.

(1) Correctness. We first show that ParDis and SeqDis discover the same set of GFDs. Observe that the differences between ParDis and SeqDis are at lines 7 and 13 (Algorithms 1 and 2), where ParDis performs verification and validation in parallel, while SeqDis checks these sequentially. Since ParDis generates all work units and GFDs in a single machine (i.e., the master S_c) like SeqDis, we only need to show that ParDis conducts verification and validation correctly. In the following, we prove the correctness of parallel verification; the proof for its parallel validation is similar.

We now show that ParDis conducts verification correctly. Observe that for each pattern Q, ParDis constructs a work unit (Q_1, e) such that Q is an extension of Q_1 with an edge e, and computes $Q(G)$ by performing a join $Q_1(F_s) \bowtie e(F_t)$ to compute $Q(F_s)$ locally at fragment F_s, for all $t \in [1, n]$. Then it suffices to prove that $\bigcup_{s\in[1,n]} Q(F_s) = Q(G)$. We show this by induction on the number of nodes in Q. (a) When Q has only one pattern node v, ParDis finds all nodes with the same label as v in fragment F_s; then obviously $\bigcup_{s\in[1,n]} Q(F_s) = Q(G)$. (b) Assume that for pattern Q with size k ($k > 1$), ParDis constructs a work unit (Q_1, e) such that Q_1 has $k - 1$ nodes and $\bigcup_{s\in[1,n]} Q_1(F_s) = Q_1(G)$; since ParDis computes $Q_1(F_s) \bowtie e(F_t)$ for all $t \in [1, n]$, we know that ParDis computes $Q_1(F_s) \bowtie \bigcup_{t\in[1,n]} e(F_t) = Q_1(F_s) \bowtie e(G)$ in fragment F_s. Therefore, ParDis finds $\bigcup_{s\in[1,n]}(Q_1(F_s) \bowtie e(G)) = Q_1(G) \bowtie e(G) = Q(G)$. That is, ParDis correctly verifies $Q_1(G)$.

(2) Parallel scalability. It suffices to show that its parallel matching and validation of each candidate φ are in $O(\frac{|G|^k}{n})$ time, no matter in which superstep φ is processed. For if it holds, ParDis takes at most $\tilde{O}(\mathsf{C}(k, G) \cdot \frac{|G|^k}{n}) = \tilde{O}(\frac{t_1(|G|,k,\sigma)}{n})$ time.

We analyze the computation and communication costs of parallel matching; the argument for parallel validation is similar. The dominating cost of matching at each worker is incurred by the following steps: (a) broadcast its local share of $e(F_s)$ to other workers, which is in $O(\frac{|G|}{n})$ time, since vertex cut evenly distributes $e(F_s)$; (b) receive $e(F_t)$ from other workers, in time $O(\frac{(n-1)|G|}{n}) < O(\frac{|G|^2}{n})$, since $n \ll |G|$ (note that we assume $k \geq 2$ to simplify the discussion); (c) balance load $Q(F_s) \bowtie e(G)$, where $e(G)$ denotes the set of matches of pattern edge e in G; (d) locally compute $Q(F_s) \bowtie e(G)$, in time $O(\frac{|G|^k}{n})$, since the load is evenly distributed in step (c). We show below that step (c) is in $O(\frac{|G|^k}{n})$ time. Taken together, the parallel cost of pattern matching is $O(\frac{|G|^k}{n})$.

We next show that when computing $Q(F_s) = Q'(F_s) \bowtie e(G)$, each worker P produces at most $O(\frac{|G|^{|\bar{x}|}}{n})$ tuples, denoted by $P(Q(F_s))$, where $Q[\bar{x}]$ is a graph pattern, $|\bar{x}|$ is the number of nodes in Q. If this holds, then each worker sends or receives at most $O(\frac{|G|^{|\bar{x}|}}{n})$ tuples when ParDis distributes $Q(F_s)$ for load balancing, in time $O(\frac{|G|^{|\bar{x}|}}{n})$; this verifies the cost of step (c) in the analysis above.

We show that $P(Q(F_s)) \leq \frac{|G|^{|\bar{x}|}}{n}$ by induction on the number $|E_Q|$ of edges of Q.

When $|E_Q| = 1$, $Q(F_s) = Q'(F_s) \bowtie e(G)$ and Q' consists of a single node. Here $Q'(F_s)$ is a set of nodes of G, $e(G)$ is a set of edges of G, and $|\bar{x}| = 2$. Thus, $|P(Q(F_s))| \leq |G| \leq \frac{|G|^2}{n}$ as $n \ll |G|$. This is the first step of ParDis via VSpawn. It takes $O(\frac{|G|^2}{n})$ time: evenly distributing $Q(F_s)$ takes $O(|G|) \leq O(\frac{|G|^2}{n})$ time as $n \ll |G|$, and fetching $e(G)$ takes $O(|G|) \leq O(\frac{|G|^2}{n})$ time (step (b) above).

Assume that $|P(Q(F_s))| \leq \frac{|G|^{|\bar{x}|}}{n}$ for any Q with w edges. Consider $Q(F_s) = Q'(F_s) \bowtie e(G)$ with $w+1$ edges. Suppose that $e(G)$ is to match edge (x, y) in Q, where x and y are variables in $Q[\bar{x}]$. There are two cases to consider. (a) When both x and y are in $\bar{x}'$ of $Q'[\bar{x}']$, $\bar{x} = \bar{x}'$. Then $|P(Q(F_s))| \leq |P(Q'(F_s))|$, since $Q'(F_s) \bowtie e(G)$ further restricts $Q'(F_s)$ with the connectivity of x and y. Thus, $|P(Q(F_s))| \leq \frac{|G|^{|\bar{x}|}}{n}$ by the induction hypothesis. (b) Otherwise, at least one of x and y is not in $\bar{x}'$. Thus, $|\bar{x}| \geq |\bar{x}'| + 1$. Then, $|P(Q(F_s))| \leq |P(Q'(F_s))||G| \leq \frac{|G|^{|\bar{x}'|}}{n}|G| \leq \frac{|G|^{\bar{x}}}{n}$ by the induction hypothesis. Hence, $|P(Q(F_s))| \leq \frac{|G|^{\bar{x}}}{n}$, i.e., the claim holds when $|E_Q| = w + 1$.

One can verify that when discovery of negative GFDs is considered, algorithm ParDis remains parallel scalable relative to its sequential counterpart SeqDis. Indeed, while NVSpawn introduces additional pattern matching cost (see Section 6.2), the extra workload can be evenly distributed and hence ParDis is parallel scalable relative to the corresponding parts in algorithm SeqDis.

8 PARALLEL IMPLICATION CHECKING

We next develop a parallel scalable algorithm ParCover to find a cover Σ_c of the set Σ of k-bounded minimum σ-frequent GFDs returned by algorithm ParDis. Based on the characterization of GFD implication [27], we present algorithm ParCover, followed by its complexity analysis.

Group checking. Algorithm ParCover parallelizes algorithm SeqCover by leveraging the characterization of GFD implication (see Section 4). (a) It partitions Σ into "groups" $\Sigma^{Q_1}, \ldots \Sigma^{Q_m}$, where each $\Sigma^{Q_j} \subseteq \Sigma$ ($j \in [1, m]$) is the set of GFDs in Σ that pertain to "the same pattern" Q_j. Thus, for any GFD in a group, its pattern is not isomorphic to the pattern of any GFD in another group. (b) It checks implication of the GFDs within each group, in parallel among all the groups. That is, the implication checking is *pairwise independent* among the groups. More specifically, denote by $\Sigma_{Q_j} \subseteq \Sigma$ the set of GFDs with patterns embedded in Q_j.

One can verify the following property based on the characterization of GFD implication.

ALGORITHM 3: Algorithm~ParCover

Input: A set Σ of GFDs , and the GFD tree T generated by ParDis .
Output: A cover Σ_c of Σ.

```
1 set Σ_c := ∅; set W := ∅;
2 for pattern Q_j of T /* partition of Σ with T */ do
3     create groups Σ^{Q_j} ⊆ Σ;
4     construct Σ_{Q_j}; W := W ∪ Σ_{Q_j};
5 end
6 evenly distribute work units W to all workers;
7 Σ_{c_i} := ParImp(W_i); /* local checking load W_i at worker P_i to find non-redundant GFDs in Σ^{Q_i} */ ;
8 Σ_c := the union of all Σ_{c_i};
9 return Σ_c;
```

Lemma 8.1 [Independence]. *For any* GFD $\varphi \in \Sigma^{Q_j}$ *(for* $j \in [1, m]$*),* $\Sigma \setminus \{\varphi\} \models \varphi$ *if and only if* $\Sigma_{Q_j} \setminus \{\varphi\} \models \varphi$.

Algorithm. As shown in Algorithm 3, given a set Σ of GFDs, ParCover first partitions Σ into the groups $\Sigma^{Q_1}, \ldots \Sigma^{Q_m}$ (lines 2–5). More specifically, ParCover first creates a partition of Σ as $\Sigma^{Q_1}, \ldots \Sigma^{Q_m}$, one for each pattern Q_j in Σ. It then constructs group Σ_{Q_j} for each pattern Q_j. This is done by taking advantage of the GFD generation tree T (see Section 6.1). It traces the ancestors of Q_j that are in T, by following the parent edges of $P(Q_j)$ maintained by VSpawn, and so on. Then Σ_{Q_j} includes such GFDs and those in Σ^{Q_j}. This reduces isomorphism tests when computing groups Σ_{Q_j}.

Algorithm ParCover then creates a work unit Σ_{Q_j} for each Σ^{Q_j}. It collects the work units in a set W (line 4), and distributes W evenly to all the workers with load balancing (line 6; see details below). Upon receiving the assigned work units W_j, each worker P_j invokes a (sequential) procedure ParImp(W_j) that checks implication (line 7), in parallel at different workers.

For each group $\Sigma_{Q_i} \in W_j$, ParImp(Σ_{Q_i}) computes the set Σ_{N_i} of all *non-redundant* GFDs in Σ^{Q_i} such that $\Sigma_{Q_i} \setminus \Sigma_{r_i} \models \Sigma_{Q_i}$, where $\Sigma_{r_i} = \Sigma^{Q_i} \setminus \Sigma_{N_i}$, i.e., Σ_{r_i} is the set of all redundant GFDs of Σ^{Q_j}. In other words, Σ_{N_i} consists of those GFDs in Σ^{Q_i} that are not entailed by other GFDs in Σ, based on Lemma 8.1. Procedure ParImp(W_j) returns the union Σ_{c_i} of Σ_{N_i}'s for each $\Sigma_{Q_i} \in W_j$.

Algorithm ParCover terminates when all the work units have been processed, and returns Σ_c as the union of all non-redundant GFDs in these Σ_{c_i}'s (lines 8 and 9).

Example 8.2. Consider a set $\Sigma = \{\varphi_1, \varphi_1', \varphi_3, \varphi_4, \varphi_5, \varphi_6\}$ of GFDs, where (1) φ_1, φ_1', φ_5, φ_6 are verified GFDs at level 1 of generation tree T; φ_1 has pattern Q_1 in Figure 1, φ_1' has a pattern of one edge receive(y, z) in Figure 4, and φ_5 (respectively, φ_6) has a pattern with a single edge parent(x, y) (respectively, parent(y, x)) in Figure 1; and (2) φ_3 and φ_4 are at level 2 of T; φ_3 has pattern Q_3 in Figure 1, and φ_4 has Q_1' of Figure 4. Then $\varphi_1, \varphi_1', \varphi_4$ are embedded in Q_1', and $\varphi_3, \varphi_5, \varphi_6$ are embedded in Q_3.

To compute cover Σ_c of Σ, algorithm ParCover partitions Σ and constructs work units $\Sigma_{Q_1} = \{\varphi_1\}, \Sigma_{\text{receive}(y,z)} = \{\varphi_1'\}, \Sigma_{\text{parent}(x,y)} = \{\varphi_5, \varphi_6\}, \Sigma_{Q_3} = \{\varphi_3, \varphi_5, \varphi_6\}$ and $\Sigma_{Q_1'} = \{\varphi_1, \varphi_1', \varphi_4\}$. The checking breaks down to 5 independent implication tests that are distributed to all workers and conducted in parallel; e.g., for φ_3 and φ_4, it checks whether $\Sigma_{Q_3} \setminus \{\varphi_3\} \models \varphi_3$ and $\Sigma_{Q_1'} \setminus \{\varphi_4\} \models \varphi_4$, respectively.

Load balancing. On a real-life graph G, there are many more distinct patterns Q_j (i.e., work units) than the number n of workers. Hence, we can balance the workload by evenly distributing the units to n workers. We next present the load balancing strategy. By the characterization of

Reference [27], algorithm ParDis estimates cost $c(\Sigma_{Q_j}) = |\Sigma_{Q_j}| * |Q_j|^{|Q_j|}$ for each work unit Σ_{Q_j}. It then solves the *load balancing* problem. Given a set W of work units and n workers, the load balancing problem is to compute an assignment that sends each $\Sigma_{Q_j} \in W$ to a worker P_j, such that $\text{load}(P_i) = \text{load}(P_j)$ for $i, j \in [1, n]$. Here $\text{load}(P_i) = \sum c(\Sigma_{Q_j})$ is the total load of P_j, for all work units Σ_{Q_j} sent to P_j.

This load balancing problem is nontrivial. Nonetheless, there exist efficient approximation algorithms for it, which we can employ in parallel GFD implication.

Proposition 8.3. *The load balancing problem for* GFD *implication is (1)* NP*-complete, and (2) 2-approximable.*

Proof. (1) We first prove Proposition 8.3(1). The decision problem of load balancing for GFD implication is as follows. Given workload W, a set $\mathcal{P}$ of n workers, and a constant ϵ, it is to decide whether there exists an assignment that balances the load of each worker, bounded by $\epsilon \frac{\sum_{w \in W} c(w)}{n}$.

Upper bound. The problem is in NP. Indeed, an NP problem guesses a partition of Σ, and checks whether the sum of the costs is bounded for each partition in PTIME.

Lower bound. We show that the problem is NP-hard by reduction from the *number partition problem*, which is NP-complete [44]. The latter problem is to decide, given a set $S = \{n_1, \ldots, n_m\}$, whether there exists a n-partition of S such that the sum of the numbers in each set is equal.

Given a set S of numbers and a number n, we construct an instance of the load balancing problem as follows. (1) For each number n_i in S, we define a set Σ_{Q_i} of GFDs, which contains n_i GFDs: (a) Σ^{Q_i} contains only one GFD $\varphi_i = Q_i[x, y](x.A = 1 \rightarrow y.B = 1)$, where Q_i consists of a single edge (x, y) labeled with i, and both x and y are labeled with i; and (b) Σ_{Q_i} also contains GFDs $\varphi_i^j = Q_i'[x](x.B = j \rightarrow x.C = j)$ $(j \in [1, n_i - 1])$, where Q_i' consists of only one node labeled with i. We set (a) $c(\Sigma_{Q_i})$ as $|\Sigma_{Q_i}| * |Q_i|^{|Q_i|} = 4 \cdot n_i$, since the number of nodes in Q_i is 2; (b) for each $i_1, i_2 \in [1, m]$, $\Sigma^{Q_{i_1}}$ and $\Sigma^{Q_{i_2}}$ are pairwise independent; (c) $\Sigma = \{\Sigma_{Q_1}, \ldots, \Sigma_{Q_m}\}$; and (d) the threshold $\epsilon = 1$. It is easy to verify that there exists a balanced partition of S if and only if there exists a balanced load of Σ. From this it follows that the load balancing problem is NP-hard.

(2) We next show Proposition 8.3(2). Given W and $\mathcal{P}$, algorithm ParCover adopts a greedy strategy for a general load balancing problem that is developed in References [4, 31] to iteratively assigns work units with the smallest cost to the workers with the (dynamically updated) least load. More specifically: (1) It first computes the cost $c_j = |\Sigma_{Q_j}| \times |Q_j|^{|Q_j|}$ for each group Σ_{Q_j}, and initiates the work load $w_i = 0$ for each processor i. (2) It then does the following until Σ is empty: (a) selects the work unit $\Sigma_{Q_j} \in \Sigma$ with the smallest $c(\Sigma_{Q_j})$ and a worker P_i with a smallest work load w_i; (b) assigns Σ_{Q_j} to P_i, and update $w_i := w_i + c(\Sigma_{Q_j})$; and (c) removes Σ_{Q_j} from Σ.

To show that the algorithm yields a 2-approximation, we construct an L-reduction from the load balancing problem to the general load balancing problem (denoted by GLBP). GLBP is stated as follows [4, 31]: given a set of jobs $J = \{J_1, \ldots, J_m\}$ and a set of processing units $U = \{U_1, \ldots, U_n\}$, compute an assignment λ of jobs in J to processing units in U such that the makespan is minimum. Here makespan is the largest work load among all processing units. Assume that the running time of each job J_i $(i \in [1, m])$ is T_i.

The reduction consists of two functions f and g. (1) Given an instance of the load balancing problem (i.e., a set of work units $W = \{c_1, \ldots, c_m\}$ and a set of workers $P = \{P_1, \ldots, P_n\}$), the function f computes an instance of GLBP such that (a) $J = \{J_1, \ldots, J_m\}$ with $T_i = c_i$ for each J_i $(i \in [1, m])$, and (b) $U = \{P_1, \ldots, P_n\}$. (2) Given an assignment λ of an instance of GLBP constructed via f, the function g assigns the work unites in W to workers in P such that the workload at worker P_i consists of all jobs assigned to processing unit P_i $(i \in [1, n])$ via λ, i.e., if $\lambda(J_k) = P_i$, then the work

unit c_k in W is assigned to worker P_i. It is easy to verify that given a 2-approximate assignment λ for GLBP, i.e., $\sum_{\lambda(J_k)=P_j} T_k \leq 2 \times C_{\mathsf{OPT}}$ for all $j \in [1, n]$, where C_{OPT} is the minimum makespan, the function g guarantees that the workload generated by g satisfies that $\sum_{\lambda(J_k)=P_j} c_k \leq 2 \times C'_{\mathsf{OPT}}$, where $C'_{\mathsf{OPT}} = C_{\mathsf{OPT}}$ is the minimum maximum workload among all workers.

We can verify that the load balancing problem is 2-approximable as follows. Observe that the greedy strategy in References [4, 31] is a 2-approximation for GLBP, and ParCover is a variant of the greedy strategy. From the reduction above it follows that ParCover generates a 2-approximation. Therefore, the load balancing problem is 2-approximable. □

Correctness. We show that algorithm ParCover is correct. Denote by Σ_c the output of algorithm ParDis. We show the following: (1) Σ_c is minimal; and (2) $\Sigma_c \equiv \Sigma$. From these and the proofs in Section 7.2 it follows that SeqDisGFD and DisGFD find equivalent sets of GFDs.

We first show that Σ_c is minimal as warranted by Lemma 8.1. Suppose by contradiction that there exists a GFD $\varphi \in \Sigma_c$ such that $\Sigma_c \setminus \{\varphi\} \models \varphi$. Let $\varphi = Q[\bar{x}](X \to l)$. Then by the characterization of GFD implication [27], there exists a group Σ' of GFDs in $\Sigma_c \setminus \{\varphi\}$ such that the pattern of each GFD in Σ' is embedded in Q and $\Sigma' \models \varphi$. Since each GFD in Σ' is embedded in Q, $\Sigma' \subseteq \Sigma_Q$. Then algorithm ParDis processes Σ^Q, finds φ redundant, and removes it. In other words, φ is not included in Σ_c by ParDis, a contradiction. Hence, Σ_c includes only non-redundant GFDs of Σ.

We now show that $\Sigma_c \equiv \Sigma$. Since $\Sigma_c \subseteq \Sigma$, obviously $\Sigma \models \Sigma_c$. For the other direction, assume that $\Sigma_c \not\models \Sigma$ by contradiction. Then we show that there must exist a GFD $\varphi = Q[\bar{x}](X \to l)$ in Σ such that $\Sigma_c \not\models \varphi$, and $\Sigma_c \models \Sigma_Q \setminus \Sigma^Q$. This suffices. Indeed, since $\varphi \notin \Sigma_c$, φ is removed by ParCover by using GFDs in $\Sigma_Q \setminus \Sigma^Q$ and non-redundant GFDs in Σ^Q. Because $\Sigma_c \models \Sigma_Q \setminus \Sigma^Q$ and $\Sigma_c \models (\Sigma_c \cap \Sigma^Q)$, we have that $\Sigma_c \models \varphi$, a contradiction. Therefore, $\Sigma_c \models \Sigma$.

We next show that if $\Sigma_c \not\models \Sigma$, then there exists $\varphi = Q[\bar{x}](X \to l)$ in Σ, such that $\Sigma_c \not\models \varphi$, and $\Sigma_c \models \Sigma_Q \setminus \Sigma^Q$. Denote by Σ_r the set of all GFDs ψ in Σ such that $\Sigma_c \not\models \psi$. Let φ be a GFD in Σ_r that carries the "smallest" pattern Q, i.e., for each $\psi \in \Sigma_r$, the pattern of ψ either is isomorphic to Q (hence $\psi \in \Sigma^Q$) or cannot be embedded in Q. As a result, no $\psi \in \Sigma_r$ is in $\Sigma_Q \setminus \Sigma^Q$. Therefore, $\Sigma_c \models \Sigma_Q \setminus \Sigma^Q$, since by the definition of Σ_r, all GFDs in $\Sigma_Q \setminus \Sigma^Q$ are entailed by Σ_c.

Parallel scalability. We next show that algorithm ParCover is parallel scalable. Suppose that $\Sigma = \{\varphi_1, \ldots, \varphi_M\}$. Recall that ParCover computes all *non-redundant* GFDs in Σ^{Q_i} at each worker in parallel, by "plugging" in a sequential algorithm ParImp, no matter what ParImp is chosen. By evenly balancing the workload, its cost is $\tilde{O}(\frac{T(\Sigma_{Q_1}, \varphi_1) + \cdots + T(\Sigma_{Q_m}, \varphi_M)}{n})$. Since $\Sigma_{Q_i} \subseteq \Sigma$, we have that $T(\Sigma_{Q_i}, \varphi_j) \leq T(\Sigma, \varphi_j)$. Then ParCover is in $O(\frac{t_2(\Sigma,k)}{n})$ time, where $t_2(\Sigma, k) = T(\Sigma, \varphi_1) + \cdots + T(\Sigma, \varphi_M)$ (see Section 6.3). The load balancing itself takes $O(|\Sigma| n \log n)$ time, which is much less than $O(\frac{t_2(\Sigma,k)}{n})$ in practice, since the latter is inherently exponential unless P = NP.

This and the analysis of ParDis (Section 7.2) show that algorithm DisGFD takes in total $O(\frac{t_1(|G|,k,\sigma)}{n}) + O(\frac{t_2(\Sigma,k)}{n})$ time, and is thus parallel scalable relative to its yardstick SeqDisGFD.

This completes the proof of Theorem 7.1.

9 EXPERIMENTAL STUDY

Using real-life and synthetic data, we conducted five sets of experiments to evaluate algorithm DisGFD for (1) the parallel scalability with the increase of workers used, (2) the scalability with graphs, (3) the impact of bound k, support threshold σ and active attributes Γ, (4) the parallel scalability of ParCover embedded in DisGFD, and (5) the effectiveness of finding useful GFDs.

Experimental setting. We used three real-life graphs: (a) DBpedia, a knowledge graph [2] with 1.72M entities of 200 types and 31M edges of 160 types, (b) YAGO2, an extended knowledge base

of YAGO [54] with 1.99M nodes of 13 types, and 5.65M edges of 36 types; and (c) IMDB [1], a knowledge base with 3.4M nodes of 15 types and 5.1M edges of 5 types. For *each entity*, we picked 5 active attributes from cleaned ontology (e.g., WordNet [54]). Each entity in YAGO2 and IMDB has at most 4 attributes, and 98% of nodes in DBpedia have at most 7 attributes; thus, 5 is reasonable. To strike a balance between the cost of GFD mining and the number of discovered GFDs, for each attribute $A \in \Gamma$, we picked 5 most frequent values from the active domain of A in the graphs, and used these values when constructing candidate GFDs. We adopted 5 as the threshold for the number of frequent attribute values, since for all attributes there exist on average 5 distinct values that have support of at least 500.

We also developed a generator for synthetic graphs $G = (V, E, L, F_A)$, controlled by the numbers $|V|$ of nodes (up to 30M) and $|E|$ of edges (up to 60M), where L is drawn from a set of 30 labels, F_A assigns a set Γ of 5 active attributes, and each $A \in \Gamma$ draws a value from 1,000 values.

GFD *generator*. To test the scalability of ParCover for GFD implication, we developed a generator to produce sets Σ of GFDs, controlled by $|\Sigma|$ (up to 10,000) and k (up to 6). It generates GFDs with frequent edges and values from real-life graphs, using the same attribute set Γ.

Algorithms. We implemented the following, all in Java: (1) sequential algorithm SeqDisGFD, including SeqDis and SeqCover; (2) algorithm DisGFD for parallel GFD mining, including ParDis and ParCover; (3) ParGFD$_n$, a version of DisGFD without GFD pruning (Lemma 6.5) for ParDis; (4) ParGFD$_{nb}$, DisGFD without load balancing (Proposition 8.3); and (5) ParCover$_n$, a version of ParCover without using GFD grouping (Lemma 8.1) and GFD generation trees.

We also implemented two other baselines. (1) Algorithm ParAMIE, the parallel algorithm to discover AMIE rules [12]. (2) Algorithm ParArab splits the pattern mining and dependency discovery phases. (a) It first discovers all frequent patterns Q in parallel, by using Arabasque [55], a state-of-the-art parallel graph pattern mining system. (b) It then extends each pattern Q to GFDs with literals, and verifies the latter in parallel. It uses the same procedure ParCover for implication.

In addition, we developed algorithm DisGCFD for mining GCFDs, an extension of relational CFDs [21] with path patterns [32], which makes a special case of GFDs.

We set the value of the support threshold σ such that the induced support of patterns is comparable to the counterparts used in frequent pattern mining [55]. For an application in practice, one picks σ to balance the complexity and interpretability of GFDs.

We deployed these algorithms on Amazon EC2 m4.xlarge instances; each is powered by an Intel Xeon processor with 2.3 GHz. We used up to 20 instances. Each experiment was run five times and the average is reported here.

Experimental results. We next report our findings. We took a synthetic graph G with 20M nodes and 40M edges as default. We fixed $k = 4$, $|\Gamma| = 150$, the support threshold $\sigma = 500$, and the number of processors $p = 8$ for parallel algorithms unless stated otherwise.

Infeasibility of ParGFD$_n$ *and* ParArab. Our first observation is that baseline algorithms ParGFD$_n$ and ParArab do not work well on large graphs. (1) Without effective pruning, ParGFD$_n$ fails to complete on all real-life graphs even when $n = 20$. It quickly consumes the available memory, due to a large number of GFD candidates. (2) Without integrated discovery, ParArab fails at the parallel verification step on real-life graphs when $n = 20$. The failures justify the need for our pruning strategy and the strategy to interleave vertical and horizontal expansions.

We hence report only the performance of the other algorithms.

Exp-1: Parallel scalability. We first evaluated the parallel scalability of DisGFD by varying the number n of workers from 4 to 20, compared with ParGFD$_{nb}$. The results are reported in

Figures 5(a)–5(d) over DBpedia, YAGO2, IMDB, and Synthetic G, respectively. A cost breakdown demonstrates that parallel pattern verification and GFD validation dominate the cost. Nonetheless, the parallel costs are reduced when more workers are used.

As shown in Figure 5(a) on DBpedia, (1) DisGFD is parallel scalable. It is 3.6 times faster on average when the number n of processors increases from 4 to 20. Moreover, when increasing n from 4 to 8, the runtime decreases more dramatically than the other cases, since the running time of DisGFD is "inversely proportional" to n as shown in Theorem 7.1. (2) DisGFD outperforms ParGFD$_{nb}$ by 1.5 times on average, and in particular, by 2.2 times when $n = 20$. This verifies the effectiveness of our load balancing strategy. (3) DisGFD is feasible in practice: it takes 30 min when $n = 20$.

The results in Figures 5(b), 5(c), and 5(d) are consistent. (1) DisGFD is 4, 3.8, and 3.4 times faster on YAGO2, IMDB and Synthetic G, respectively, when n varies from 4 to 20. (2) It outperforms ParGFD$_{nb}$ by 1.2, 1.3, and 1.3 times on the three graphs, respectively. Load balancing works better on DBpedia, since it is denser than YAGO2, IMDB and Synthetic G, and yields more patterns.

We also compared algorithm DisGFD with DisGCFD and ParAMIE on YAGO2. We set bound $k = 3$ for GFDs and GCFDs, which is the default size of the variable set for each AMIE rule [12]. Figure 5(e) tells us the following. (1) DisGFD is comparable to DisGCFD in efficiency, although it finds more GFDs with general patterns than GCFDs found by DisGCFD. (2) Although GFDs are more expressive than AMIE rules, DisGFD still outperforms ParAMIE by 3.4 times on average, due to its pruning strategies. The results on the other graphs are consistent.

Sequential cost. Figure 7 reports the cost of sequential algorithm SeqDisGFD. While ParGFD$_n$ and ParArab fail to complete, SeqDisGFD performs reasonably well: it takes 1.3 h to mine GFDs from YAGO2 of 7.64M entities and edges. The results on DBpedia and IMDB are consistent.

We report the number and average support of mined rules on three real-life knowledge-base datasets in Figure 7. While GFDs discovered have to be satisfied by the graphs, AMIE rules are soft constraints (when confidence is less than 1) that are not necessarily satisfied by the datasets. We set the PCA (partial completeness assumption) confidence [29] threshold of AMIE as 0.5. We found that (1) most of the mined graph patterns are acyclic (96%, 99%, and 97% for DBPedia, YAGO2, and IMDB, respectively); (2) the discovered GFDs can express all the AMIE rules with PCA confidence 1; and (3) most AMIE rules cannot capture inconsistencies via constant bindings (see Exp-5).

Exp-2: Scalability with $|G|$. Fixing $n = 20$, $\sigma = 500$ and $k = 4$, we evaluated the scalability of algorithm DisGFD by varying the size of synthetic graph $|G| = (|V|, |E|)$ from (10M, 20M) (denoted by G_1) to (30M, 60M) (denoted by G_5). As shown in Figure 5(f), (1) the algorithm takes longer to discover GFDs from larger graphs, as expected; and (2) GFD discovery is feasible in large-scale graphs. DisGFD takes less than 30 min to discover GFDs in graph G of size (30M, 60M).

As indicated in Figure 7, the impact of $|G|$ on sequential algorithm SeqDisGFD is consistent: the larger graph G is, the longer SeqDisGFD takes to mine GFDs in G.

Exp-3: Impact of parameters. We evaluated the impact of pattern size k, support threshold σ and active attributes Γ on the performance of the algorithms. We fixed $n = 8$ in this set of experiments.

Varying k. Fixing $\sigma = 500$ and $|\Gamma| = 150$, we varied k from 2 to 6, and report the results on DBpedia, YAGO2, IMDB and Synthetic G in Figures 5(g)–5(j), respectively. We find the following. (1) It takes both DisGFD and ParGFD$_{nb}$ longer to find GFDs with larger patterns, as expected; (2) DisGFD outperforms ParGFD$_{nb}$ by 1.32, 1.28, 1.26, and 1.24 times on the four graphs, respectively. (3) It is feasible for DisGFD to discover GFDs with reasonably large graph patterns: it takes 76, 14, 22 and 60 min to find all 5-bounded GFDs on DBpedia, YAGO2, IMDB, and Synthetic G, respectively.

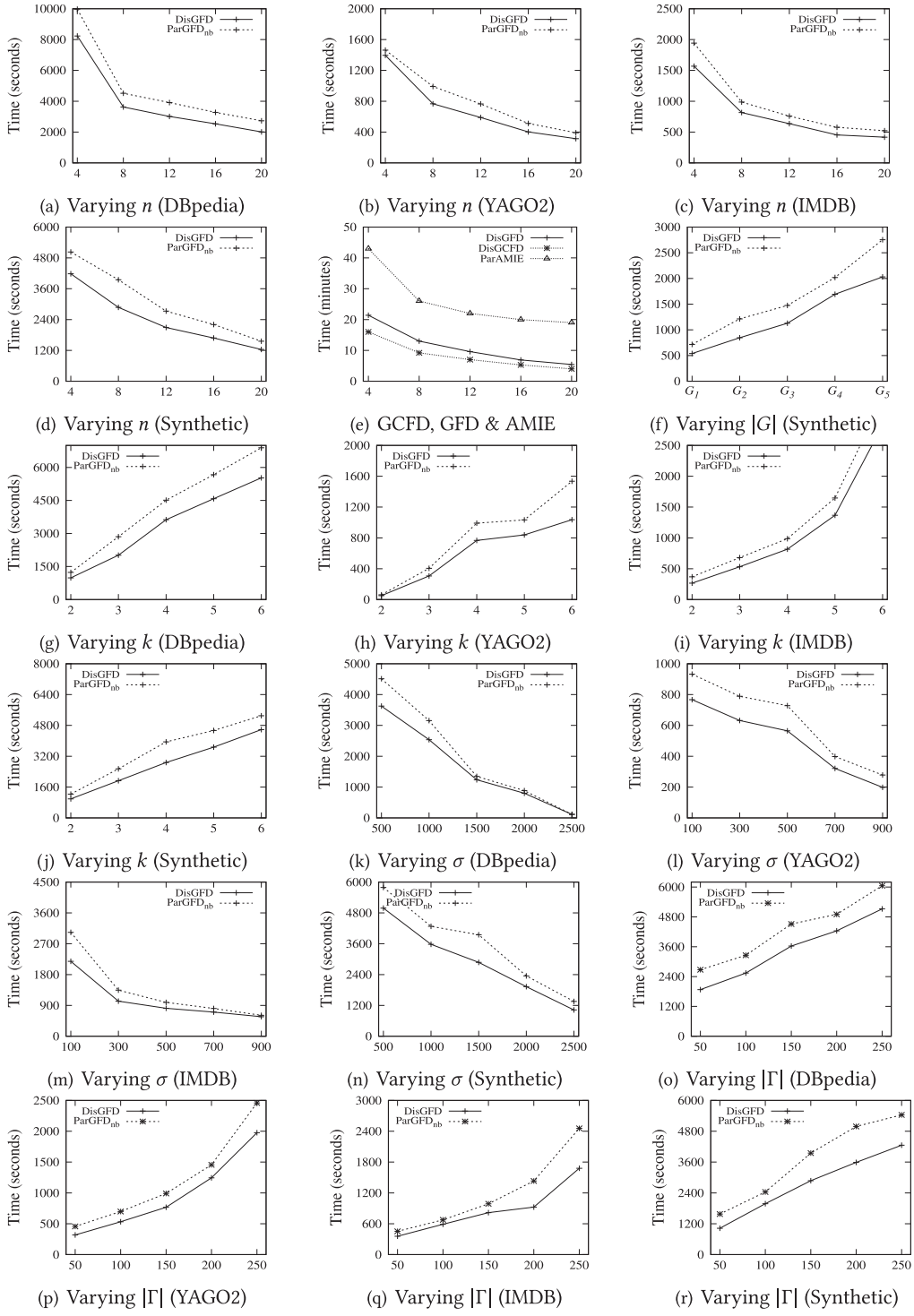
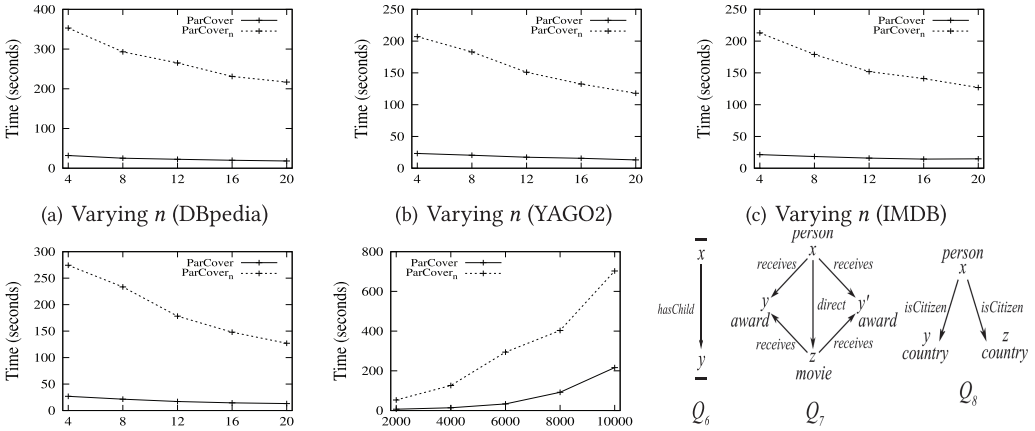


Fig. 5. Performance evaluation of parallel GFD discovery.



(a) Varying n (DBpedia) (b) Varying n (YAGO2) (c) Varying n (IMDB)

(d) Varying n (Synthetic) (e) Varying $|\Sigma|$ (Synthetic) (f) Real-life GFDs (YAGO2)

Fig. 6. Performance evaluation of parallel GFD cover computation.

dataset	SeqDisGFD	SeqCover	**GFDs**	**GCFDs**	**AMIE**
DBpedia	24387s	45s	321/1724	202/1578	481/1500
YAGO2	4963s	32.5s	145/605	104/698	69/600
IMDB	5896s	36.8s	206/500	167/500	115/500

Fig. 7. Sequential cost and rule #/avg. support.

Varying σ. Fixing $k = 4$ and $|\Gamma| = 150$, we varied σ from 500 to 2,500, and evaluated DisGFD and ParGFD$_n$. As shown in Figures 5(k)–5(n) on the four graphs, both algorithms take less time with larger σ. This is because higher σ prunes more GFD candidates, and reduces both GFD generation and verification time. This again verifies the effectiveness of our pruning strategy.

Varying $|\Gamma|$. Fixing $k = 4$ and $\sigma = 500$, we varied $|\Gamma|$ from 50 to 250. As reported in Figures 5(o)–5(r) on the four graphs, both algorithms take longer with larger $|\Gamma|$, as more GFD candidates are generated.

Exp-4: GFD cover computation. In the same setting as Figures 5(a)–5(c), we report the scalability of ParCover on DBpedia, YAGO2, IMDB, and Synthetic G in Figures 6(a)–6(d), respectively, compared with ParCover$_n$. On average, (1) the performance of ParCover is improved by 1.75 times when n is increased from 4 to 20 on real-life graphs; and (2) it outperforms ParCover$_n$ by 10 times. This validates the effectiveness of GFD generation tree, grouping and load balancing strategies.

As shown in Figure 7, sequential algorithm SeqCover also does well. It takes at most 45.1 s over the three real-life datasets to compute the cover of GFDs discovered. *Varying* $|\Sigma|$. Fixing n=4, we evaluated ParCover by varying the number of GFDs from 2,000 to 10,000. As shown in Figure 6(e), ParCover takes longer when $|\Sigma|$ is larger, as expected. It is less sensitive to $|\Sigma|$ than ParCover$_n$, since tree T and grouping and load balancing mitigate the impact of $|\Sigma|$ in the parallel implication. The impact of $|\Sigma|$ on sequential SeqCover is consistent, as indicated by Figure 7.

Exp-5: Effectiveness. Finally, we validated the GFDs discovered.

Error detection accuracy. We make a comparison between AMIE and GFDs in terms of the accuracy of detecting errors introduced to YAGO2. We discovered a set Σ of GFDs and a set Σ^A of AMIE rules from YAGO2. We then introduced noise to YAGO2: We randomly drew α% of nodes and for

$(\sigma, k, \lvert\Gamma\rvert)$ (YAGO2)	GFDs	GCFDs	AMIE
(500,3,150)	74.3%	63.5%	68.7%
(1000,3,150)	67.5%	56.4%	62.8%
(1000,4,150)	71.4%	60.5%	64.2%
(1000,4,200)	73.2%	62.18%	64.2%

Fig. 8. Error detection accuracy.

each such node v, changed $\beta\%$ of either the active attribute values or the labels of edges of v (to favor AMIE, which does not support wildcard), with values that did not appear in YAGO2. We took care to make changes that involve the consequence l of $X \rightarrow l$ in Σ and Σ^A discovered. For GFDs, we apply the methods of Reference [27] to validate the mined GFDs in the graph.

The *accuracy* of AMIE rules (respectively, GFDs) is defined as $\frac{|V^A \cap V^E|}{|V^E|}$ (respectively, $\frac{|V^{\text{GFD}} \cap V^E|}{|V^E|}$), where (a) V^E is the set of all the nodes with introduced noise; and (b) V^A for AMIE (respectively, V^{GFD} for GFDs) refers to the set of all the nodes that do not have the predicted relation (respectively, contained in the violations of GFDs). The accuracy of GCFDs is defined similarly.

The accuracy of GFDs, GCFDs and AMIE and the impact of σ, k and Γ are reported in Figure 8. We selected σ based on the frequency of edge labels to favor AIME, which does not support wildcard on edges. We find the following. (1) GFDs achieve the best accuracy among all rules. (2) GFDs have better accuracy if discovered with smaller σ, larger k and larger $|\Gamma|$, since more GFDs are discovered to "cover" inconsistencies. (3) The accuracy of AMIE is close to that of GFDs, since more AIME rules may be mined (their PCA confidence is not "1" and the AMIE rules may have conflicts).

Real-world GFDs. We also manually inspected the GFDs and validated their usefulness. As examples, we give three GFDs found in YAGO2 by DisGFD, with patterns shown in Figure 6(f).

GFD_1: $Q_6[x, y]$ ($\emptyset \rightarrow x.\text{familyname} = y.\text{familyname}$) is a "variable-only" GFD, where x and y are labeled wildcard. The rule states that a child inherits the family name of his/her parent.

GFD_2: $Q_7[x, y, z, y']$($y.\text{name} =$ "Gold Bear" $\wedge$ $y'.\text{name} =$ "Gold Lion" $\rightarrow$ false). It tells us that no movie receives both awards. To interpret this, we looked into the Italian and German film festivals and found that both require their participants to be "initial release" at their festivals.

GFD_3: $Q_8[x, y, z](X_8 \rightarrow \text{false})$, a negative GFD, where X_8 consists of $y.\text{name} =$ "US" and $z.\text{name} =$ "Norway." The GFD states that Norway does not admit dual citizenship. It is obtained by expanding a positive $Q_8[x, y, z](y.\text{name} =$ "US" $\rightarrow x.\text{livesIn} =$ "US") by adding a literal. The closest AMIE rule to GFD_3 is $\text{lives}(x, y) \rightarrow \text{isCitizen}(x, y)$, which cannot identify errors captured by GFD_3.

These GFDs carry a DAG pattern, constants, wildcard "_" or false, beyond the capacity of most graph FD proposals and AMIE rules [12, 29]. They help us detect errors and extract knowledge, e.g., GFD_3 reveals that Norway does not admit dual citizenship, a fact not familiar to some people.

Algorithm ParDis is also able to help us verify and extend a set Σ of GFDs provided by users. More specifically, it can initialize the GFD generation tree T with patterns and attributes specified in Σ, starts discovery with T, and returns GFDs that are beyond a support threshold set by the users. This would reduce the cost of GFD discovery starting from scratch.

Summary. We find the following. (1) GFD discovery is feasible in practice. It takes 913 s for DisGFD to find frequent 4-bounded GFDs from real-life graphs on average when $p = 20$. (2) GFD discovery is parallel scalable: algorithm DisGFD (respectively, ParCover) is 3.78 (respectively, 1.75) times faster on average when n is increased from 4 to 20. The optimization strategies in ParCover

further improve the performance of ParCover$_n$ by 10 times. (3) Our integrated method for vertical and horizontal expansions and strategies for pruning and load balancing are effective. While ParArab and ParGFD$_n$ fail to complete GFD discovery, algorithm DisGFD works well on real-life graphs, and outperforms ParGFD$_{nb}$ by 1.31 times on average. (3) GFDs discovered from real-life graphs by DisGFD can catch data inconsistencies with higher accuracy when compared with AMIE rules and GCFDs. In addition, GFDs can also suggest knowledge enrichment with interesting new facts.

10 CONCLUSION

We have formalized and studied the discovery problem for GFDs. The novelty of the work consists of (1) the fixed-parameterized complexity of three classical problems underlying GFD discovery, (2) a notion of support for GFDs on graphs, (3) sequential and parallel algorithms for discovering GFDs and computing a cover of GFDs, in particular, the parallel algorithms guarantee the parallel scalability, and (4) new techniques for spawning and validating GFDs. Our experimental results have verified that our algorithms scale with large graphs and are able to discover interesting GFDs.

We are currently extending algorithm DisGFD to discover other forms of graph dependencies, e.g., GFDs with built-in comparison predicates and arithmetic expressions [24]. Another topic is to adapt the algorithm to knowledge bases, adopting the support and confidence of Reference [48].

REFERENCES

[1] IMDB. 2008. IMDB. Retrieved from http://www.imdb.com/interfaces.

[2] DBpedia. 2015. DBpedia. Retrieved from http://wiki.dbpedia.org/Datasets.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Gagan Aggarwal, Rajeev Motwani, and An Zhu. 2003. The load rebalancing problem. In *Proceedings of the SPAA*.

[5] Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. 2010. Constraints in RDF. In *Proceedings of the SDKB*. 23–39.

[6] Khaled Ammar and M. Tamer Özsu. 2018. Experimental analysis of distributed graph systems. *Proc. Very Large Data Base* 11, 10 (2018), 1151–1164.

[7] Anonymous. 2018. Details omitted due to double-blind reviewing.

[8] Ismailcem Budak Arpinar, Karthikeyan Giriloganathan, and Boanerges Aleman-Meza. 2006. Ontology quality by detection of conflicts in metadata. In *Proceedings of the EON*.

[9] Vladimir Batagelj and Andrej Mrvar. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Soc. Netw.* 23, 3 (2001), 237–243.

[10] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An analytical study of large SPARQL query logs. *Proc. Very Large Data Base* 11, 2 (2017), 149–161.

[11] Diego Calvanese, Wolfgang Fischl, Reinhard Pichler, Emanuel Sallinger, and Mantas Simkus. 2014. Capturing relational schemas and functional dependencies in RDFS. In *Proceedings of the AAAI*.

[12] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. 2016. Ontological pathfinding. In *Proceedings of the SIGMOD*. 835–846.

[13] Fei Chiang and Renee Miller. 2008. Discovering data quality rules. In *Proceedings of the VLDB*.

[14] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proc. Very Large Data Base* 6, 13 (2013), 1498–1509.

[15] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of constraints in RDFS. In *Proceedings of the AMW*. 75–90.

[16] Rodney G. Downey and Michael R. Fellows. 1995. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theor. Comput. Sci.* 141, 1&2 (1995), 109–131.

[17] Rodney G. Downey and Michael R. Fellows. 2013. *Fundamentals of Parameterized Complexity*. Springer.

[18] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GRAMI: Frequent subgraph and pattern mining in a single large graph. *Proc. Very Large Data Base* 7, 7 (2014), 517–528.

[19] David Eppstein. 1995. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the SODA*, Vol. 95. 632–640.

[20] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for graphs. *Proc. Very Large Data Base* 8, 12 (2015), 1590–1601.

[21] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *Trans. Database Syst.* 33, 1 (2008).
[22] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.* 23, 5 (2011), 683–698.
[23] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *Proceedings of the ICDE*. 39–50.
[24] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2018. Catching numeric inconsistencies in graphs. In *Proceedings of the SIGMOD*.
[25] Wenfei Fan and Ping Lu. 2017. Dependencies for graphs. In *Proceedings of the PODS*.
[26] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association rules with graph patterns. *Proc. Very Large Data Base* 8, 12 (2015), 1502–1513.
[27] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *Proceedings of the SIGMOD*.
[28] Jörg Flum and Martin Grohe. 2006. *Parameterized Complexity Theory*. Springer.
[29] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. 2013. AMIE: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the WWW*.
[30] Mario Arias Gallego, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. 2011. An empirical study of real-world SPARQL queries. In *USEWOD Workshop*.
[31] R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell Syst. Tech. J.* 45, 9 (1966), 1563–1581.
[32] Binbin He, Lei Zou, and Dongyan Zhao. 2014. Using conditional functional dependency to discover abnormal data in RDF graphs. In *Proceedings of the SWIM*. 1–7.
[33] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. 2014. Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. In *Proceedings of the FoIKS*.
[34] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. 2004. Spin: Mining maximal frequent subgraphs from graph databases. In *Proceedings of the SIGKDD*.
[35] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.* 42, 2 (1999), 100–111.
[36] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An Apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the PKDD*.
[37] Takehiro Ito, Marcin Kamiński, Hirotaka Ono, Akira Suzuki, Ryuhei Uehara, and Katsuhisa Yamanaka. 2014. On the parameterized complexity for token jumping on graphs. In *Proceedings of the TAMC*. Springer, 341–351.
[38] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Rev.* 28, 01 (2013), 75–105.
[39] Yiping Ke, James Cheng, and Jeffrey Xu Yu. 2009. Efficient discovery of frequent correlated subgraph pairs. In *Proceedings of the ICDM*.
[40] Mijung Kim and K. Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowledge Eng.* 72 (2012), 285–303.
[41] Myung-Sup Kim, Hun-Jeong Kong, Seong-Cheol Hong, Seung-Hwa Chung, and James W. Hong. 2004. A flow-based method for abnormal network traffic detection. In *Proceedings of the NOMS*, Vol. 1. IEEE, 599–612.
[42] Christian Komusiewicz and Manuel Sorge. 2012. Finding dense subgraphs of sparse graphs. In *Proceedings of the IPEC*. Springer, 242–251.
[43] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. 2014. Test-driven evaluation of linked data quality. In *Proceedings of the WWW*. 747–758.
[44] Richard E. Korf. 2009. Multi-Way Number Partitioning. In *Proceedings of the IJCAI*.
[45] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *Trans. Comput. Sci.* 71, 1 (1990), 95–132.
[46] Georg Lausen, Michael Meier, and Michael Schmidt. 2008. SPARQLing constraints for RDF. In *Proceedings of the EDBT*. ACM, 499–509.
[47] Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. 2014. Large-scale frequent subgraph mining in MapReduce. In *Proceedings of the ICDE*.
[48] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. 2015. Yago3: A knowledge base from multilingual wikipedias. In *Proceedings of the CIDR*.
[49] Dániel Marx. 2006. Parameterized graph separation problems. *Theoret. Comput. Sci.* 351, 3 (2006), 394–406.
[50] Luke Mathieson and Stefan Szeider. 2008. Parameterized graph editing with chosen vertex degrees. In *Proceedings of the COCOA*. Springer, 13–22.
[51] Linsey Xiaolin Pang, Sanjay Chawla, Wei Liu, and Yu Zheng. 2011. On mining anomalous patterns in road traffic streams. In *Proceedings of the ADMA*. Springer, 237–251.
[52] François Picalausa and Stijn Vansummeren. 2011. What are real SPARQL queries like? In *Proceedings of the SWIM*.

[53] Prakash Shelokar, Arnaud Quirin, and Óscar Cordón. 2014. Three-objective subgraph mining using multiobjective evolutionary programming. *J. Comput. Syst. Sci.* 80, 1 (2014), 16–26.
[54] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A core of semantic knowledge. In *Proceedings of the WWW*. 697–706.
[55] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of the SOSP*. 425–440.
[56] David P. Woodruff and Qin Zhang. 2013. When distributed computation is communication expensive. In *Proceedings of the DISC*.
[57] Catharine Wyss, Chris Giannella, and Edward Robertson. 2001. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Proceedings of the DaWaK*.
[58] Yang Yu and Jeff Heflin. 2011. Extending functional dependency to detect abnormal data in RDF graphs. In *Proceedings of the ISWC*. 794–809.
[59] Amrapali Zaveri, Dimitris Kontokostas, Mohamed Ahmed Sherif, Lorenz Bühmann, Mohamed Morsey, Sören Auer, and Jens Lehmann. 2013. User-driven quality evaluation of DBpedia. In *Proceedings of the ISEM*. 97–104.

Received March 2019; revised February 2020; accepted April 2020