Discovering Graph Functional Dependencies

Wenfei Fan[1,2] Chunming Hu[2] Xueli Liu[3] Ping Lu[2]
[1]University of Edinburgh [2]BDBC, Beihang University [3]Harbin Institute of Technology
wenfei@inf.ed.ac.uk, xueli.hit@gmail.com, {hucm, luping}@buaa.edu.cn

ABSTRACT

This paper studies discovery of 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. 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 for discovering GFDs 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.

KEYWORDS

GFD discovery, parallel scalable, fixed-parameter tractability

ACM Reference Format:
Wenfei Fan, Chunming Hu, Xueli Liu, Ping Lu. 2018. Discovering Graph Functional Dependencies. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3196916

1 INTRODUCTION

Functional dependencies have recently been studied for property graphs [18, 20], referred to as *graph functional dependencies* (GFDs). Unlike relational databases, real-life graphs often do not come with a schema. On such graphs, GFDs provide a primitive form of integrity constraints to specify a fundamental part of the semantics of the 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: 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 Fig. 1.

(a) YAGO3 [36]: A person John Winter is given credit for producing film Selling Out (G_1 in Fig. 1). But John is a high jumper. In fact, the film was created by producer Jack Winter.

(b) YAGO3: Saint Petersburg is located in two places, as a city in both Russia and Florida (G_2 in Fig. 1).

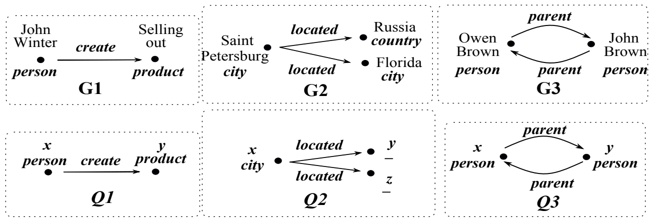


Figure 1: Graphs and graph patterns

(c) DBpedia [1]: John Brown and Owen Brown are claimed to be a parent of each other (G_3 in Fig. 1).

GFDs of [18, 20] 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 Fig. 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 $\varphi_2 = Q_2[x,y,z](\emptyset \rightarrow y.\text{name} = z.\text{name})$, where pattern Q_2 is shown in Fig. 1, $\emptyset$ denotes an empty set of literals, and name is an attribute. It says that if city x is located in y and z, then y and z must be the same place; *i.e.,* a city can be located in only one place. It catches the error in G_2. Note that nodes y and z are labeled with wildcard '_', which can match, *e.g.,* country and city.

(3) Consider GFD $\varphi_3 = Q_3[x,y](\emptyset \rightarrow \text{false})$, where Q_3 is depicted in Fig. 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.,* Q_3 specifies an "illegal" structure. It catches the error in 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, φ_2), or *negative* (specifying "illegal" cases with false, *e.g.,* φ_3). GFD discovery is much harder than discovering relational FDs [27, 41], as GFDs additionally require topological constraints Q. It is more challenging than graph pattern mining [13, 19, 26, 28, 35, 37], 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 [20], which are embedded in GFD discovery.

Contributions. This paper tackles these challenges.

(1) We investigate three fundamental problems related to GFD discovery (Section 3). The satisfiability problem is to determine whether GFDs discovered are not "dirty", *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 mined from a graph G are satisfied by G.

We show that while the implication and satisfiability problems are fixed-parameter tractable [21], the validation problem is co-W[1]-hard [11]. 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 4). 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. 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 a sequential algorithm for discovering GFDs of [20] (Section 5). 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 develop a parallel algorithm for discovering GFDs in fragmented graphs (Section 6). We employ distributed incremental joins to balance the workload. We show that the algorithm is parallel scalable [33] 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.

(5) Using real-life and synthetic graphs, we experimentally evaluate the algorithms (Section 7). We find the following. (a) GFD discovery is parallel scalable. It is on average 3.78 times faster on real-life graphs when processors n increase from 4 to 20. (b) GFD discovery is feasible in practice. The sequential GFD mining algorithm takes 1.3 hours on *YAGO2* with 7.64 millions of entities and edges. The performance is substantially improved by parallelization. When n = 20, it takes 591 seconds on average on real-life graphs (314 seconds on *YAGO2*), and 30 minutes on synthetic graphs with 30M nodes and 60M edges. (c) Computing GFD cover is also 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.

Related work. We categorize related work as follows.

FDs *for graphs*. FDs have been studied for RDF [5–7, 10, 22, 24, 25, 34, 42]. A direct extension of FDs to relational encoding was studied in [34]. Based on triple patterns with variables, [5, 10] 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 [10, 25] via relational encoding of RDF. Using clustered values, [42] defines FDs with path patterns; [42] is extended to support CFDs (conditional functional dependencies [15]) for RDF [24]. FDs are also defined in [6], using tree patterns. AMIE [7, 22] extends association rules with conjunctive horn clauses for knowledge graph enhancement.

This work adopts GFDs of [20] for the following reasons. (a) GFDs are defined for general property graphs, not limited to RDF. (b) GFDs support (cyclic) graph patterns with variables, *e.g.,* φ_3 of Example 1, as opposed to [6, 24, 42]. (c) GFDs support bindings of semantically related values like CFDs [15], *e.g.,* φ_1, and a negative form with false, *e.g.,* φ_3, which cannot be expressed as the FDs of [5–7, 10, 22, 24]. The need for supporting these is evident in consistency checking, as indicated by axioms for knowledge bases (*e.g.,* [32]), and by the experience of cleaning relational data [15].

Note that neither [20] nor [18] considers GFD discovery.

In contrast to GFDs, AMIE supports neither pattern matching via subgraph isomorphism nor constant-value binding. Moreover, it cannot express negative rules and rules with wildcard.

Dependency discovery. Discovery algorithms have been well studied for relational dependencies, *e.g.,* FDs [27, 41], CFDs [8, 16] and denial constraints [9]. As remarked earlier, GFD discovery is much harder. Closer to this work are algorithms for discovering FDs over graphs [24, 42]. The method of [42] first pre-clusters property values; it then adapts the levelwise process of TANE [27] to discover FDs defined with path patterns over RDF. It is extended in [24], which first enumerates frequent graph structures, and then adopts CFDMiner [16] 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 [24, 42], (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 [32], (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, for domain experts to inspect and select before they are used as 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 [26, 28, 30]. Apriori [28] and pattern-growth [26] methods expand a frequent pattern by adding nodes and edges. [30] connects two graphs via Pearson correlation. Multi-objective subgraph mining [37] optimizes subgraphs via skyline processing. As observed in [29], pattern mining over graph databases does not help GFD discovery, since the anti-monotonicity of the support of [26, 28, 30] no longer holds over a single graph.

Closer to this work are mining techniques for a single graph [13, 19, 35, 39]. GRAMI [13] considers patterns without edge labels, and models isomorphic subgraph enumeration as constraint satisfaction. The method of [35] mines frequent subgraphs via a two-step filter-and-refinement process, in MapReduce. Arabesque [39] uses "pattern-centric" MapReduce programming in pattern mining. The method of [19] mines top-k diversified association rules $Q \Rightarrow p$ defined with a graph pattern Q and a single edge p.

GFD discovery differs from the prior work in the following. (a) It requires both graph pattern mining and FD mining. We develop new data structures and techniques to combine the two in a single process. (b) No prior work has studied "negative patterns" coupled with FDs. (c) To find a cover of 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 [19, 20]. Our algorithms differ from [19, 20] in problem statements and methods. We balance the workload via distributed and incremental joins, while [19, 20] require special treatments for skewed graphs.

2 GRAPH FUNCTIONAL DEPENDENCIES

We first review GFDs [20], starting with basic notations.

2.1 Preliminaries

Assume an alphabet Θ of the node and edge labels in graphs. We consider directed graphs $G = (V, E, L, F_A)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$, in which (v, v') is an edge from node v to v'; (3) each $v \in V$ is labeled $L(v) \in \Theta$; each $e \in E$ is labeled $L(e) \in \Theta$; and (4) 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, A_i is an *attribute* of v, written as $v.A_i = a_i$; and $A_i \neq A_j$ if $i \neq j$. The attributes carry its content as in property graphs, social networks and knowledge bases.

We use two notions of subgraphs.

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

Graph patterns. A *graph pattern* is a graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (resp. E_Q) is a set of pattern nodes (resp. edges); (2) L_Q is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) to each node $u \in V_Q$ (resp. edge $e \in E_Q$); we allow $L_Q(u)$ and $L_Q(e)$ to be labeled with wildcard '_'; (3) $\bar{x}$ is a list of variables, 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.

Example 2: Figure 1 shows three graph patterns: (1) Q_1 depicts a person connected to a product with an edge labeled create; here μ maps x to person and y to product; (2) Q_2 shows a city x located in y and z labeled '_'; and (3) Q_3 is a pattern of person entities. □

Pattern matching. 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 $\ell = \ell'$. Intuitively, a wildcard '_' indicates generic entities or properties, and hence may map to any label in Θ.

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 V_Q to V' 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 (*iff*) $e' = (h(u), h(u'))$ is an edge in G' and $L'(e') \preceq L_Q(e)$.

We also denote the match as a vector $h(\bar{x})$, consisting of $h(x)$ (*i.e.,* $h(\mu(x))$) for all $x \in \bar{x}$, in the same order as $\bar{x}$.

For instance, a match h_2 of pattern Q_2 in G_2 of Fig. 1 is $x \mapsto$ Saint Petersburg, $y \mapsto$ Russia and $z \mapsto$ Florida.

2.2 Functional Dependencies for Graphs

A *graph functional dependency* (GFD) is $Q[\bar{x}](X \to Y)$ [20], where

- $Q[\bar{x}]$ is a graph pattern, called 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 (not specified in Q), and c is a constant. Intuitively, φ is a combination of two constraints:

- a *topological constraint* imposed by pattern Q, and
- *attribute dependency* specified by $X \to Y$.

Here Q specifies the scope of φ such that $X \to Y$ is imposed only on matches of Q. Literals $x.A = c$ enforce constant bindings like CFDs [15]. 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 shows GFDs φ_1, φ_2 and φ_3.

Semantics. Consider a match $h(\bar{x})$ of Q in a graph G, and a literal $x.A = c$. We say that $h(\bar{x})$ *satisfies* the literal if *there exists* attribute A at the node $v = h(x)$ and $v.A = c$; 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 $h(\bar{x}) \models X \to Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$.

A graph G *satisfies* GFD φ, denoted by $G \models \varphi$, if *for all* matches $h(\bar{x})$ of Q in G, $h(\bar{x}) \models X \to Y$. 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,

1. for $x.A = c$ in X, if $h(x)$ has *no* attribute A, then $h(\bar{x})$ satisfies $X \to Y$. Indeed, node $h(x)$ is not required to have attribute A since graphs have no schema. 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 $x.A = y.B$.
2. When X is $\emptyset$, $h(\bar{x}) \models X$ 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* $X \to Y$, *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.

Positive and negative. A GFD is called *negative* if it has the form $Q[\bar{x}](X \to \text{false})$ and X is satisfiable, *i.e.,* there exist graph G and a match $h(\bar{x})$ of Q in G such that $h(\bar{x}) \models X$. It is *positive* otherwise.

There are two cases of negative GFDs φ.

(a) When $X = \emptyset$, *i.e.,* φ has the form $Q[\bar{x}](\emptyset \to \text{false})$; 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 Fig. 1.

(b) When $X \neq \emptyset$, it states that the combination of pattern Q and condition X is "inconsistent".

Example 3: In Fig. 1, $G_2 \not\models \varphi_2$. Indeed, a match of Q_2 in G_2 is h_2 $x \mapsto$ Saint Petersburg, $y \mapsto$ Russia and $z \mapsto$ Florida. Here X in φ_2 is trivially true ($\emptyset$) but y.name $\neq$ z.name (Russia vs. Florida). 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 are positive. □

Normal form. We consider *w.l.o.g.* positive GFDs of the form $\varphi = Q[\bar{x}](X \to l)$, where l is a literal, *i.e.,* Y in φ has a single l. Note that this does not lose generality, a positive GFD $Q[\bar{x}](X \to Y)$ is equivalent to a set of GFDs $Q[\bar{x}](X \to l)$ 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 for positive GFDs, we consider the normal form.

3 FIXED PARAMETER TRACTABILITY

We next revisit three fundamental problems for GFDs.

(1) A set Σ of GFDs is *satisfiable* if there exists a graph G such that (a) $G \models \Sigma$, and (b) there exists a GFD $Q[\bar{x}](X \rightarrow Y)$ in Σ such that Q has a match in G. Intuitively, condition (b) ensures that at least one of the GFDs can be applied to nonempty graphs.

The *satisfiability problem* for GFDs is to decide whether a given set Σ of GFDs is satisfiable.

It is to check, *e.g.,* whether discovered GFDs are meaningful.

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

The implication analysis is necessary for computing a cover of discovered GFDs, and eliminating redundant GFDs.

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

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 is satisfied by the entire G.

Fixed-parameter tractability. It is shown that the satisfiability, implication and validation problems for GFDs are coNP-complete, NP-complete and coNP-complete, respectively [20].

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 conventional complexity theory, and k is a parameter that characterizes the structure of x. It is called *fixed-parameter tractable* 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.,* [11] for details). Intuitively, if k is small, then it is feasible to solve the problem efficiently, although $f(k)$ could be exponential, *e.g.,* 2^k.

In practice, 97.25% of SWDF patterns and 66.41% of DBPedia queries consist of only one single-triple patterns [23]. Therefore, it is practical to study the parameterized problems for GFDs.

For a set Σ of GFDs, we use k to denote $\max(|\bar{x}|)$ for all $Q[\bar{x}](X \rightarrow Y)$ 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: $k = \max\{|\bar{x}| \mid Q[\bar{x}](X \rightarrow Y) \in \Sigma \cup \{\varphi\}\}$.
- Question: Does $\Sigma \models \varphi$?

Similarly, we parameterize the other problems by k.

Theorem 1: *For* GFDs*, (a) the implication and satisfiability problems are fixed-parameter tractable with parameter k. However, (b) the validation problem is co-*W[1]*-hard even with parameters k and d, where d denotes the maximum degree of the nodes in graph G.* □

Here W[1] is the class of parameterized problems that are FPT-reducible to a weighted satisfiability problem (see [11]). It is conjectured that W[1]-hard problems are not fixed-parameter tractable. Thus the validation analysis remains nontrivial when k is small.

To prove Theorem 1, we first review characterizations of GFD satisfiability and implication (Lemmas 3 and 7 of [20]).

Characterization. We start with some notations. A GFD $\varphi' = Q'[\bar{x}'](X \rightarrow Y)$ is *embedded in a pattern* Q if there exists an isomorphism from pattern Q' of φ' to a subgraph of Q.

The set $\Sigma_Q \subseteq \Sigma$ of GFDs *embedded in* Q consists of all GFDs $\varphi' \in \Sigma$ that are embedded in Q.

For a set X of literals, closure(Σ_Q, X) is the set of literals deduced by applying Σ_Q to Q and by the transitivity of equality in X. We refer to closure(Σ_Q, X) as enforced(Σ_Q) when X is empty.

We say that closure(Σ_Q, X) is *conflicting* if it contains $x.A = c$ and $x.A = d$ for $c \neq d$, *i.e.,* for all $G \models \Sigma$, X is not satisfiable.

The characterizations are given as follows [20].

A set Σ of GFDs is satisfiable *if and only if* for all patterns Q that appear in Σ, enforced(Σ_Q) is not conflicting.

For a GFD $\varphi = Q[\bar{x}](X \rightarrow l)$, $\Sigma \models \varphi$ *if and only if* either closure(Σ_Q, X) is conflicting or $l \in$ closure(Σ_Q, X).

Proof: (a) Based on the characterization, we give an algorithm for the satisfiability analysis of GFDs as follows: (1) compute the set Σ_Q of GFDs embedded in Q for each GFD $Q[\bar{x}](X \rightarrow l)$ in Σ, and compute enforced(Σ_Q); (2) if enforced(Σ_Q) is conflicting for all GFDs $Q[\bar{x}](X \rightarrow l)$ in Σ, then return false; otherwise, return true. This process is in $O(|\Sigma|^2 \times k^k)$ time, thus in PTIME for constant k.

Given a set Σ of GFD and GFD $\varphi = Q[\bar{x}](X \rightarrow l)$, we check whether $\Sigma \models \varphi$ as follows: (1) compute the set Σ_Q of GFDs embedded in Q and closure(Σ_Q, X); (2) if closure(Σ_Q, X) is conflicting or if $l \in$ closure(Σ_Q, X), then return true; otherwise return false. It takes $O((|\varphi| + |\Sigma|) \times k^k)$ time, which is in PTIME for constant k.

(b) We show that the validation problems is co-W[1]-hard by reduction from the complement of the k-clique problem, which is W[1]-complete [12]. The k-clique problem is to decide, given an undirected graph G and a natural number k, whether there is a clique of size k in G. We omit the details of the reduction here. □

***k*-bounded** GFDs. A pattern $Q[\bar{x}]$ is *k-bounded* if $|\bar{x}| \leq k$, for a constant k. A set Σ of GFDs is *k-bounded* if for all $Q[\bar{x}](X \rightarrow Y)$ in Σ, $Q[\bar{x}]$ is k-bounded. It is easy to verify the following.

Proposition 2: *When k is a constant, the satisfiability, implication and validation problems are in* PTIME *for k-bounded* GFDs. □

Proof: The algorithms given in the proof of Theorem 1 for satisfiability and implication are in PTIME for constant k. For validation, an $O(|\Sigma| \cdot |G|^k)$ time algorithm works as follows: for each GFD $Q[\bar{x}](X \rightarrow Y)$ in Σ, enumerate all matches $h(\bar{x})$ of Q in G and check whether $h(\bar{x}) \models X \rightarrow Y$; this is in PTIME for constant k. □

4 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 contain no redundant and trivial GFDs, and (2) *frequent* GFDs that capture regularities and constraints.

We focus on connected patterns and defer the handling of disconnected patterns to a later paper. A pattern $Q[\bar{x}]$ is connected if every pair of nodes in $Q[\bar{x}]$ is connected by an undirected path.

4.1 Reduced GFDs and GFD Cover

We first formulate nontrivial and reduced GFDs, following the practice of mining relational dependencies (*e.g.,* [8, 16]).

Nontrivial GFDs. A GFD $\varphi = Q[\bar{x}](X \rightarrow l)$ is *trivial* if either (a) X is "equivalent" to false, *i.e.,* it cannot be satisfied; or (b) l can be derived from X via the transitivity of equality. Obviously, we are only interested in discovering nontrivial GFDs.

Reduced GFDs. We start with an ordering.

(1) Given 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 upgrades some labels in Q' to wildcard. That is, Q is a topological constraint less restrictive than Q'. For instance, $Q_2 \ll G_2$ if we treat G_2 of Fig. 1 as a pattern.

(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 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 Q_2 of Example 1, we may pick pivot x and write φ_2 as $Q_2[\underline{x}, y, z](\emptyset \to y.\text{name} = z.\text{name})$; similarly for φ_1 and φ_3.

(3) Consider positive GFDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \to l_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to l_2)$. We say that φ_1 *reduces* φ_2, denoted by $\varphi_1 \ll \varphi_2$, if there is an isomorphism f from Q_1 to a subgraph of Q_2 such that (a) $f(z_1) = z_2$, where z_i denotes the pivot of Q_i, *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 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 4: Recall $\varphi_1 = Q_1[x, y](X_1 \to l)$ (Example 1), where X_1 = {y.type = "film"}, and l is x.type = "producer". Let x be pivot. (1) Consider GFD $\varphi_1^1 = Q_1^1[x, y, z](X^1 \to l)$ 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.name = 'Selling out'}. Then $\varphi_1 \ll \varphi_1^1$. (2) Consider GFD $\varphi_1^2 = Q_1^1[x, y, z](X_1^2 \to l)$, where X_1^2 consists of y.name = '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.

A reduced positive φ guarantees the following: (a) *left-reduced* [8, 16, 27], *i.e.,* $G \not\models Q[\bar{x}](X' \to l)$ for any proper subset $X' \subsetneq X$, and hence X does not include redundant literals; and (b) *pattern-reduced, i.e.,* $G \not\models Q'[\bar{x}'](X \to l)$ for any $Q'[\bar{x}']$ with $Q'[\bar{x}'] \ll Q[\bar{x}]$, $X \to l$ is defined on $Q'[\bar{x}']$ (with variable renaming).

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

Cover of GFDs. Consider a set Σ of GFDs such that $G \models \Sigma$.

We say that Σ is *minimal* if for all $\varphi \in \Sigma$, $\Sigma \not\equiv \Sigma \setminus \{\varphi\}$, *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 [3] for more about covers).

4.2 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 $\varphi = Q[\bar{x}](X \to Y)$. Following the conventional notion, the support of φ would be defined as the number of matches of Q in G that satisfy $X \to Y$. However, as observed in [13, 29], 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 in terms of the support of its pattern and correlation of its attributes.

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

We define the *support* of pattern Q as:

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

It quantifies the frequency of the entities in G that satisfy the topology constraint posed by Q "pivoted" at z.

It is for the anti-monotonicity of support of GFDs that we employ pivots in the definition above. The anti-monotonicity allows us to speed up the discovery process along the same lines as conventional data mining. To simplify the discussion we do not include pivots in supp(Q, G) when it is clear from the context.

Correlation measure. To quantify the variable dependencies in $Q[\bar{x}]$, we define the *correlation* $\rho(\varphi, G)$ of GFD φ as

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

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

Intuitively, $\rho(\varphi, G)$ characterizes the dependency of l on X with "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 $Q(G, z)$ with $h(\bar{x}) \models X \to l$. That is, it does not accurately measure the correlation of X and l.

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

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

Anti-monotonicity. We next justify that the support is well defined in terms of anti-monotonicity and GFD ordering.

Theorem 3: *For any graph G and nontrivial positive* GFDs φ_1 *and* φ_2, *if* $\varphi_1 \ll \varphi_2$ *then* supp$(\varphi_1, G) \geq$ supp(φ_2, G). □

Proof: Let $\varphi_1 = Q_1[\bar{x}_1](X_1 \to l_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to l_2)$, with pivot z_1 and z_2, respectively. To see supp$(\varphi_1, G) \geq$ supp(φ_2, G), we show that $Q_2(G, X_2l_2, z_2) \subseteq Q_1(G, X_1l_1, z_1)$. That is, for any node v in G, if there is 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 is 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 $h_1(\bar{x}_1) \models l_1$. □

For instance, for φ_1 and φ_1^1 of Example 4, we have $\mathrm{supp}(\varphi_1, G) \geq \mathrm{supp}(\varphi_1^1, G)$ 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. A negative $\varphi = Q[\bar{x}](X \rightarrow \text{false})$ has two forms (Section 2): (a) $X = \emptyset$, and hence $Q(G, z) = \emptyset$ at any pivot z in a "consistent" graph G; and (b) $X \neq \emptyset$ and is satisfiable. Putting φ in the normal form $Q[\bar{x}](X \rightarrow l)$ by taking false as a literal l, in both cases, $Q(G, Xl, z) = \emptyset$. Thus we can no longer discover negative GFDs by computing $Q(G, Xl, z)$ and $Q(G, z)$.

We are interested in negative GFD φ that results from a "minimal trigger" to a positive $Q'[\bar{x}'](X' \rightarrow l')$, either by "vertical extension" of Q' with a single edge (possible with new nodes), or by "horizontal extension" of X' with a single literal. Thus we define

$$\mathrm{supp}(\varphi, G) = \max_{\varphi' \in \Phi'}(\mathrm{supp}(\varphi', G)),$$

where (1) if $X = \emptyset$, Φ' consists of patterns $Q'[\bar{x}']$ with the same pivot z such that $\mathrm{supp}(Q', G) > 0$, and Q' is obtained from Q by removing an edge (possibly nodes too); and (2) if $X \neq \emptyset$, Φ' consists of positive GFDs $\varphi' = Q[\bar{x}](X' \rightarrow l)$ with the same pivot z such that $G \models \varphi'$, and there exists a literal l' in X with $X = X' \cup \{l'\}$. In case (1) (resp. case (2)), we refer to pattern $Q' \in \Phi'$ (resp. positive GFD $\varphi' \in \Phi'$) with the maximum support in Φ' as a *base* of φ.

That is, $\mathrm{supp}(\varphi, G)$ 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, φ suggests a meaningful negative GFD.

Moreover, Theorem 3 holds on generic GFDs, positive or negative (proof omitted here). This is the first anti-monotonicity result for mining functional dependencies with graph patterns. It ensures the feasibility of GFD discovery (see Section 5 for details).

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

Open World Assumption (OWA). The OWA states that absent data cannot be used as counterexamples in knowledge bases [19, 22]. The support of GFDs 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.

4.3 The Discovery Problem

We now 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.,* $\mathrm{supp}(\varphi, G) \geq \sigma$.

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.

We take k 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 3), and (b) by Proposition 2, 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 (Section 7).

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 "cleaned". 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 low σ to find non-frequent GFDs, just like [7, 22]. (3) Our techniques apply to the discovery of general GFDs without these restrictions.

5 SEQUENTIAL GFD DISCOVERY

We start with a sequential algorithm for GFD discovery, denoted as SeqDisGFD. It consists of two algorithms: (1) SeqDis that, given G, k and σ, discovers the set Σ of k-bounded minimum σ-frequent GFDs, and (2) SeqCover that, given Σ, computes a cover Σ_c of Σ. We present SeqDis and SeqCover in Sections 5.1 and 5.2, respectively.

5.1 Sequential GFD Mining

A brute-force algorithm first enumerates all frequent patterns Q in G following conventional graph pattern mining (*e.g.,* [13, 39]), and then generates GFDs with Q by adding literals. However, enumeration of 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. 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. In the first iteration, it "cold-starts" GFD discovery by initializing a GFD *generation tree* T with frequent GFDs that carry a single-node pattern. The tree T is then expanded by interleaving two levelwise spawning processes: *vertical spawning* to extend graph patterns Q, and *horizontal spawning* to generate dependencies $X \rightarrow Y$. At each iteration i $(0 < i < k^2)$, SeqDis generates and verifies GFD candidates, and fills the level-i part of tree T. It works in two steps as follows.

(1) Pattern verification. Algorithm SeqDis first performs *vertical spawning*, which generates new graph patterns at level i of 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.

(2) GFD Validation. It then performs *horizontal spawning*, which associates a set of literals with the newly verified graph patterns at level i of T to generate a set of GFD candidates. For each batch of GFD candidates, it performs GFD *validation* to find GFDs in Σ_i, *i.e.,* those candidates at level i that are satisfied by G, and are frequent and minimum. The validation process terminates when all the GFD candidates pertaining to the patterns at level i are validated.

The two steps iterate until no new GFDs can be spawned, or all the k-bounded GFDs are checked (*i.e.,* $i = k^2$).

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

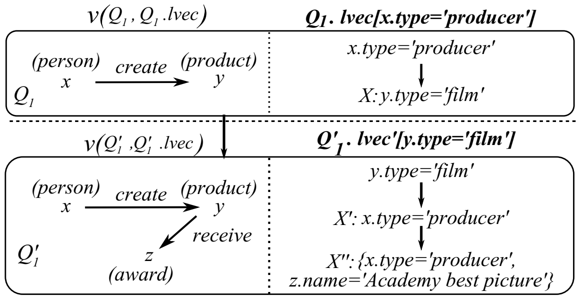


Figure 2: GFD **generation tree**

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

Example 5: A fraction of GFD generation tree T is shown in Fig. 2. It contains a node $v(Q_1[\bar{x}], \text{lvec})$ at level 1, and $v(Q'_1[\bar{x}], \text{lvec})$ at level 2. Node $v(Q_1[\bar{x}], \text{lvec})$ stores graph pattern $Q_1[\bar{x}]$ of Fig. 1, and its literal tree Q_1.lvec$[l]$ rooted at x.type = "producer". At node $v(Q'_1[\bar{x}], \text{lvec})$, a literal tree is rooted at a different literal y.type = "film". There exists an edge from X' to X'' in Q'_1.lvec, since $X'' = X' \cup \{z.\text{name} = \text{"Academy best picture"}\}$. There exists an edge from $v(Q_1[\bar{x}], \text{lvec})$ to $v(Q'_1[\bar{x}], \text{lvec})$ in T, since Q'_1 is obtained by adding a single edge (y, z) to pattern Q_1. The literal at node X in Q_1.lvec encodes the GFD φ_1 of Example 1. Similarly, X'' in node Q'_1.lvec encodes GFD $\varphi_4 = Q'_1[x, y, z](\{x.\text{type} = \text{"producer"}, z.\text{name} = \text{"Academy best picture"}\} \to y.\text{type} = \text{"film"})$. □

For $\varphi = Q[\bar{x}](X \to l)$ at level i, the length $|X|$ is at most $J = i|\Gamma|(|\Gamma| + 1)$, where Γ consists of active attributes in G.

GFD Spawning. Generation tree T spawns new GFD candidates 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 $1 \leq i \leq k^2$. 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 Q_1 at level i, VSpawn maintains a set $P(Q_1)$ of edges from the *parents* of Q_1 at level $i-1$ (see its use in Section 6). If Q'_1 is added to iso(Q_1), $P(Q'_1)$ 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: Consider tree T of Fig. 2. A pattern Q'_1 is spawned by VSpawn(2) from Q_1, by adding an 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 GFD candidates $\varphi = Q[\bar{x}](X \to l)$, 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 $Q[\bar{x}](\emptyset \to l)$ 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 T levelwise horizontally, but adds no patterns. We denote the set of GFDs generated by HSpawn(i, j) as $\Sigma_{C_{ij}}$.

Example 7: Continuing Example 6, HSpawn$(2, j)$ is performed on the newly added patterns at level 2 of T. For pattern Q'_1 and literal tree Q'_1.lvec rooted at $l = (y.\text{type} = \text{"film"})$, HSpawn(2, 2) extends X' at level $j = 1$ to X'' at $j = 2$, by adding z.name = "Academy best picture". This yields $\varphi_4 = Q_1[x, y, z](X'' \to l)$ (see Fig. 2). □

Algorithm SeqDis also "upgrades"a node label $L_Q(v)$ in a GFD φ to wildcard '_' if $L_Q(v)$ ranges over all labels of Θ in variants of φ, to get a more general pattern; similarly for edge $L_Q(e)$.

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

Lemma 4: *For a cover* Σ_c *of* GFDs *above support* σ,

(a) Σ_c *includes no trivial* GFDs φ;

(b) *for any* $\varphi = Q[\bar{x}](X \to l)$, *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* supp$(Q, G) < \sigma$, *then* Σ_c *does not include* $\varphi' = Q'[\bar{x}](X' \to l')$ *if* $Q \ll Q'$. □

Proof: (a) For trivial GFD φ, we have that supp$(\varphi, G) = 0 < \sigma$. (b) After we check a GFD φ as stated in (b) above, HSpawn can safely stop generating φ'. Indeed, if supp$(\varphi) \geq \sigma$, then φ is a candidate for Σ_c but φ' is not, since φ' is not reduced. If supp$(\varphi) < \sigma$, then by Theorem 3, supp$(\varphi') \leq$ supp(φ) and hence φ' cannot make a candidate for Σ_c either. (c) If $Q \ll Q'$, then supp$(Q, G) \geq$ supp(Q', G) as verified in the proof of Theorem 3 (we omit the details here). Moreover, supp$(\varphi', G) \leq$ supp(Q', G) by the definition of supp(φ', G). Since supp$(Q, G) < \sigma$, supp$(\varphi', G) \leq$ supp$(Q', G) \leq$ supp$(Q, G) < \sigma$. That is, φ' cannot be in Σ_c, and can be pruned by VSpawn. □

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, Q is expanded from a pattern Q' by adding a single edge, where supp$(Q', G) \geq \sigma$, since otherwise supp$(\varphi, G) = \max$ 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 supp$(Q', \bar{z}) = 0$, and adds $\varphi = Q'[\bar{x}](\emptyset \to \text{false})$ to Σ_N. It guarantees that 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 $\text{NHSpawn}(i, j)$ extends $\text{HSpawn}(i, j)$ as follows. As soon as $\text{HSpawn}(i, j)$ 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 8: The negative GFD φ_3 of Example 3 is discovered as follows, in case (a). 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 Q_3 as in Fig. 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. □

5.2 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 [20] reviewed in Section 3.

Algorithm SeqCover. Making use of the characterization and following the algorithms for computing cover of relational FDs (see, *e.g.,* [3]), SeqCover works as follows: for each $\varphi \in \Sigma$, it checks whether $\Sigma \setminus \{\varphi\} \models \varphi$ based on the characterization; if so, 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 4. Moreover, SeqCover correctly computes a cover of Σ. (1) For each $\varphi \in \Sigma$, it performs implication test following the characterization of GFD implication. (2) When it terminates, no φ is implied by Σ_c. Thus Σ_c is a cover of Σ.

5.3 Analysis of Sequential GFD Discovery

We next analyze the complexity of algorithm 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" [14]. 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 as 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 sequential cost for GFD discovery, which subsumes subgraph isomorphism.

6 PARALLEL GFD DISCOVERY

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

6.1 Parallel Scalability Revisited

To characterize the effectiveness of parallel GFD discovery in large-scale graphs, we revisit the notion of *parallel scalability* [33]. 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 $t(|G|, k, \sigma)$.

A GFD discovery algorithm $\mathcal{A}_p$ is *parallel scalable relative to* $\mathcal{A}$ if its cost by using n processors can be expressed as

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

where the notation $\tilde{O}$ hides $\log(n)$ factors (see, *e.g.,* [40]).

Intuitively, parallel scalability guarantees speedup of $\mathcal{A}_p$ *relative to* a "yardstick" sequential algorithm [33]. A parallel scalable $\mathcal{A}_p$ "linearly" reduces the sequential cost of $\mathcal{A}$. We show the following.

Theorem 5: *There exists an algorithm* DisGFD *for* GFD *discovery that is parallel scalable relative to* SeqDisGFD. □

The main conclusion we can draw from Theorem 5 is that the more processors are used, the faster algorithm DisGFD is. Hence DisGFD can scale with large G by adding processors as needed. It makes GFD discovery feasible in real-life graphs.

A proof sketch. We provide such a DisGFD as a proof of Theorem 5. It consists of two algorithms: (a) ParDis (Section 6.2) that "parallelizes" its sequential counterpart SeqDis to discover the set Σ of k-bounded minimum σ-frequent GFDs from G, and (b) ParCover (Section 6.3) that "parallelizes" SeqCover to compute a cover Σ_c of Σ. We will show that both algorithms are parallel scalable relative to their sequential counterparts in SeqCover (Section 5.2). □

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 [31], and distributed across n workers $(P_1, \ldots, P_n)$.

6.2 Parallel GFD Mining

We start with algorithm ParDis, shown in Fig. 3. The algorithm runs in supersteps. Similar to SeqDis, it uses Σ_i to store all minimum σ-frequent GFDs with i edges, at superstep i. Algorithm ParDis first initializes Σ and tree T (lines 1-2), and then performs at most k^2 supersteps (lines 3-15). At each superstep i $(0 < i < k^2)$, 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) of SeqDis, respectively.

(1) Parallel pattern verification. ParDis performs vertical spawning $\text{VSpawn}(i)$ (Section 5.1) at master S_c to generate graph patterns at level i of T (line 4). 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 GFD candidates.

(2) Parallel GFD *validation.* Algorithm ParDis then performs horizontal spawning $\text{HSpawn}(i, j)$ (Section 5.1) at S_c with the verified graph patterns to generate a set of GFD candidates. Operation $\text{HSpawn}(i, j)$ iterates for $j \in [1, J]$, where $J = i|\Gamma|(|\Gamma| + 1)$ (see Section 5.1), followed by parallel validation of the candidate GFDs (lines 9-14). Once each superstep i terminates, Σ is expanded with all verified minimum frequent GFDs Σ_i (lines 15).

The two steps iterate until no new GFDs can be spawned, or all the k-bounded GFDs are checked (*i.e.,* $i = k^2$).

```
Algorithm ParDis
Input: a fragmented graph $G$, integer $k$, support threshold $\sigma$.
Output: a set $\Sigma$ of all $k$-bounded minimum $\sigma$-frequent GFDs $\varphi$.
1.  set $\Sigma := \emptyset$; GFD tree $T := \emptyset$; integer $i := 1$; $\text{flag}_V :=$ true;
2.  SpawnGFD(T); /* initialize $T$ with single-node GFDs;
3.  while $i \leq k^2$ and $\text{flag}_V$ do /* superstep $i$ */;
4.      VSpawn($i$); $\Sigma_i := \emptyset$;
5.      $\text{flag}_V :=$ false if no new pattern is spawned;
6.      if $\text{flag}_V$ then
7.          parallel graph pattern matching;
8.          $j := 1$; $\text{flag}_H :=$ true;
9.          while $j \leq J$ and $\text{flag}_H$ do
10.             set $\Sigma_{C_{ij}} :=$ HSpawn($i, j$);
11.             $\text{flag}_H :=$ false if no new GFD candidates are spawned;
12.             if $\text{flag}_H$ then
13.                 parallel GFD validation for $\Sigma_{C_{ij}}$;
14.                 $\Sigma_i := \Sigma_i \cup \Sigma_{C_{ij}}$; $j := j + 1$;
15.     $\Sigma := \Sigma \cup \Sigma_i$;  $i := i + 1$;
16. return $\Sigma$;
```

Figure 3: Algorithm ParDis

When one of the conditions is satisfied, ParDis returns Σ (line 16).

We next present the details of *parallel verification* and *parallel validation*, which dominate the cost of GFD discovery.

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

(1) At S_c, for each pattern $Q' \in \mathcal{Q}'_i$, ParDis constructs a *work unit* (Q, e) that "decomposes" Q' into a verified pattern Q, and an edge e added to Q to obtain Q'. 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 $t \in [1, n]$". That is, 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* computes $Q'(F_s)$, 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 $Q'(F_s)$ with isomorphism check. After this, each worker P_i stores matches $Q'(F_s)$ for the next round. Once all the patterns are verified, it sends a flag Terminate to S_c.

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

Load balancing. We initially evenly partition the edges of G across n workers via vertex cut. This helps us cope with skewed graphs, in which a large number of low-degree nodes connect to dense, small groups. Moreover, at each superstep, if $Q'(F_s)$ is "skewed", *i.e.,* much larger than other $Q'(F_t)$'s, we re-distribute $Q'(F_s)$ evenly across workers. It balances (a) parallel validation workload, and (b) parallel matching work unit $Q'(F_s) \bowtie e'(F_t)$ in the next superstep.

Parallel validation. Once all workers send Terminate to S_c, algorithm ParDis starts to perform HSpawn(i,j) to generate GFD candidates $\Sigma_{C_{ij}}$ master at (i,j) of T. It then posts $\Sigma_{C_{ij}}$ to the n workers, to validate the GFD candidates in parallel.

Workload balancing is performed when necessary, using the same strategy as for parallel pattern matching.

(1) For each $\varphi = Q[\bar{x}](X \rightarrow l)$ in $\Sigma_{C_{ij}}$, each worker P_s computes in parallel (a) local supports $\text{supp}(\varphi, F_s) = Q(F_s, Xl, z)$ at pivot z and (b) a Boolean flag SAT^i_φ, set to true if $F_s \models \varphi$. It then sends $\text{supp}(\varphi, F_s)$ and SAT^i_φ to S_c by using $Q(F_s)$ from VSpawn(i).

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

The two steps iterate until either no new GFDs can be spawned, or when j reaches $J = i|\Gamma|(|\Gamma| + 1)$, the maximum length of X. By the levelwise generation of candidates and Lemma 4, the GFDs validated are guaranteed to be minimum.

Parallel scalability. To show that ParDis is parallel scalable relative to SeqDis, 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 next analyze the costs of parallel matching; the argument for validation is similar.

The cost at each worker is dominated 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 $n \ll |G|$ and $k \geq 2$; (c) balancing 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 $O(\frac{|G|^k}{n})$, as the load is evenly distributed in step (c) (load balancing). One can verify by induction on the size $|Q|$ of Q that step (c) is in $O(\frac{|G|^k}{n})$ time. Taken together, the parallel cost of pattern matching is $O(\frac{|G|^k}{n})$. Note that this is the worst-case time complexity. In practice, (a) redundant GFD candidates are pruned early by the strategies of Lemma 4, and (b) incremental pattern matching reduces unnecessary recomputation.

```
Algorithm ParCover
Input: A set $\Sigma$ of GFDs, and the GFD tree $T$ generated by ParDis.
Output: A cover $\Sigma_c$ of $\Sigma$.
1.  set $\Sigma_c := \emptyset$; set $W := \emptyset$
2.  for each pattern $Q_j$ of $T$ do /* partition of $\Sigma$ with $T$ */
3.      create groups $\Sigma^{Q_j} \subseteq \Sigma$;
4.      construct $\Sigma_{Q_j}$; $W := W \cup \Sigma_{Q_j}$
5.  evenly distribute work units $W$ to all workers;
6.  $\Sigma_{c_i} :=$ ParImp($W_i$); /* local checking load $W_i$ at worker $P_i$ */
7.  $\Sigma_c :=$ the union of all $\Sigma_{c_i}$;
8.  return $\Sigma_c$;
```

Figure 4: Algorithm ParCover

6.3 Parallel Cover Computation

We next develop algorithm ParCover, shown in Fig. 4.

Group checking. Algorithm ParCover parallelizes SeqCover by leveraging the characterization of GFD implication (see Section 3). (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 (*i.e.,* the two graphs are not isomorphic). (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.

Lemma 6: ***[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. Given Σ, algorithm ParCover partitions Σ into $\Sigma^{Q_1}, \ldots \Sigma^{Q_m}$, one for each pattern Q_j in Σ (line 3). It constructs group Σ_{Q_j} for each pattern Q_j (line 4). This is done by taking advantage of the GFD generation tree T (see Section 5.1). It traces the ancestors of Q_j that are in T, by following the parent edges of $P(Q_j)$, 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}.

ParCover first constructs workload W as a set of Σ_{Q_j} for $1 \leq j \leq m$, and then distributes W to n workers via load balancing (line 5; see details below). Upon receiving the assigned work units W_j, each worker P_j invokes a (sequential) procedure ParImp(W_j) (line 6), 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 $\Sigma_{r_i} = \Sigma^{Q_i} \setminus \Sigma_{N_i}$. That is, Σ_{N_i} includes GFDs in Σ^{Q_i} that are not entailed by other GFDs in Σ (Lemma 6). ParImp(W_j) returns the union Σ_{c_j} of Σ_{N_i} for all $\Sigma_{Q_i} \in W_j$. ParCover takes the union of Σ_{c_j}'s as Σ_c (lines 7-8). One can verify the following: (1) Σ_c is minimal; and (2) $\Sigma_c \equiv \Sigma$.

Example 9: 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 Fig 1, φ'_1 has a pattern of one edge receive(y, z) in Fig 2, and φ_5 (resp. φ_6) has a pattern of one edge parent(x, y) (resp. parent(y, x)) in Fig 1; and (2) φ_3 and φ_4 are at level 2 of T; φ_3 has pattern Q_3 in Fig 1, and φ_4 has Q'_1 of Fig 2. 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, ParCover partitions Σ and constructs 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 tests 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, by an approximation algorithm of factor 2 by using the techniques of [4] (details omitted for the lack of space).

Parallel scalability. Suppose that $\Sigma = \{\varphi_1, \ldots, \varphi_M\}$. We show that ParCover is parallel scalable. Recall that ParCover computes all *non-redundant* GFDs in Σ^{Q_i} at each worker in parallel, by "plugging" in a sequential algorithm ParImp. By evenly balancing the workload, its cost is $\tilde{O}(\frac{T(\Sigma_{Q_1}, \varphi_1) + \ldots + T(\Sigma_{Q_m}, \varphi_M)}{n})$. Since $\Sigma_{Q_i} \subseteq \Sigma$, $T(\Sigma_{Q_i}, \varphi_j) \leq T(\Sigma, \varphi_j)$. Then ParCover is in $O(\frac{t_2(\Sigma, k)}{n})$ time, where $t_2(\Sigma, k) = T(\Sigma, \varphi_1) + \ldots + T(\Sigma, \varphi_M)$ (see Section 5.2). The load balancing itself takes $O(|\Sigma| n \log n)$ time, 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 6.2) 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.

7 EXPERIMENTAL STUDY

Using real-life and synthetic data, we experimentally evaluated algorithm DisGFD for (1) the parallel scalability with the increase of workers, (2) the scalability with graphs, (3) the impact of bound k, support threshold σ and active attributes Γ, (4) the parallel scalability of ParCover, and (5) the effectiveness of finding useful GFDs.

Experimental setting. We used three real-life graphs: (a) *DBpedia*, a knowledge graph [1] with 1.72 million entities of 200 types and 31 million links of 160 types, (b) *YAGO2*, an extended knowledge base of YAGO [38] with 1.99 million nodes of 13 types, and 5.65 million links of 36 types; and (c) *IMDB* [2], a knowledge base with 3.4 million nodes of 15 types and 5.1 million edges of 5 types. For *each entity*, we picked 5 active attributes from cleaned ontology (*e.g.,* WordNet [38]). Each entity in *YAGO2* has at most 4 attributes and 98% of nodes in *DBpedia* have at most 7 attributes; thus 5 is reasonable. For each attribute $A \in \Gamma$, we took 5 most frequent values.

We also developed a generator for synthetic graphs $G = (V, E, L, F_A)$, controlled by the numbers $|V|$ of nodes (up to 30 million) and edges $|E|$ (up to 60 million), with L drawn from a set of 30 labels, and F_A assigning a set Γ of 5 active attributes, where each $A \in \Gamma$ draws a value from 1000 values.

GFD *generator*. To test the scalability of GFD implication, we developed a generator to produce sets Σ of GFDs, controlled by $|\Sigma|$ (up to 10000) 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 SeqDisGFD, including SeqDis and SeqCover; (2) DisGFD for parallel GFD mining, including ParDis and ParCover; (3) ParGFD$_n$, a version of DisGFD without GFD pruning (Lemma 4) for ParDis; (4) ParGFD$_{nb}$, DisGFD without load balancing (Section 6.2); and (5) ParCover$_n$, ParCover without GFD grouping (Lemma 6).

We also implemented two baselines. (1) Algorithm ParArab splits the pattern mining and dependency discovery. (a) It first discovers all frequent patterns Q in parallel, by using Arabasque [39], a state-of-the-art parallel graph pattern mining system. (b) It then extends each Q to GFDs with literals, and verifies the latter in parallel. It uses the same procedure ParCover for implication. (2) Algorithm ParAMIE, the parallel algorithm to discover AMIE rules [7].

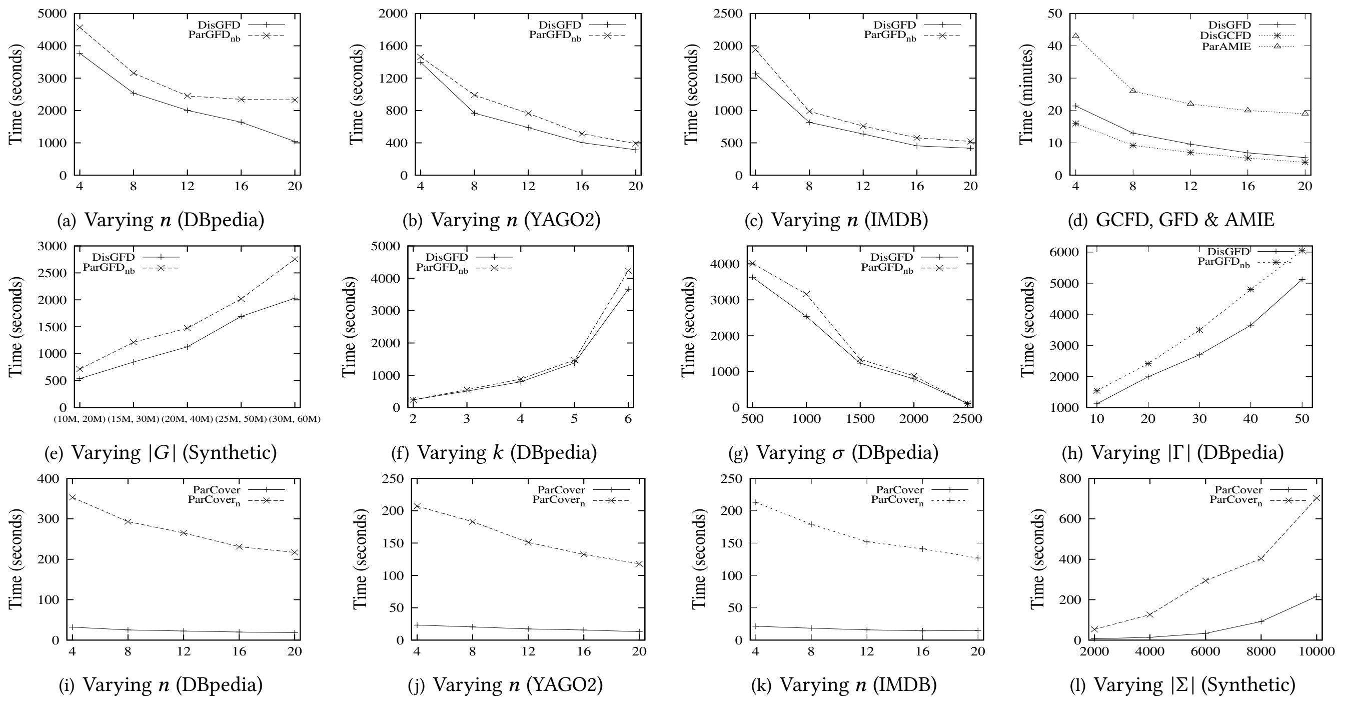
In addition, we developed algorithm ParCGFD for mining GCFDs, an extension of relational CFDs [15] with path patterns [24], which makes a special case of GFDs.

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 [39]. For an application, 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 2.3GHz. We used up to 20 instances. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

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



(a) Varying n (DBpedia) (b) Varying n (YAGO2) (c) Varying n (IMDB) (d) GCFD, GFD & AMIE

(e) Varying $|G|$ (Synthetic) (f) Varying k (DBpedia) (g) Varying σ (DBpedia) (h) Varying $|\Gamma|$ (DBpedia)

(i) Varying n (DBpedia) (j) Varying n (YAGO2) (k) Varying n (IMDB) (l) Varying $|\Sigma|$ (Synthetic)

Figure 5: Performance evaluation of parallel GFD discovery

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 integrated process and pruning strategy.

We hence report only the performance of 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$_{nb}$. We fixed $k = 4$ and $\sigma = 500$, and report the results on *DBpedia*, *YAGO2* and *IMDB* in Figures 5(a), 5(b) and 5(c), respectively. As shown in Fig. 5(a), (1) DisGFD is parallel scalable. It is 3.6 times faster on average when n changed from 4 to 20. (2) DisGFD outperforms ParGFD$_{nb}$ by 1.5 times on average, and by 2.2 times when $n = 20$. This verifies the effectiveness of load balancing. (3) DisGFD is feasible in practice: it takes 17 minutes when $n = 20$. The results in Figures 5(b) and 5(c) are consistent. DisGFD (1) is 4 and 3.8 times faster when n varies from 4 to 20, and (2) outperforms ParGFD$_{nb}$ by 1.2 and 1.3 times on *YAGO2* and *IMDB*, respectively. Load balancing is more effective on *DBpedia* since it is denser than *YAGO2* and *IMDB*, and yields more graph patterns.

A cost breakdown demonstrate that the parallel pattern verification and GFD validation dominate the cost. Nonetheless, the parallel costs are reduced when more workers are used.

We also compared DisGFD with ParCGFD and ParAMIE on *YAGO2*. We set $k = 3$ for GFDs and GCFDs, which is the default size of variable set per AMIE rule [7]. Figure 5(d) shows the following. (1) DisGFD is comparable to ParCGFD, although it finds more GFDs with general patterns. (2) Although GFDs are more expressive, DisGFD outperforms ParAMIE by 3.4 times on average, due to its pruning strategies. The results on other graphs are consistent.

Sequential cost. Figure 6 reports the cost of sequential SeqDisGFD.

dataset	SeqDisGFD	SeqCover	**GFDs**	**GCFDs**	**AMIE**
DBpedia	14322s	45s	321/1724	202/1578	481/1500
YAGO2	4963s	32.5s	145/605	104/698	69/600

Figure 6: Sequential cost and rule #/avg. support

While ParGFD$_n$ and ParArab fail to complete, algorithm SeqDisGFD performs reasonably well: it takes 1.3 hours to discover GFDs from *YAGO2* of 7.64 million entities and edges.

We report the number and average support of mined rules on two real-life knowledge-base datasets in Fig. 6. While GFDs discovered have to be satisfied by the graphs, AMIE rules are soft constraints that are not necessarily satisfied by the datasets. We set the PCA (partial completeness assumption) confidence [22] threshold of AMIE as 0.5. We found that the discovered GFDs can express all the AMIE rules with PCA confidence 1. Moreover, most AMIE rules cannot capture inconsistencies via constant bindings (see Exp-5).

Exp-2: Scalability with $|G|$. Fixing $k = 4$, $\sigma = 500$ and $n = 20$, we evaluated the scalability of DisGFD by varying the size of synthetic graph $|G| = (|V|, |E|)$ from (10M, 20M) to (30M, 60M). As shown in Fig. 5(e), (1) it 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 minutes in G of size $(30M, 60M)$.

As indicated in Fig. 6, the impact of $|G|$ on SeqDisGFD is consistent: the larger G is, the longer SeqDisGFD takes.

Exp-3: Impact of parameters. We evaluated the impact of pattern size k, support σ and active attributes Γ. We report the results on *DBpedia* here; the results on the other datasets are consistent.

Varying k. Fixing $n = 8$, $\sigma = 1000$ and $|\Gamma| = 150$, we varied k from 2 to 6, As shown in Fig. 5(f), (1) it takes both DisGFD and ParGFD$_{nb}$ longer to find GFDs with larger patterns, as expected; (2) DisGFD

(σ, k, $\|\Gamma\|$) (YAGO)	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%

Figure 7: Error detection accuracy

outperforms ParGFD$_{nb}$ by 1.2 times; and (3) DisGFD can find GFDs with reasonably large patterns: it takes 20 minutes to find 5-bounded GFDs. Note that we use different y-axis from Exp-1.

Varying σ. Fixing $n = 8$, $k = 4$ and $|\Gamma|$= 150, we varied σ from 500 to 2500, and evaluated DisGFD and ParGFD$_n$. As shown in Fig. 5(g), both algorithms take less time with larger σ, as higher σ prunes more GFD candidates, and reduces generation and verification time. This again verifies the effectiveness of our pruning strategy.

Varying $|\Gamma|$. Fixing $n = 8$, $k = 4$, $\sigma = 500$, we varied $|\Gamma|$ from 50 to 250. Figure 5(h) tells us that both algorithms take longer with larger $|\Gamma|$, as more GFD candidates are generated.

The impacts of k, σ and Γ on SeqDisGFD are consistent.

Exp-4: Cover computation. In the same setting as Figures 5(a)-5(c), we report the scalability of ParCover on *DBpedia*, *YAGO2* and *IMDB* in Figures 5(i)-5(k), 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 on average. This validates the effectiveness of GFD grouping and load balancing.

As shown in Fig. 6, the sequential SeqCover also does well. It takes at most 45.1 seconds over the three real-life datasets.

Varying $|\Sigma|$. Fixing $n = 4$, we evaluated ParCover by varying the number of GFDs from 2000 to 10000. As shown in Fig. 5(l), ParCover takes longer when $|\Sigma|$ is larger, as expected. It is less sensitive to $|\Sigma|$ than ParCover$_n$, since its 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 Fig. 6.

Exp-5: Effectiveness. We validate the GFDs discovered.

Error detection accuracy. We make a direct 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 each such node v, changed β% of either the active attribute values or the labels of edges of v (to favor AMIE, which do not support wildcard), with values that did not appear in *YAGO2*. We took care to make changes that involve the consequence Y of $X \to Y$ in Σ and Σ^A discovered. For GFDs, we apply methods of [20] to validate the mined GFDs in the graph.

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

The accuracy of GFDs, GCFDs and AMIE and the impact of σ, k and Γ are reported in Fig. 7. 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 the rules. (2) GFDs have better accuracy if discovered with smaller σ, larger k and larger $|\Gamma|$, since more GFDs are discovered to "cover" the inconsistencies. (3) The accuracy of AMIE is close to the accuracy of GFDs since more AIME rules are mined (their PCA confidence is not "1" and the AMIE rules may be "dirty").

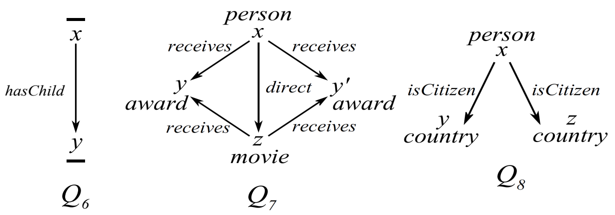


Figure 8: Real-life GFDs (YAGO2)

Real-world GFDs. We also manually inspected the GFDs and validated their usefulness. As examples, we give 3 GFDs found in *YAGO2* by DisGFD, with patterns shown in Fig. 8.

GFD$_1$: $Q_6[x, y]$ ($\emptyset \to x$.familyname = y.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.name = "Gold Bear" $\wedge$ y'.name = "Gold Lion" $\to$ false). It tells us that no movie receives both Gold Bear and Gold Lion awards, because the Italian and German film festivals require their participants to be "initial release".

GFD$_3$: $Q_8[x, y, z]$($X_8 \to$ false), a negative GFD, where X_8 consists of y.name = "US" and z.name = "Norway". It states that Norway does not admit dual citizenship.

These GFDs carry a DAG pattern, constants, wildcard _ or false, beyond the capacity of most graph FD proposals and AMIE rules [7, 22]. They help us detect errors and extract knowledge (especially facts that are not familiar to some people, *e.g.,* GFD$_2$ and GFD$_3$).

Summary. We find that (1) GFD discovery is feasible in practice. It takes 591 seconds for DisGFD to find frequent 4-bounded GFDs from real-life graphs on average. GFD discovery is parallel scalable: DisGFD (resp. ParCover) is 3.78 (resp. 1.75) times faster on average when n is increased from 4 to 20. The grouping strategy in ParCover improves its performance by 10 times. (2) Our integrated method with pruning and load balancing is effective. While ParArab and ParGFD$_n$ fail to complete GFD discovery, DisGFD works well on real-life graphs, and outperforms ParGFD$_{nb}$ by 1.31 times on average. (3) GFDs discovered by DisGFD can catch data inconsistency with higher accuracy when compared with AMIE rules.

8 CONCLUSION

We have formalized and studied the discovery problem for GFDs. The novelty of the work consists of (a) the fixed-parameterized complexity of three classical problems underlying GFD discovery, (b) a notion of support for GFDs, (c) algorithms for discovering GFDs and computing a cover of GFDs that guarantee the parallel scalability, and (d) new techniques for spawning and validating GFDs. Our experimental results have verified that our algorithms are scalable with graphs and are able to discover interesting GFDs.

We are extending DisGFD to discover other forms of graph dependencies, *e.g.,* GFDs with built-in comparison predicates and arithmetic expressions [17]. Another topic is to adapt the algorithm to knowledge bases, adopting the support and confidence of [36].

REFERENCES

[1] Dbpedia. *http://wiki.dbpedia.org/Datasets*.
[2] IMDB. *http://www.imdb.com/interfaces*.
[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[4] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *SPAA*, 2003.
[5] W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in RDF. In *SDKB*, pages 23–39, 2010.
[6] D. Calvanese, W. Fischl, R. Pichler, E. Sallinger, and M. Simkus. Capturing relational schemas and functional dependencies in RDFS. In *AAAI*, 2014.
[7] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In *SIGMOD*, pages 835–846, 2016.
[8] F. Chiang and R. Miller. Discovering data quality rules. In *VLDB*, 2008.
[9] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.
[10] A. Cortés-Calabuig and J. Paredaens. Semantics of constraints in RDFS. In *AMW*, pages 75–90, 2012.
[11] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theor. Comput. Sci.*, 141(1&2):109–131, 1995.
[12] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.
[13] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.
[14] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, volume 95, pages 632–640, 1995.
[15] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(1), 2008.
[16] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.
[17] W. Fan, X. Liu, P. Lu, and C. Tian. Catching numeric inconsistencies in graphs. In *SIGMOD*, 2018.
[18] W. Fan and P. Lu. Dependencies for graphs. In *PODS*, 2017.
[19] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.
[20] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, 2016.
[21] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.
[22] L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.
[23] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD workshop*, 2011.
[24] B. He, L. Zou, and D. Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In *SWIM*, pages 1–7, 2014.
[25] J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. In *FoIKS*, 2014.
[26] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *SIGKDD*, 2004.
[27] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.
[28] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, 2000.
[29] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(01):75–105, 2013.
[30] Y. Ke, J. Cheng, and J. X. Yu. Efficient discovery of frequent correlated subgraph pairs. In *ICDM*, 2009.
[31] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[32] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri. Test-driven evaluation of linked data quality. In *WWW*, 2014.
[33] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *TCS*, 71(1):95–132, 1990.
[34] G. Lausen, M. Meier, and M. Schmidt. SPARQLing constraints for RDF. In *EDBT*, pages 499–509. ACM, 2008.
[35] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in MapReduce. In *ICDE*, 2014.
[36] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *CIDR*, 2015.
[37] P. Shelokar, A. Quirin, and Ó. Cordón. Three-objective subgraph mining using multiobjective evolutionary programming. *JCSS*, 80(1):16–26, 2014.
[38] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
[39] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.
[40] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. In *DISC*, 2013.
[41] C. Wyss, C. Giannella, and E. Robertson. FastFDs: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *DaWaK*, 2001.
[42] Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *ISWC*, pages 794–809. 2011.

Acknowledgments. Fan, Liu and Lu are supported in part by 973 Program 2014CB340302, ERC 652976, NSFC 61602023 and 61421003, Beijing Advanced Innovation Center for Big Data and Brain Computing, EPSRC EP/M025268/1, joint-lab between Edinburgh and Huawei, and an Innovative Research Grant from Huawei. The authors thank Yingjie Cao for helping with some of the experiments. Liu is the corresponding author of the paper.