Discovering Graph Functional Dependencies

Wenfei Fan[1,2], Xueli Liu[3], Ping Lu[2], Yinghui Wu[4], Jingbo Xu[1,2], Luara Chen[5], Demai Ni[5]
[1]University of Edinburgh [2]Beihang University [3]Harbin Institute of Technology
[4]Washington State University [5]Huawei America Research Center
{wenfei@inf, jingbo.xu@}ed.ac.uk, xueli.hit@gmail.com, luping@buaa.edu.cn, yinghui@eecs.wsu.edu, {Yu.Chen1, Demai.Ni}@huawei.com

Abstract

This paper studies discovery of GFDs, a class of functional dependencies defined on property 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, striking a balance between the complexity and interestingness. 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, with performance guarantee to reduce running time with the increase of processors. Using real-life and synthetic data, we experimentally verify the effectiveness and scalability of the algorithms.

1. Introduction

Functional dependencies have recently been studied for property graphs [19, 21], referred to as *graph functional dependencies* and denoted by 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. As remarked in [21], GFDs are useful 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. Inconsistencies are commonly found in real-world knowledge bases, such as those depicted in Fig. 1.

(a) YAGO3 [38]: A person John Winter is given credit for producing film Selling Out, as shown in graph G_1 of Fig. 1, although the film was created by producer Jack Winter, while John is a high jumper, not a producer.

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

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

GFDs are able to catch these inconsistencies.

(1) Consider GFD $\varphi_1 = Q_1[x, y](y.\mathsf{type} = \text{“film”} \to x.\mathsf{type} = \text{“producer”})$. Here Q_1 is a graph pattern 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 inconsistency in G_1.

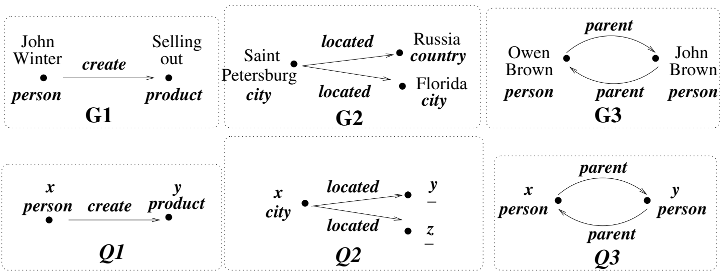


Figure 1: Graphs and graph patterns

(2) Consider now GFD $\varphi_2 = Q_2[x, y, z](\emptyset \to y.\mathsf{name} = z.\mathsf{name})$, where pattern Q_2 is shown in Fig. 1, $\emptyset$ denotes an empty set of literals, and name is an attribute. It ensures 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 $\varphi_3 = Q_3[x, y](\emptyset \to \mathsf{false})$, 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; it catches the inconsistency in G_3. □

To make practical use of GFDs, however, we need effective algorithms to discover interesting GFDs from real-life graphs. This is challenging. As will be seen shortly, a GFD $Q[\bar{x}](X \to Y)$ is a combination of a topological constraint Q and a functional dependency (FD) $X \to Y$, *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 functional dependencies (FDs) [28, 43], as GFDs additionally require topological constraints. It is more challenging than graph pattern mining [14, 20, 27, 29, 37, 39], 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 [21], which are embedded in GFD discovery but are not an issue for graph pattern mining.

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”; implication is to decide whether a GFD discovered is “redundant”, *i.e.,* implied by a set GFDs already known; and validation is to ensure that GFDs discovered from a (possibly fragmented and distributed) graph G are indeed satisfied by G.

We show that the implication and satisfiability problems are fixed-parameter tractable [22], but the validation problem is co-W[1]-hard [12]. 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 develop practical discovery algorithms.

(2) We formalize the discovery problem for GFDs (Section 4). We introduce a notion of topological support for *positive and negative* GFDs in graphs, and define reduced GFDs, which are a departure from their conventional counterparts. We show that this notion of support is anti-monotonic. Based on these, we formalize the discovery problem for GFDs, to strike a balance between the complexity of the discovery problem and the interestingness of GFDs discovered.

(3) We develop a sequential algorithm for discovering GFDs (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 to reduce the cost. 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 [35] 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 verify the scalability and effectiveness of 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.

The algorithms yield a promising tool for exploring interesting GFDs in real-life graphs, which can express "axioms" for knowledge bases [2]. We provide new techniques for parallel discovery and verification of graph dependencies, *e.g.,* vertical and horizontal GFD spawning, handling of negative GFDs, and methods to achieve parallel scalability.

Proofs of the results of the paper can be found in [2].

Related work. We categorize related work as follows.

FDs *for graphs.* FDs have been studied for RDF [6, 8, 11, 25, 26, 36, 44]. The study started from [36], which extends FDs to relational encoding of RDF, and interprets the "scope" of an FD with a class type that corresponds to a relation name. Based on triple patterns with variables, [6, 11] define FDs with triple embedding and homomorphism. The FDs were extended in [44] to relations of arbitrary arity. The implication and satisfiability problems for the FDs are shown decidable [6], but their complexity bounds are open; axiom systems are provided [11, 26] via relational encoding of RDF. Using clustered values, [44] defines FDs with path patterns; [44] is extended to support CFDs (conditional functional dependencies [17]) for RDF [25]. FDs are also defined by mapping relations to RDF [8], using tree patterns.

This work adopts GFDs of [21] 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 [8, 25, 44]. (c) GFDs support bindings of semantically related values like CFDs [17], *e.g.,* φ_1, and a negative form with false, *e.g.,* φ_3, which cannot be expressed as the FDs of [6, 8, 11, 25]. The need for supporting these is evident in consistency checking, as indicated by axioms studied for knowledge bases [2], and as verified by the experience of cleaning relational data [17].

GFDs were recently extended in [19] by supporting id-based keys and graph homomorphism, referred to as GEDs. While [19] studied the chase, complexity and axiom system for GEDs, it did not consider the discovery problem.

Dependency discovery. Discovery algorithms have been well studied for relational dependencies, *e.g.,* FDs [28, 43], CFDs [9, 18] and denial constraints [10]. As remarked earlier, GFD discovery is much harder. Closer to this work are algorithms for discovering FDs over graphs [25, 44]. The method of [44] first pre-clusters property values; it then adapts the levelwise process of TANE [28] to discover FDs defined with path patterns over RDF. It is extended in [25], which first enumerates frequent graph structures, and then adopts CFDMiner [18] 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 subgraph isomorphic mappings and is inherently intractable, as opposed to path patterns [25, 44], (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 [2], (c) dependencies whose validation is intractable, (d) topological support and reduced dependencies, (e) parallel discovery algorithms, not to mention parallel scalability as performance guarantees, and (f) pattern mining and attribute-dependency discovery in a single process. GFD discovery is unique in these aspects.

Graph pattern mining. Related to GFD discovery is graph pattern mining from graph databases [27, 29, 31]. Apriori [29] and pattern-growth [27] methods expand a frequent pattern by adding nodes and edges. [31] connects two graphs in a graph database via Pearson correlation. Multiobjective subgraph mining [39] optimizes subgraphs with single-graph metrics (*e.g.,* size, support) via skyline processing. As observed in [30], pattern mining over graph databases does not help GFD discovery, since the anti-monotonicity of the support of [27, 29, 31] no longer holds over a single graph.

Closer to this work are mining techniques for a single graph [14, 20, 37, 41]. GRAMI [14] considers patterns without edge labels, and models isomorphic subgraph enumeration as constraint satisfaction. The method of [37] mines frequent subgraphs via a two-step filter-and-refinement process, in MapReduce. Arabesque [41] uses "pattern-centric" MapReduce programming in pattern mining. It treats patterns and their potential embedding as key-values, and balances pattern verification in each MapReduce step. The method of [20] 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, not just pattern 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. For such GFDs, the computation of the support is more complex than a direct isomorphic counting of [14,20]. (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 [20,21]. Our algorithms differ from [20,21] in problems and methods. We balance the workload via distributed and incremental joins, while [20,21] require special treatments for skewed graphs.

2. Graph Functional Dependencies

We first review GFDs [21], 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 will use two notions of subgraphs. A graph $G' = (V', E', L', F'_A)$ is a *subgraph of* $G = (V, E, L, F_A)$ if $V' \subseteq V$, $E' \subseteq E$; 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 directed 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 pattern 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 entity connected to a product entity 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 with '_'; and (3) Q_3 depicts a pattern of person entities. □

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 $\ell = \ell'$.

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 (written as *iff*) $e' = (h(u), h(u'))$ is an edge in G' and moreover, $L'(e') \preceq L_Q(e)$.

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 $\bar{x}$.

Example 3: 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

Following [21], we define a *graph functional dependency* (GFD) as $Q[\bar{x}](X \to Y)$, 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. We refer to $x.A = c$ as a *constant literal*, and $x.A = y.B$ as a *variable literal.*

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 the matches of Q. Constant literals $x.A = c$ enforce bindings of semantically related constants, like CFDs [17].

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 literal $x.A = y.B$. We denote by $h(\bar{x}) \models X$ if $h(\bar{x})$ satisfies *all* the literals in X; similarly for $h(\bar{x}) \models Y$.

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 $\varphi \in \Sigma$, $G \models \varphi$.

Given the semantics, we can write $Q[\bar{x}](X \to Y)$ as

$$Q[\bar{x}]\big(\textstyle\bigwedge_{l \in X} l \to \bigwedge_{l' \in Y} l'\big).$$

Observe that to check whether $G \models \varphi$, we need to examine all matches of Q in G. In addition,

1. for a literal $x.A = c$ in X, if $h(x)$ has *no* attribute A, then $h(\bar{x})$ trivially satisfies $X \to Y$. That is, node $h(x)$ is not required to have attribute A, to accommodate the semi-structured nature of graphs.
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 above; similarly for $x.A = y.B$.
3. 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.

Example 4: In Fig. 1, $G_2 \not\models \varphi_2$. Indeed, a match of Q_2 in G_2 is h_2 given in Example 3, 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 $G_3 \not\models \varphi_3$. □

Positive and negative GFDs. 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 $h(\bar{x}) \models X$. The GFD is called *positive* otherwise.

For instance, φ_3 is negative, while φ_1 and φ_2 are positive.

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".

The notations of the paper are summarized in Table 1.

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

Table 1: Notations

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) for some GFD $Q[\bar{x}](X \to Y)$ in Σ, Q has a match in G. The *satisfiability problem* for GFDs is to decide whether a given set Σ of GFDs is satisfiable.

Intuitively, this is to check whether discovered GFDs are meaningful. To ensure that the GFDs can be applied to nonempty graphs, we require that there exists at least one GFD $Q[\bar{x}](X \to Y)$ in Σ such that Q finds a match in G.

(2) A set Σ of GFDs *implies* another GFD φ, denoted by $\Sigma \models \varphi$, if for all graphs G, $G \models \Sigma$ implies $G \models \varphi$. 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 to eliminate redundant GFDs.

(3) The *validation problem* is to decide, given a set Σ of GFDs and a graph G, whether $G \models \Sigma$.

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 [21]. 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.,* [12] 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.

For a set Σ of GFDs, we use k to denote $\mathsf{max}(|\bar{x}|)$ for all $Q[\bar{x}](X \to Y)$ 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: $k = \mathsf{max}\{|\bar{x}| \mid Q[\bar{x}](X \to Y) \in \Sigma \cup \{\varphi\}\}$.
∘ 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 all parameterized problems that are FPT-reducible to a certain weighted satisfiability problem (see [12]). It is conjectured that W[1]-hard problems are not fixed-parameter tractable. This tells us that the validation analysis remains nontrivial when k is small.

Proof: (a) We develop an algorithm for the satisfiability analysis of k-bounded GFDs with complexity $O(g(|\Sigma|) \cdot f(k))$, where g is a polynomial and f is exponential; similarly for implication. The algorithms are based on characterizations of GFD satisfiability and implication analyses given in [21].

(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 [13]. 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. The reduction is nontrivial and can be found in [2]. □

k-bounded GFDs. Not all is lost. For a constant k, we say that a pattern $Q[\bar{x}]$ is *k-bounded* if $|\bar{x}| \leq k$. A set Σ of GFDs is *k-bounded* if for all $Q[\bar{x}](X \to Y)$ in Σ, $Q[\bar{x}]$ is k-bounded.

It often suffices to consider k-bounded GFDs in practice with a small k. 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 4 nodes and 5 edges [24]. This is also witnessed by GFDs expressing axioms in knowledge bases [2]. As pattern (query) verification is typically a crucial step in rule mining [23], the size of practical queries indicates a reasonable k.

Better still, the three problems above become tractable for k-bounded GFDs (see proof in [2] for interested readers).

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

Proof: For the satisfiability problem, we give an $O(|\Sigma|^3 \times k^k)$-time algorithm, based on a characterization given in [21]. The algorithm is in PTIME when k is a constant. Similarly we verify the tractability of the implication problem. For validation, it suffices to do the following: for each GFD $Q[\bar{x}](X \to Y)$ in Σ, enumerate all matches $h(\bar{x})$ of Q in G and check whether $h(\bar{x}) \models X \to Y$; this yields an $O(|\Sigma| \cdot |G|^k)$ time algorithm, in PTIME for constant k. □

4. The Discovery Problem

We next formalize the discovery problem for GFDs. Given a graph G, we want to find a set Σ of "interesting" GFDs such that $G \models \Sigma$. Its unique challenges consist of the following: (a) GFD discovery is a combination of frequent graph pattern mining and dependency discovery; (b) for a negative GFD $\varphi = Q[\bar{x}](X \to \mathsf{false})$, Q may not find a match in G, *e.g.,* Q_3 of Fig. 1 in a "consistent" knowledge base; and (c) wildcard '_' makes it harder to discover "most general" patterns.

To cope with these, we introduce supports for positive GFDs in Section 4.1, and for negative GFDs in Section 4.2. We formulate the discovery problem in Section 4.3.

4.1 Interestingness Measure for Positive GFDs

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

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 [14,30], 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 propose a notion of support for GFDs in terms of the support of its pattern and correlation of its attributes.

Normal form. We consider positive GFDs of the form $\varphi = Q[\bar{x}](X \to l)$, where l is a literal, *i.e.,* Y in φ consists of a single l. This does not lose generality, as argued below.

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.

One can verify that a positive GFD $Q[\bar{x}](X \to Y)$ is equivalent to a set of GFDs $Q[\bar{x}](X \to l)$ for each $l \in Y$. In the sequel for positive GFDs, we consider the normal form only.

We also focus on nontrivial GFDs. A GFD φ 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, trivial GFDs are not interesting.

Support. We first define the support of Q. Pattern Q may be disconnected, in the form of $(Q_1, \ldots, Q_m)$, where Q_i's are all maximum connected components of Q, by treating Q as an undirected graph. We pick a list $\bar{z} = (z_1, \ldots, z_m)$, where for each $i \in [1, m]$, z_i is a variable in $\bar{x}$ such that $\mu(z_i)$ is a node in Q_i, where μ is the mapping from variables to nodes in Q. In practice, z_i may indicate user specified interests. We assume *w.l.o.g.* an order on variables and take the minimum variable z_i following this order for each Q_i. We refer to such $\bar{z}$ as *the pivot of* φ, denoted by $\bar{z}$ in $\bar{x}$.

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

Pattern support. Consider a graph G, and a GFD φ with pattern $Q[\bar{x}]$, where Q has pivot $\bar{z} = (z_1, \ldots, z_m)$. Denote by $Q(G, \bar{z})$ the set of nodes that match $\bar{z}$ induced by $h(\bar{z})$ for all matches h of Q in G. We define the *support* of Q as:

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

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

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

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

Here $Q(G, Xl, \bar{z})$ is the subset of $Q(G, \bar{z})$ with $h(\bar{x}) \models X$ 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 $Q(G, X \to l, \bar{z}) = Q(G, \bar{z})$ as long as $G \models \varphi$, where $Q(G, X \to l, \bar{z})$ is the subset of $Q(G, \bar{z})$ with $h(\bar{x}) \models X \to l$. That is, it does not accurately measure the correlation of X and l.

In particular, we define $\rho(\varphi, G) = 0$ if φ is trivial.

Support of GFDs. We define the *support of* φ *in* G as

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

Anti-monotonicity. We next justify that the support and correlation defined above in terms of anti-monotonicity. To this end, we first define orders on patterns and GFDs.

Ordering patterns. Consider 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.

More specifically, $Q[\bar{x}] \ll Q'[\bar{x}']$ if there exists a bijective mapping f from Q to a subgraph of Q' such that (a) for each node $u \in V_Q$, $L'_Q(f(x)) \preceq L_Q(x)$; (b) $e = (u, u')$ is an edge in Q iff $e' = (f(u), f(u'))$ is an edge in G' and $L'_Q(e') \preceq L_Q(e)$; and (c) either f maps Q to a proper subgraph of Q', or there is a node x (resp. edge e) of Q such that $L'_Q(f(x)) \prec L_Q(x)$ (resp. $L'_Q(e') \prec L_Q(e)$; see Section 2.1 for $\prec$).

Ordering GFDs. Consider $\varphi_1 = Q_1[\bar{x}_1](X_1 \to l_1)$ 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 Q_1 reduces Q_2 and X_1 reduces X_2. That is, there exists an isomorphism f from Q_1 to a subgraph of Q_2 such that (a) $f(\bar{z}_1) = \bar{z}_2$, where $\bar{z}_i$ 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 $f(X_1) \subsetneq X_2$.

Example 5: Recall $\varphi_1 = Q_1[x, y](X_1 \to l)$ (Example 1), where $X_1 = \{y.\mathsf{type} = \text{"film"}\}$, and l is $x.\mathsf{type} = \text{"producer"}$. Let $\bar{z} = \{x\}$. (1) Consider GFD $\varphi_1^1 = Q_1^1[x, y, z](X^1 \to l)$ with $\bar{z} = \{x\}$, 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 $X_1 \cup \{y.\mathsf{name} = \text{'Selling out'}\}$. Then $\varphi_1 \ll \varphi_1^1$, and $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_1^1, G)$. Indeed, every producer x induced from the match $Q_1^1(G, X_1^1 l, \bar{z})$ is a producer in $Q_1(G, X_1 l, \bar{z})$. (2) Consider GFD $\varphi_1^2 = Q_1^1[x, y, z](X_1^2 \to l)$, where X_1^2 consists of $y.\mathsf{name} = \text{'Selling out'}$. Then $\varphi_1 \not\ll \varphi_1^2$ since $X_1 \not\subseteq X_1^2$. □

We show that GFD support is anti-monotonic.

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

Proof: We prove this for positive GFDs here, and defer the case for negative GFDs to Section 4.2. 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 pivots $\bar{z}_1$ and $\bar{z}_2$, respectively. To see that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$, we show that $Q_2(G, X_2 l_2, \bar{z}_2) \subseteq Q_1(G, X_1 l_1, \bar{z}_1)$. That is, for any set $\bar{v}$ of nodes in G, if there exists a match h_2 of Q_2 in G such that $h_2(\bar{z}_2) = \bar{v}$, $h_2(\bar{x}_2) \models X_2$ and $h_2(\bar{x}_2) \models l_2$, then there exists a match h_1 of Q_1 in G such that $h_1(\bar{x}_1) = \bar{v}$, $h_1(\bar{x}_1) \models X_1$ and $h_1(\bar{x}_1) \models l_1$ (see [2] for details). □

4.2 Support of Negative GFDs

A negative $\varphi = Q[\bar{x}](X \to \mathsf{false})$ has two forms (see Section 2): (a) $X = \emptyset$, and hence $Q(G, \bar{z}) = \emptyset$ in a "consistent" graph G; and (b) $X \neq \emptyset$ and is satisfiable. Putting φ in the normal form $Q[\bar{x}](X \to l)$ by taking false as a literal l, in both cases, $Q(G, Xl, \bar{z}) = \emptyset$. Thus we can no longer discover negative GFDs by computing $Q(G, Xl, \bar{z})$ and $Q(G, \bar{z})$.

We are interested in negative GFD φ that results from a "minimal trigger" to a positive $Q'[\bar{x}'](X' \to l')$, 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 the support of negative φ as:

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

where (1) if $X = \emptyset$, Φ' consists of patterns $Q'[\bar{x}']$ with the same pivot $\bar{z}$ such that $\mathsf{supp}(Q', G) > 0$, and Q' is obtained from Q by removing an edge (possibly nodes too); and (2) if $X \neq \emptyset$, Φ' consists of positive GFDs $\varphi' = Q[\bar{x}](X' \to l)$ with the same pivot $\bar{z}$ 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, $\mathsf{supp}(\varphi, G)$ 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.

Anti-monotonicity. For negative GFDs φ_1 and φ_2, we say that $\varphi_1 \ll \varphi_2$ if (a) when $\varphi_1 = Q_1[\bar{x}_1](\emptyset \to \mathsf{false})$ and $\varphi_2 = Q_2[\bar{x}_2](\emptyset \to \mathsf{false})$, there exists a base Q'_i of φ_i for $i \in [1, 2]$ such that $Q'_1 \ll Q'_2$; and (b) when $\varphi_1 = Q_1[\bar{x}_1](X_1 \to \mathsf{false})$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to \mathsf{false})$, there exists a base φ'_i for $i \in [1, 2]$ such that $\varphi'_1 \ll \varphi'_2$. When $\varphi_1 = Q_1[\bar{x}_1](\emptyset \to \mathsf{false})$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to \mathsf{false})$ ($X \neq \emptyset$), the two are incomparable since they have different types of bases. Given this, Theorem 3 holds on generic GFDs, positive or negative.

Open World Assumption (OWA). The OWA states that absent data cannot be used as counterexamples, as adopted by knowledge bases [20, 23]. 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), counting no unknown data.

4.3 The Discovery Problem

We want to eliminate excessive GFDs that are not very interesting, *e.g.,* trivial GFDs (Section 4.1). In addition, we want minimum GFDs only, defined as follows.

(1) Minimum GFDs. We say that a positive GFD φ is *reduced* on graph G if $G \models \varphi$ but $G \not\models \varphi'$ for any GFD $\varphi' \ll \varphi$. It is *minimum on* G if it is nontrivial and reduced.

Observe that if a GFD $\varphi = Q[\bar{x}](X \to l)$ is reduced, then it is (a) *left-reduced* [9, 18, 28], *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 $Q'[\bar{x}'] \ll Q[\bar{x}]$, when $X \to l$ is defined on $Q'[\bar{x}']$ (with variable renaming).

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 $\varphi = Q'[\bar{x}](\emptyset \to \mathsf{false})$, or (b) adding a literal to X and obtaining $\varphi = Q[\bar{x}](X' \to \mathsf{false})$. That is, it is triggered by minimum change to ψ.

(2) 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 does not contain redundant or non-interesting GFDs.

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.,* $\mathsf{supp}(\varphi, G) \geq \sigma$.

We assume $k \geq 2$ since GFDs with a single-node are not interesting. 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 strike a balance between the complexity of discovery and the interestingness of GFDs discovered. Indeed, (a) as remarked in Section 3, k-bounded GFDs suffice to cover rules of real-life interests when k is fairly small; and (b) by Proposition 2, the implication and validation problems for GFDs are in PTIME when k is fixed.

To reduce excessive literals, we often select a set Γ of *active attributes* from G that are of users' interest. We only discover GFDs with literals composed of attributes in Γ.

5. Sequential GFD Discovery

We start with a sequential algorithm for GFD discovery, denoted as SeqDisGFD. Given a graph G, a number k and a threshold σ, SeqDisGFD finds a cover Σ_c of k-bounded minimum σ-frequent GFDs in G. 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.

To simplify the discussion, below we focus on GFDs with connected patterns. We show how the algorithms can be extended to handle GFDs with disconnected patterns in [2].

5.1 Sequential GFD Mining

A brute-force algorithm first enumerates all frequent patterns Q in G following conventional graph pattern mining (*e.g.,* [14, 41]), and then generates GFDs with Q by adding literals. However, enumeration of all k-bounded GFDs is rather 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 carrying 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 $X \to Y$. At each iteration i ($0 < i < k^2$), SeqDis generates and verifies GFD candidates, and fills the level-i part of tree T, 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.

GFD 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 $(Q[\bar{x}], \mathsf{lvec})$, where (a) $v.Q[\bar{x}]$ is a graph pattern with i edges, and (b) $v.\mathsf{lvec}$ is a vector,

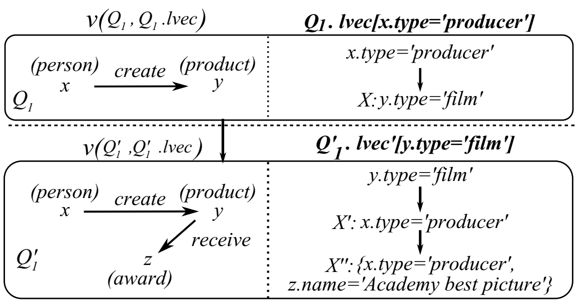


Figure 2: GFD **generation tree**

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, B are active attributes in Γ, and c is a constant in G. Each node at level j of $\mathsf{lvec}[l]$ is a literal set X such that $Q[\bar{x}](X \to l)$ is a GFD candidate. There exists an edge (X_1, X_2) in $v.\mathsf{lvec}[l]$ if $X_1 = X_2 \cup \{l'\}$ for a literal 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 Q' extends Q by adding a single edge.

Example 6: A fraction of GFD generation tree T is shown in Fig. 2. It contains a node $v(Q_1[\bar{x}], \mathsf{lvec})$ at level 1, and $v(Q'_1[\bar{x}], \mathsf{lvec})$ at level 2. Node $v(Q_1[\bar{x}], \mathsf{lvec})$ stores graph pattern $Q_1[\bar{x}]$ of Fig. 1, and its literal tree $Q_1.\mathsf{lvec}[l]$ rooted at $x.\mathsf{type}$ = "producer". At $v(Q'_1[\bar{x}], \mathsf{lvec})$, a literal tree is rooted at a different literal $y.\mathsf{type}$ = "film". There is an edge from X' to X'' in $Q'_1.\mathsf{lvec}$, as $X''{=}X' \cup \{z.\mathsf{name}$ = "Academy best picture"$\}$. There exists an edge from $v(Q_1[\bar{x}], \mathsf{lvec})$ to $v(Q'_1[\bar{x}], \mathsf{lvec})$ in T, since Q'_1 is obtained by adding a single edge (y, z) to Q_1. The literal at node X in $Q_1.\mathsf{lvec}$ encodes the GFD φ_1 of Example 1. Similarly, X'' in node $Q'_1.\mathsf{lvec}$ encodes GFD $\varphi_4 = Q'_1[x, y, z](\{x.\mathsf{type}$ = "producer", $z.\mathsf{name}$ = "Academy best picture"$\} \to y.\mathsf{type}$ = "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. Tree T spawns new GFD candidates by performing the following two "atomic" operations.

Vertical spawning. $\mathsf{VSpawn}(i)$ 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, $\mathsf{VSpawn}(i)$ adds new patterns to T, for i from 1 up to k^2 (k-bounded). 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 $h(x).A = c$ for $x \in \bar{x}$, taking A from Γ and constants c from G. VSpawn expands patterns with edges e' and HSpawn generates literals with the attributes and constants.

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 $i-1$. If Q'_1 is added to $\mathsf{iso}(Q_1)$, $P(Q'_1)$ is merged into $P(Q_1)$. Here $\mathsf{iso}(Q_1)$ is the set of patterns at level i that are isomorphic to Q_1.

Example 7: Consider the tree T of Fig. 2. When SeqDis executes $\mathsf{VSpawn}(2)$, it spawns a new pattern Q'_1 (among others) from Q_1, by adding an edge $e = (y, z)$. □

Horizontal spawning. $\mathsf{HSpawn}(i, j)$ 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.

That is, when $j = 0$, $\mathsf{HSpawn}(i, j)$ 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 does not add new patterns. We denote the set of GFD candidates generated by $\mathsf{HSpawn}(i, j)$ as $\Sigma_{C_{ij}}$.

Example 8: Continuing Example 7, $\mathsf{HSpawn}(2, j)$ is performed on the newly generated patterns at level 2 of T. For pattern Q'_1 and literal tree $Q'_1.\mathsf{lvec}$ rooted at $l = (y.\mathsf{type}$ = "film"), $\mathsf{HSpawn}(2, 2)$ extends X' at level $j = 1$ to X'' at $j = 2$, by adding $z.\mathsf{name}$ = "Academy best picture". This yields $\varphi_4 = Q_1[x, y, z](X'' \to l)$ (see Fig. 2). □

Upgrade. Algorithm SeqDis also "upgrades" GFDs at level i. For each $\varphi = Q[\bar{x}](X \to l)$ discovered, it examines its variants $\varphi' = Q'[\bar{x}](X' \to l')$ such that Q' is isomorphic to Q except for different node or edge labels, and X' (resp. l') is the same as X (resp. l) subject to variable renaming based on the isomorphism. It checks in the variants whether a label $L_Q(v)$ (resp. $L_Q(e)$) ranges over all labels of Γ; if so it upgrades φ to φ^- by replacing $L_Q(v)$ (resp. $L_Q(e)$) to '_', and substitutes φ^- for all the corresponding variants (see [2] for an example). The set $P(Q)$ of parents is also extended by merging all $P(Q')$ of corresponding variants.

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 $\mathsf{supp}(Q, G) < \sigma$.

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 $\mathsf{supp}(Q, G) < \sigma$, 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 $\mathsf{supp}(\varphi, G) = 0 < \sigma$. (b) After we check a GFD φ as stated in (b) above, HSpawn can safely stop generating φ'. Indeed, if $\mathsf{supp}(\varphi) \geq \sigma$, then φ is a candidate for Σ_c but φ' is not, since φ' is not reduced. If $\mathsf{supp}(\varphi) < \sigma$, then by Theorem 3, $\mathsf{supp}(\varphi') \leq \mathsf{supp}(\varphi)$ and hence φ' cannot make a candidate for Σ_c either. (c) If $Q \ll Q'$, then $\mathsf{supp}(Q, G) \geq \mathsf{supp}(Q', G)$ as verified in the proof of Theorem 3 (see [2]). Moreover, $\mathsf{supp}(\varphi', G) \leq \mathsf{supp}(Q', G)$ by the definition of $\mathsf{supp}(\varphi', G)$. Since $\mathsf{supp}(Q, G) < \sigma$, $\mathsf{supp}(\varphi', G) \leq \mathsf{supp}(Q', G) \leq \mathsf{supp}(Q, G) < \sigma$. That is, φ' cannot be in Σ_c, and can be pruned by VSpawn. □

Discovering negative GFDs. Unlike conventional rule 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) $Q[\bar{x}](\emptyset \to \mathsf{false})$, or (b) $Q[\bar{x}](X \to \mathsf{false})$ when $X \neq \emptyset$. 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 must be expanded from a pattern Q' by adding a single edge, where $\mathsf{supp}(Q', G) \geq \sigma$, since otherwise $\mathsf{supp}(\varphi, G) = \max \mathsf{supp}(Q', G) < \sigma$. Hence $\mathsf{NVSpawn}(i)$ is triggered by $\mathsf{VSpawn}(i)$ at iteration i, once all the level-i patterns $\mathcal{Q}_i$ are generated and verified. It finds all patterns $Q' \in \mathcal{Q}_i$ with $\mathsf{supp}(Q', \bar{z}) = 0$, and adds $\varphi = Q'[\bar{x}](\emptyset \to \mathsf{false})$ 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 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 $\varphi' = Q[\bar{x}](X' \to \text{false})$, where X' extends X with a single literal. It checks whether $Q(G, X', \bar{z}) = 0$, and adds φ' to Σ_N if so. It guarantees that $\text{supp}(\varphi', G) \geq \sigma$ due to the existence of φ.

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 a characterization of GFD implication [21], reviewed as follows.

Characterization (Lemma 7 of [21]). Given a set Σ of GFDs and $\varphi = Q[\bar{x}](X \to l)$, $\Sigma \models \varphi$ *if and only if* (a) there exists a set $\Sigma_Q \subseteq \Sigma$ such that each $\varphi' \in \Sigma_Q$ is embedded in Q, *i.e.,* there is an isomorphism from pattern Q' of φ' to a subgraph of Q, and (b) $l \in \text{closure}(\Sigma_Q, X)$ or $\text{closure}(\Sigma_Q, X)$ is conflicting. Here $\text{closure}(\Sigma_Q, X)$ is the set of literals deduced by applying Σ_Q to Q and by the transitivity of equality literals in X. It 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.

Algorithm SeqCover. Making use of the characterization and following the algorithms for computing cover of relational FDs (see, *e.g.,* [4]), 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.

5.3 Analysis of Sequential GFD Discovery

One can verify that algorithm SeqDis correctly generates and validates all k-bounded σ-frequent minimum GFDs, and that algorithm SeqCover correctly computes a cover, following the characterization of GFD implication [21].

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" [15]. This is due to the intractable nature of the problem, unless $\mathsf{P} = \mathsf{NP}$. Thus SeqDis takes $O(\mathsf{C}(k, G) \cdot |G|^k)$ time in the worst case, denoted by $t_1(|G|, k, \sigma)$.

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

Taken together, the overall cost of 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 that makes use of the best existing algorithm for 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* [35]. 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

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

where the notation $\tilde{O}$ hides $\log(n)$ factors (see, *e.g.,* [42]). We assume that $n \ll |G|$ as commonly found in practice.

Intuitively, parallel scalability guarantees speedup of $\mathcal{A}_p$ *relative to* a "yardstick" sequential algorithm [35]. A parallel scalable $\mathcal{A}_p$ "linearly" reduces the sequential cost of $\mathcal{A}$.

The main result of this section is as follows.

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 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. As a proof of Theorem 5, we provide such a DisGFD. DisGFD 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.

Both algorithms work with a coordinator S_c and n workers, on a graph G partitioned into n fragments $(F_1, \ldots, F_n)$ and distributed across n workers $(P_1, \ldots, P_n)$. We adopt vertex cut [32] to evenly partition G. We assume a set Γ of active attributes from G, selected by the users.

Parallel scalability. We will show the following: (a) ParDis is in $\tilde{O}(\frac{t_1(|G|, k, \sigma)}{n})$ time, where $t_1(|G|, k, \sigma)$ is the time complexity of SeqDis (Section 5.1) and is measured in terms of the number of GFD candidates φ to be checked and the cost of validating φ (Section 5.3), and (b) ParCover can plug in any sequential algorithm $\mathcal{A}_c$ for computing cover Σ_c of Σ, and parallelizes the process of redundancy removal by balancing the workload across n processors. When we plug in SeqCover of Section 5.2, ParCover is in $\tilde{O}(\frac{t_2(\Sigma, k)}{n})$ time, where $t_2(\Sigma, k)$ is the cost of SeqCover (Section 5.3).

We study GFDs with connected patterns, and present the handling of GFDs with disconnected patterns in [2].

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)

```
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 $i := 1$; $\mathsf{flag}_V :=$ true;
2.  SpawnGFD(T); /* initialize $T$ with single-node GFDs;
3.  while $i \leq k^2$ and $\mathsf{flag}_V$ do /* superstep $i$ */;
4.    VSpawn($i$); $\Sigma_i := \emptyset$;
5.    $\mathsf{flag}_V :=$ false if no new pattern is spawned;
6.    if $\mathsf{flag}_V$ then
7.      parallel graph pattern matching;
8.      $j := 1$; $\mathsf{flag}_H :=$ true;
9.      while $j \leq J$ and $\mathsf{flag}_H$ do
10.       set $\Sigma_{C_{ij}} :=$ HSpawn($i, j$);
11.       $\mathsf{flag}_H :=$ false if no new GFD candidates are spawned;
12.       if $\mathsf{flag}_H$ 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

and GFD validation (horizontal spawning) of SeqDis, respectively. More specifically, it performs the following.

(1) Parallel pattern verification. Algorithm ParDis performs vertical spawning VSpawn(i) (Section 5.1) at coordinator 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 HSpawn(i, j) (Section 5.1) at S_c with the verified graph patterns to generate a set of GFD candidates. Operation 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.,* $i = k^2$). 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 back 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 partition the edges of G evenly 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. 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 P_s. Otherwise we redistribute $Q(F_s) \bowtie e(F_t)$ evenly across workers and compute it in parallel (see more details in [2]).

Parallel validation. Once all workers return Terminate to S_c, ParDis starts to perform HSpawn(i, j) to generate GFD candidates $\Sigma_{C_{ij}}$ 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, 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 VSpawn(i).

(2) When all P_s's complete their local validation, for each GFD $\varphi \in \Sigma_{C_{ij}}$, ParDis checks at coordinator S_c whether $\mathsf{supp}(\varphi, G) = \sum_{s\in[1,n]} \mathsf{supp}(\varphi, F_s) \geq \sigma$, and $\bigwedge_{s\in[1,n]} \mathsf{SAT}^s_\varphi =$ true. If so, it adds φ to Σ_i as a verified frequent GFD. It also upgrades GFDs at S_c triggered by newly discovered ones.

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 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 analyze the computation and communication 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 (see [2] for a proof). 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 Lemma 4, and (b) incremental pattern matching reduces unnecessary recomputation.

6.3 Parallel Cover Computation

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

Group checking. Recall that algorithm SeqCover (Section 5.2) inspects each GFD in Σ one by one. In contrast, algorithm ParCover parallelizes the process by leveraging the characterization of GFD implication. (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.

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 $\Sigma_c := \emptyset$; 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

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. Then one can verify the following based on the characterization of [21].

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 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 $P(Q_j)$ 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}.

ParCover 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). As verified in [2]: (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 $\mathsf{receive}(y, z)$ in Fig 2, and φ_5 (resp. φ_6) has a pattern of one edge $\mathsf{parent}(x, y)$ (resp. $\mathsf{parent}(y, x)$) 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 pattern Q'_1 of Fig 2. Then $\varphi_1, \varphi'_1, \varphi_4$ are embedded in Q'_1, and φ_3, φ_5, φ_6 are embedded in Q_3.

To compute a cover Σ_c of Σ, ParCover first partitions Σ to groups $\Sigma^{Q_1}{=}\{\varphi_1\}$, $\Sigma^{\mathsf{receive}(y,z)}{=}\{\varphi'_1\}$, $\Sigma^{Q_3}{=}\{\varphi_3\}$, $\Sigma^{Q'_1}{=}\{\varphi_4\}$, $\Sigma^{\mathsf{parent}(x,y)}{=}\{\varphi_5, \varphi_6\}$. It then constructs the following work units: $\Sigma_{Q_1} = \{\varphi_1\}$, $\Sigma_{\mathsf{receive}(y,z)} = \{\varphi'_1\}$, $\Sigma_{\mathsf{parent}(x,y)} = \{\varphi_5, \varphi_6\}$, $\Sigma_{Q_3} = \{\varphi_3, \varphi_5, \varphi_6\}$ and $\Sigma_{Q'_1} = \{\varphi_1, \varphi'_1, \varphi_4\}$. The checking "breaks down" to 5 independent tests that are distributed to all workers and conducted in parallel. For example, 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 [5] (see [2]).

Parallel scalability. Suppose that $\Sigma = \{\varphi_1, \ldots, \varphi_M\}$. We next 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 $\Sigma_{Q_i} \subseteq \Sigma$, 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 $t_2(\Sigma, k) = T(\Sigma, \varphi_1) + \ldots + T(\Sigma, \varphi_M)$ (see Section 5.2). 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.

Putting this and the analysis of ParDis (Section 6.2) together, we can see that algorithm DisGFD for GFD discovery 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 sequential SeqDisGFD. This completes the proof of Theorem 5.

7. Experimental Study

Using real-life and synthetic graphs, we conducted five sets of experiments to evaluate our discovery algorithms for (1) the parallel scalability with the increase of workers, (2) the scalability with graphs, (3) the impact of bound k and support threshold σ, and (4) the effectiveness of finding useful GFDs. We also evaluated (5) the parallel scalability and scalability of the algorithms for computing a cover of GFDs.

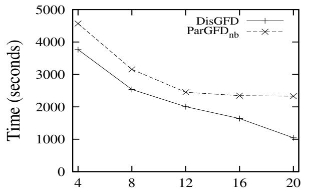
Experimental setting. We used three real-life graphs: (a) *DBpedia*, a knowledge graph [1] with 1.72 million entities of 200 types and 31 million links of 160 types, (b) *YAGO2*, an extended knowledge base of YAGO [40] with 1.99 million nodes of 13 types, and 5.65 million links of 36 types; and (c) *IMDB* [3], a knowledge base with 3.4 million nodes of 15 types, and 5.1 million edges of 5 types. We sampled 5 most frequent attributes for each entity as Γ in each dataset, and 5 most frequent values for each active attribute $A \in \Gamma$.

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.

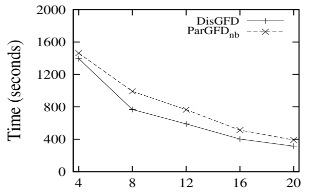
GFD *generator*. To test the scalability of GFD implication, we developed a generator to produce sets Σ of GFDs, controlled by $|\Sigma|$ (up to 10000) and k (up to 6). It follows VSpawn and HSpawn to generate GFDs with frequent edges and values drawn from real-life graphs, using the same active attribute set Γ for GFD discovery. (1) At each superstep i, it invokes VSpawn to generate m "seed" graph patterns with i edges; and (2) for each "seed" pattern Q, it invokes HSpawn to produce m' GFDs that pertain to Q.

Algorithms. We implemented the following, all in Java: (1) sequential SeqDisGFD, including SeqDis and SeqCover; (2) DisGFD for parallel GFD discovery, 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$_n$, a version of ParCover without GFD grouping (Lemma 6).

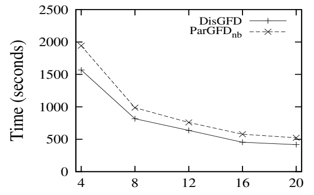
To evaluate the impact of integrating pattern mining and dependency discovery in DisGFD, we also implemented a



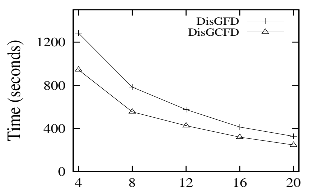
(a) Varying n (DBpedia)



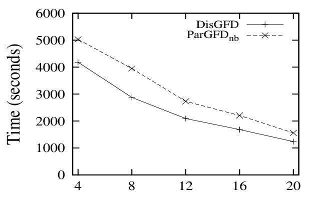
(b) Varying n (YAGO2)



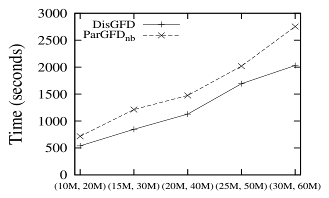
(c) Varying n (IMDB)



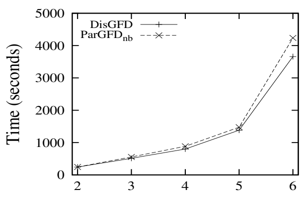
(d) GCFD vs. GFD (YAGO2)



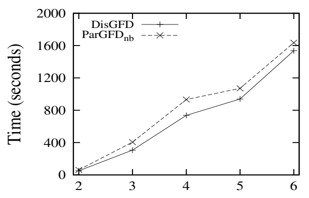
(e) Varying n (Synthetic)



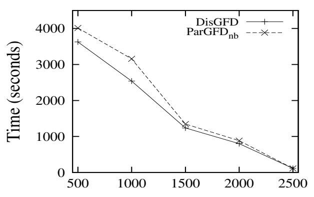
(f) Varying $|G|$ (Synthetic)



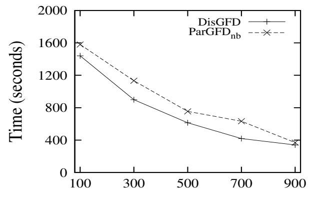
(g) Varying k (DBpedia)



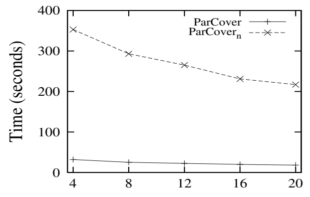
(h) Varying k (YAGO2)



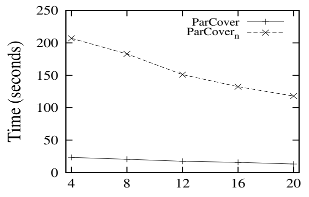
(i) Varying σ (DBpedia)



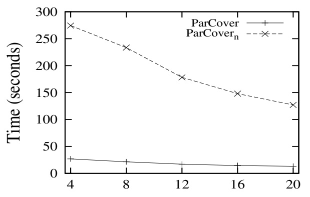
(j) Varying σ (YAGO2)



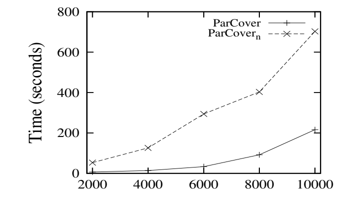
(k) Varying n (DBpedia)



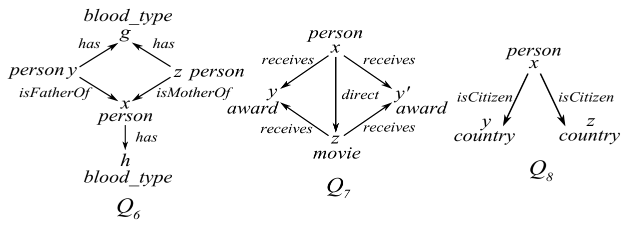
(l) Varying n (YAGO2)



(m) Varying n (Synthetic)



(n) Varying $|\Sigma|$ (Synthetic)



(o) Real-life GFDs (YAGO2)

Figure 5: Performance evaluation of parallel GFD discovery

discovery algorithm ParArab that splits the two processes as follows. (a) It first discovers all frequent patterns Q (adapted to the support $\mathsf{supp}(Q, G)$) in parallel, by using Arabasque [41], 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, in a separate process. It uses the same procedure ParCover for implication.

We also developed algorithm ParCGFD for mining GCFDs, an extension of relational CFDs [17] with path patterns [25], which makes a special case of GFDs.

We set the value of the support threshold σ such that the induced support of their patterns is comparable to the counterparts used in frequent graph patterns [41].

We deployed the 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 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 our load balancing strategy. (3) DisGFD is feasible on real-life graphs: 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) takes 350 and 416 seconds on *YAGO2* and *IMDB*, respectively, when n=20. (3) It outperforms ParGFD$_{nb}$ by 1.2 and 1.3 times on *YAGO2* and *IMDB*, respectively. The benefit of load balancing of DisGFD is more evident on *DBpedia* since it is much denser than *YAGO2* and *IMDB*, and yields far more graph patterns to be balanced.

We also compared DisGFD with ParCGFD on *YAGO2*.

datasets	GFDs: #/supp	GCFDs: #/supp
YAGO2	182/605	104/698
DBpedia	97/1724	65/1278

Figure 6: Number and support of GFDs

As shown in Fig. 5(d), DisGFD performed comparably to ParCGFD, although DisGFD finds far more GFDs and those GFDs have more complex patterns than GCFDs found by ParCGFD. The results on the other datasets are consistent.

We report the number and average support of mined rules on two real-life knowledge-base datasets in Fig. 6.

Synthetic graph. We also report the performance of DisGFD over a larger synthetic graph with 30 million nodes and 60 million edges. The result is consistent, as shown in Fig. 5(e).

Parallel vs. sequential. We report the performance of sequential GFDs mining in Table 7. While both ParGFD$_n$ and ParArab fail to complete, SeqDisGFD performs reasonably well: it takes 1.3 hours to discover GFDs from *YAGO2* with 7.64 million entities and edges. The performance is further improved by ParDis. When $n = 4$, DisGFD improves the performance of SeqDisGFD by 3.6 times on average.

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(f), (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 to discover GFDs in G of size $(30M, 60M)$.

As indicated in Table 7, the impact of $|G|$ on SeqDisGFD is consistent: the larger G is, the longer SeqDisGFD takes.

Exp-3: Impact of parameters. We next evaluated the impact of pattern size k and support threshold σ.

Varying k. Fixing $n = 8$ and $\sigma = 1000$ (resp. 500) on *DBpedia* (resp. *YAGO2*), we varied k from 2 to 6, *i.e.,* for GFDs with patterns having up to 6 nodes and 36 edges, larger than patterns commonly used in practice [24]. As shown in Fig. 5(g) over *DBpedia*, (1) it takes both DisGFD and ParGFD$_{nb}$ longer to find GFDs with larger patterns, as expected; (2) DisGFD outperforms ParGFD$_{nb}$ by 1.2 times; and (3) it is feasible for DisGFD to identify GFDs with reasonably large patterns: it takes 20 minutes to find 5-bounded GFDs. Figure 5(h) shows consistent results on *YAGO2*.

Varying σ. Fixing $n = 8$ and $k = 4$, we varied σ from 500 to 2500 (resp. from 100 to 900), and report the performance of DisGFD and ParGFD$_n$ on *DBpedia* (resp. *YAGO2*) in Fig. 5(i) (resp. Fig. 5(j)). We find that both algorithms take less time with larger σ, because higher σ prunes more GFD candidates, and reduces generation and verification time. This again verifies the effectiveness of our pruning strategy.

The impacts of k and σ on SeqDisGFD is consistent.

Exp-4: Cover computation. In the same setting as Figures 5(a)-5(e), we report the scalability of ParCover on *DBpedia*, *YAGO2* and synthetic G in Figures 5(k)-5(m), 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 Table 7, the sequential SeqCover also does well. It takes at most 45.1 seconds to compute the cover of GFDs discovered from 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(n), 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 Table 7.

algorithms	*DBpedia*	*YAGO2*	*IMDB*
SeqDisGFD	14322.4s	4963.2s	5723.8s
SeqCover	45.1s	32.5	30.1s

Figure 7: Performance of sequential discovery

Exp-5: Effectiveness. As examples, we give 3 GFDs found in *YAGO2* by DisGFD, with patterns shown in Fig. 5(o).

GFD$_1$: $Q_6[x, y, z, g, h](g.\mathsf{val} = \text{“O”} \rightarrow h.\mathsf{val} = \text{“O”})$. It says that when both parents have blood type “O”, then “O” must be the blood type of their children.

GFD$_2$: $Q_7[x, y, z, y'](y.\mathsf{name} = \text{“Gold Bear”} \wedge y'.\mathsf{name} = \text{“Gold Lion”} \rightarrow \mathsf{false})$. 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 have “initial release” at their festivals.

GFD$_3$: $Q_8[x, y, z](X_8 \rightarrow \mathsf{false})$, a negative GFD, where X_8 consists of $y.\mathsf{name} = \text{“US”}$ and $z.\mathsf{name} = \text{“Norway”}$. It is obtained by expanding a positive $Q_8[x, y, z](y.\mathsf{name} = \text{“US”} \rightarrow x.\mathsf{livesIn} = \text{“US”})$ by adding a literal (see Section 5.1).

Observe that these GFDs may bear a DAG pattern, and carry constants or false, beyond the capacity of most FD proposals for graphs. These GFDs 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.

Summary. From the experiments, we find the following. (1) DisGFD is parallel scalable. On real-life graphs, it is 3.78 times faster on average when n is increased from 4 to 20. (2) GFD discovery is feasible in practice. It takes 591 seconds for DisGFD to find frequent 4-bounded GFDs from real-life graphs on average, and less than 30 minutes from synthetic graphs with 30M nodes and 60M edges, when $n = 20$. SeqDisGFD and SeqCover are also feasible on real-life graphs (*e.g.,* 1.3 hours and 1 minutes on *YAGO2*, respectively). (3) Our integrated method with pruning and load balancing is effective. While ParArab and ParGFD$_n$ fail to complete GFD discovery, DisGFD performs well on real-life graphs, and outperforms ParGFD$_{nb}$ by 1.31 times on average. (4) Parallelization speeds up GFD cover computation. On real-life graphs, ParCover is on average 1.75 times faster when n is varied from 4 to 20; and the grouping strategy improves its performance by 10 times. (5) GFDs discovered by DisGFD can catch errors and suggest knowledge enrichment.

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.

The study of dependencies for graphs is still in its infancy. Other forms of graph dependencies need to be investigated, from formulation to discovery and applications.

9. References

[1] Dbpedia. *http://wiki.dbpedia.org/Datasets.*

[2] Full version. *http://homepages.inf.ed.ac.uk/s1400132/dis.pdf.*

[3] IMDB. *http://www.imdb.com/interfaces.*

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[5] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *SPAA*, 2003.

[6] W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in RDF. In *SDKB*, pages 23–39, 2010.

[7] I. B. Arpinar, K. Giriloganathan, and B. Aleman-Meza. Ontology quality by detection of conflicts in metadata. In *EON*, 2006.

[8] D. Calvanese, W. Fischl, R. Pichler, E. Sallinger, and M. Simkus. Capturing relational schemas and functional dependencies in RDFS. In *AAAI*, 2014.

[9] F. Chiang and R. Miller. Discovering data quality rules. In *VLDB*, 2008.

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

[11] A. Cortés-Calabuig and J. Paredaens. Semantics of constraints in RDFS. In *AMW*, pages 75–90, 2012.

[12] 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.

[13] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity.* Springer, 2013.

[14] 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.

[15] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, volume 95, pages 632–640, 1995.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[17] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(1), 2008.

[18] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.

[19] W. Fan and P. Lu. Dependencies for graphs. In *PODS*, 2017.

[20] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *PVLDB*, 8(12):1502–1513, 2015.

[21] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, 2016.

[22] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.

[23] L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[24] 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.

[25] B. He, L. Zou, and D. Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In *SWIM*, pages 1–7, 2014.

[26] 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.

[27] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *SIGKDD*, 2004.

[28] 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.

[29] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, 2000.

[30] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(01):75–105, 2013.

[31] Y. Ke, J. Cheng, and J. X. Yu. Efficient discovery of frequent correlated subgraph pairs. In *ICDM*, 2009.

[32] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.

[33] 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.

[34] R. E. Korf. Multi-way number partitioning. In *IJCAI*, 2009.

[35] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *TCS*, 71(1):95–132, 1990.

[36] G. Lausen, M. Meier, and M. Schmidt. SPARQLing constraints for RDF. In *EDBT*, pages 499–509. ACM, 2008.

[37] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in MapReduce. In *ICDE*, 2014.

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

[39] P. Shelokar, A. Quirin, and Ó. Cordón. Three-objective subgraph mining using multiobjective evolutionary programming. *JCSS*, 80(1):16–26, 2014.

[40] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[41] 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.

[42] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. In *DISC*, 2013.

[43] 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.

[44] Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *ISWC*, pages 794–809. 2011.

[45] A. Zaveri, D. Kontokostas, M. A. Sherif, L. Bühmann, M. Morsey, S. Auer, and J. Lehmann. User-driven quality evaluation of DBpedia. In *ISEM*, pages 97–104, 2013.

A. Appendix: Proofs

A.1 Proof of Theorem 1

We show that 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 maximum degree d.

(1) Satisfiability and implication. We will give an algorithm for the satisfiability analysis of GFDs in the proof of Proposition 2, which takes $O(|\Sigma| \times k^k)$ time. Similarly, the implication analysis is in $O((|\Sigma| + |\varphi|) \times k^k)$ time. Hence by the definition of fixed-parameter tractability, these problems are fixed-parameter tractable with parameter k.

(2) Validation. We prove this by reduction from the complement of k-clique problem [13]. 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 $G = (V, E)$ and k, we construct numbers k' and d, a graph G_1, and a set Σ of GFDs such that Σ is k'-bounded, the maximum degree of the nodes in G_1 is d, and G has a clique of size k iff $G_1 \not\models \Sigma$. More specifically, we define $k' = k \times k + 1$ and $d = 2$. We construct G_1 and Σ as follows.

(a) Graph G_1 consists of the following connected components. (i) It contains a single node x labeled with a distinct label a_0 that does not appear in V. We set $x.A = 0$. (ii) For each edge (u, v) in G, G_1 contains two nodes $x^u_{(u,v)}$ and $x^v_{(u,v)}$ labeled with u and v, respectively. 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. We set $x^u_{(u,v)}.A = u$ and $x^v_{(u,v)}.A = v$. (iii) There are two edges $(x^u_{(u,v)}, x^v_{(u,v)})$ and $(x^v_{(u,v)}, x^u_{(u,v)})$ labeled 0. Intuitively, these encode the structure of graph G. 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.

(b) The set Σ has 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 x_0, x^1_2, …, x^1_k, x^2_1, x^2_3, …, x^k_{k-1}; intuitively, x_0 denotes the node in G where the conflict will occur. We use a group $x^i_1, \ldots, x^i_k$ of nodes (excluding x^i_i) to denote a node v_i in G, for $i \in [1, k]$; and
- ◦ the literals in X are $(X_1 \wedge \ldots \wedge X_k)$, where $X_1 = (x^1_2.A = x^1_3.A) \wedge \ldots \wedge (x^1_k.A = x^1_2.A)$, …, and $X_k = (x^k_1.A = x^k_2.A) \wedge \ldots \wedge (x^k_{k-1}.A = x^k_1.A)$. 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 $Q[\bar{x}]$ is (V, E, L, μ), where

- $V = \{v_0, v^1_2, \ldots, v^k_{k-1}\}$;
- $E = \{(v^{i_1}_{i_2}, v^{i_2}_{i_1}), (v^{i_2}_{i_1}, v^{i_1}_{i_2}) \mid (i_1, i_2 \in [1, k]) \wedge (i_1 \neq i_2)\}$;
- 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 $\mu(x^i_j) = v^i_j$.

Intuitively, 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^1_k, x^k_1) and (x^k_1, x^1_k). 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.

By the definition of Σ, we can see that Σ is k'-bounded, and the maximum degree of the nodes in G_1 is d. . 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. But 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, …, v_k. Then we can define the following match h of Q in G_1: $h(x^i_j) = x^{v_i}_{(v_i,v_j)}$ for $i \in [1, i]$, and $h(x_0) = x$. To simplify the presentation, here we assume that $x^{v_i}_{(v_i,v_j)}$ and $x^{v_i}_{(v_j,v_i)}$ are the same variable. 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 $\varphi = Q[\bar{x}](X \to x_0.A = 1)$. 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, it is possible only when the nodes labeled 'a_0' have two distinct values for attribute A. Hence there exists a match h of $\varphi = Q[\bar{x}](X \to l)$ 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 $v_1 = h(v^1_2).A$, $v_2 = h(v^2_3).A$, …, $v_{k-1} = h(v^{k-1}_k).A$, and $v_k = h(v^k_1).A$. It suffices to show that (i) v_1, v_2, …, and v_k are nodes in G, and (ii) v_1, v_2, …, and v_k form a k-clique in G.

To see that v_1, v_2, …and v_k are nodes in G, we show that $h(v^1_2)$, $h(v^2_3)$, …, $h(v^{k-1}_k)$ and $h(v^k_1)$ are not the node labeled 'a_0'. Indeed, the node labeled 'a_0' is an isolated node in G_1, while $h(v^1_2)$, $h(v^2_3)$, …, $h(v^{k-1}_k)$ and $h(v^k_1)$ are nodes having an adjacent edge. Hence v_1, v_2, …, and v_k in G.

Next, we show that there is an edge between any two of these nodes, That is, for any $i, j \in [1, k]$ such that $i \neq j$, there exists edge (x_i, x_j) in G_1, where $x_i.A$ (resp. $x_j.A$) is v_i (resp. v_j). 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 φ, there exists an edge (v^i_j, v^j_i) in Q. Define $x_i = h(v^i_j)$ and $x_j = h(v^j_i)$. Since $h(\bar{x}) \models X$, $x_i.A = h(v^i_j).A = h(v^i_{n_i}).A = v_i$ and $x_j.A = h(v^j_i).A = h(v^j_{n_j}).A = v_j$, where $n_i = ((i + 1) \mod k)$ and $n_j = ((j + 1) \mod k)$. Hence there is an edge (x_i, x_j) in G_1, and thus there exists an edge (v_i, v_j) in G. □

A.2 Proof of Proposition 2

We show that when k is a predefined constant, the satisfiability, implication and validation problems are in PTIME for k-bounded GFDs. To this end, we give PTIME algorithms for the three problems when k is a constant.

(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 enforced(Σ_Q) (see [21] for the definition of 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 GFD satisfiability given in [21]. For the complexity, observe the following. (a) 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. Since Q' is k-bounded, there are at most k^k isomorphic mappings of Q' to subgraphs of Q. Hence this can be done in $O(|\Sigma| \times k^k)$ time.

In the same process $\mathsf{enforced}(\Sigma_Q)$ can be deduced. (b) One can verify that $|\mathsf{enforced}(\Sigma_Q)| \leq |\Sigma| \times k$ when $\mathsf{enforced}(\Sigma_Q)$ is represented as an equivalence relation on attributes, and checking whether $\mathsf{enforced}(\Sigma_Q)$ is conflicting can be done in linear time; so step (2) takes $O(|\Sigma| \times k)$ time. Hence the algorithm is in $O(|\Sigma| \times k^k)$ time, in PTIME for constant k.

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

(1) compute the set Σ_Q of GFDs embedded in Q, and $\mathsf{closure}(\Sigma_Q, X)$ accordingly;

(2) if $\mathsf{closure}(\Sigma_Q, X)$ is conflicting, or if l can be deduced from $\mathsf{closure}(\Sigma_Q, X)$ via equality transitivity, then return true; otherwise, return false.

The correctness follows from the characterization of GFD implication of [21]. For the complexity, (a) step (1) computes Σ_Q in the same way as step (1) in the satisfiability checking above, except that we handle Q instead of patterns in Σ. It takes $O((|\varphi| + |\Sigma|) \times k^k)$ time. (b) Step (2) takes $O((|\varphi| + |\Sigma|) \times k)$ time, similar to the satisfiability analysis given above. Hence the algorithm is in $O((|\varphi| + |\Sigma|) \times k^k)$ time, which is in PTIME when k is a constant.

(c) Validation. Given a set Σ of GFDs and a graph G, we develop the following PTIME algorithm to check whether $G \models \Sigma$: for all GFDs $\varphi = Q[\bar{x}](X \to l)$ in Σ and all matches h of Q in G, check whether $h(\bar{x}) \models (X \to l)$; if so, return true; otherwise, return false. The correctness follows from the definition of satisfaction of GFDs. For the complexity, there are at most $|G|^k$ matches of Q in G for each GFD $\varphi = Q[\bar{x}](X \to l)$ in Σ. Hence the algorithm is in $O(|\Sigma| \cdot |G|^k)$ time, which is in PTIME when k is a constant. □

A.3 Proof of Theorem 3

We show that for any graph G and any nontrivial GFDs φ_1 and φ_2, if $\varphi_1 \ll \varphi_2$ then $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$.

Consider two nontrivial GFDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \to l_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to l_2)$ with $\varphi_1 \ll \varphi_2$. Then 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) $h(\bar{z}_1)=\bar{z}_2$, where $\bar{z}_i$ denotes the pivots 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. By the definition of support of positive GFDs, it suffices to show $Q_2(G, X_2 l_2, \bar{z}_2) \subseteq Q_1(G, X_1 l_1, \bar{z}_1)$. That is, for any set $\bar{v}$ of nodes in G, if there exists a match h_2 of Q_2 in G such that $h_2(\bar{z}_2) = \bar{v}$, $h_2(\bar{x}_2) \models X_2$ and $h_2(\bar{x}_2) \models l_2$, then there must exist a match h_1 of Q_1 in G such that $h_1(\bar{x}_1) = \bar{v}$, $h_1(\bar{x}_1) \models X_1$, and $h_1(\bar{x}_1) \models l_1$. Given h_2, we define $h_1 = h_2 \circ h$, the composition of h_2 and h. Then $\bar{v} = h_1(\bar{z}_1)$. 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 $\bar{v} \in Q_1(G, X_1 l_1, \bar{z}_1)$, and thus we can conclude that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$.

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

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

(a) When $\varphi_1 = Q_1(\emptyset \to \mathsf{false})$ and $\varphi_2 = Q_2(\emptyset \to \mathsf{false})$. Since $\varphi_1 \ll \varphi_2$, assume *w.l.o.g.* that Q'_i with pivots $\bar{z}_i$ is the base of φ_i for $i \in [1, 2]$ such that $Q'_1 \ll Q'_2$. Suppose that $\mathsf{supp}(\varphi_2, G) > 0$. To show that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$, we prove that $\mathsf{supp}(Q'_1, G) \geq \mathsf{supp}(Q'_2, G)$. Then by the definitions of $\mathsf{supp}(\varphi_1, G)$ and $\mathsf{supp}(\varphi_2, G)$, we have that $\mathsf{supp}(\varphi_1, G) = \mathsf{supp}(Q'_1, G) \geq \mathsf{supp}(Q'_2, G) = \mathsf{supp}(\varphi_2, G)$.

By the definition of support for patterns, it suffices to show $Q'_2(G, \bar{z}_2) \subseteq Q'_1(G, \bar{z}_1)$. That is, for any set $\bar{v}$ of nodes in G, if there is a match h_2 of Q'_2 in G such that $h_2(\bar{z}_2) = \bar{v}$, then there must be a match h_1 of Q'_1 in G such that $h_1(\bar{z}_1) = \bar{v}$. 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 $h(\bar{z}_1) = \bar{z}_2$. Define $h_1 = h_2 \circ h$. Then $\bar{v} = h_1(\bar{z}_1)$. Hence $\bar{v} \in Q'_1(G, \bar{z}_1)$, and thus $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$.

(b) When one of φ_1 and φ_2 has the form $Q[\bar{x}](\emptyset \to \mathsf{false})$ while the other is $Q'[\bar{x}'](X \to \mathsf{false})$ with $X \neq \emptyset$, the two are incomparable (Section 4.2). 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, $\mathsf{supp}(Q, G) = 0$ and $\mathsf{supp}(Q', G) \neq 0$.

(c) When $\varphi_1 = Q_1(X_1 \to \mathsf{false})$, and $\varphi_2 = Q_2(X_2 \to \mathsf{false})$, where $X_1 \neq \emptyset$ and $X_2 \neq \emptyset$. We show that $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$. 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 $\mathsf{supp}(\varphi'_1, G) \geq \mathsf{supp}(\varphi'_2, G)$. By the definitions of $\mathsf{supp}(\varphi_1, G)$ and $\mathsf{supp}(\varphi_2, G)$, we have that $\mathsf{supp}(\varphi_1, G) = \mathsf{supp}(\varphi'_1, G)$ and $\mathsf{supp}(\varphi_2, G) = \mathsf{supp}(\varphi'_2, G)$. Hence $\mathsf{supp}(\varphi_1, G) \geq \mathsf{supp}(\varphi_2, G)$. □

A.4 Load Balancing and Upgrading (Section 6.2)

Load balancing. A work unit $Q(F_s) \bowtie e(F_t)(t \in [1, n])$ is *skewed* if it takes more cost than the others. Due to evenly partitioned edges, $|e(F_t)|$ is bounded by $O(\frac{|E|}{n})$. ParDis quantifies the "skewness" of $Q(F_s)$ as $\frac{|Q(F_s)|}{\mathsf{Avg}_{t \in [1,n]} |Q(F_t)|}$. If the skewness of $Q(F_s)$ exceeds a threshold ϵ, the work unit is skewed. As Q is already verified in superstep $i - 1$, the checking is performed at S_c, and 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 10: Recall pattern Q'_1 and its corresponding work unit (Q_1, e) from Example 7. Consider three workers P_1, P_2 and P_3, which store the following for (Q_1, e).

	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 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 a 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 compute their local join in parallel. The balanced loads are 101, 104, and 104 for P_1, P_2 and P_3, respectively. □

Upgrading. We next illustrate the upgrading process.

Example 11: Consider GFD $\varphi_4 = Q'_1[x, y](X'' \to l)$ (Fig. 2). Suppose that the active domain Θ_x of $L_Q(x)$ in G is {"staff", "crew", "person"}. After parallel validation, ParDis finds two GFDs $\varphi_7 = Q_7[x, y](X'' \to l)$ and $\varphi_8 = Q_8[x, y](X'' \to l)$, where $Q_7[x, y]$ (resp. $Q_8[x, y]$) is a variant of Q'_1 by changing label $L_Q(x)$ from "person" to "staff" (resp. "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 $\varphi_4^- = Q_4[x, y](X'' \to l)$, 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 $P(Q'_1)$ in the GFD tree T as $P(Q'_1) \cup P(Q_7) \cup P(Q_8)$. □

A.5 Positive GFDs with Disconnected Patterns

We next extend ParDis to discover GFDs with disconnected patterns. The idea is to generate disconnected patterns following a 'TA-style" aggregation [16], by combining "top-ranked" connected patterns with aggregated support above threshold σ. It supports early termination without verifying all GFD candidates by its spawning strategy.

Algorithm. Besides the tree T, ParDis maintains k sorted lists $\mathcal{L} = (L_1, \ldots, L_k)$, denoting (at most) k components in k-bounded GFDs. Each list L_r stores the r-th connected components Q_r in a disconnected pattern $Q'{=}(Q_1, \ldots, Q_k)$, and is sorted by non-increasing order of pattern support.

List $\mathcal{L}$ is expanded by interleaving two levelwise spawning: DVSpawn(i), a variant of VSpawn to generate disconnected patterns; and DHSpawn(i, j) that revises HSpawn to generate GFDs. At superstep i, ParDis performs two steps below.

Disconnected pattern generation. Algorithm ParDis first executes VSpawn(i) to generate all connected patterns $\mathcal{Q}_i$ at level i of T, and performs parallel pattern matching, as in Section 6.2. It then executes DVSpawn(i) as follows.

(1) It first sorts connected patterns in $\mathcal{Q}_i$ following a descending order of its verified support supp(Q_r, G). Each list L_r has a cursor c_r that initially points at its top element.

(2) DVSpawn(i) then generates disconnected patterns $\mathcal{Q}'_i$ at level i of T. It generates a *new* disconnected pattern $Q = <Q_1, \ldots, Q_k>$, by fetching a pattern Q_r (Q_r can be $\emptyset$) from each list L_r that is pointed by cursor c_r, and moves down the cursor for the next pattern. It ensures that the total number of edges in Q does not exceed i, and computes parent set $P(Q)$ as the union of $P(Q_r)$ for $r \in [1, k]$.

It follows sorted access to $\mathcal{L}$, and computes supp(Q, G) by combining supp(Q_r, G) (see Section 4 for supp(Q, G)). As soon as $\theta < \sigma$, it stops spawning patterns and *terminates early* [16]. The correctness is ensured by Lemma 4(c).

Example 12: Consider GFD $\varphi = Q[\bar{x}](X' \to l)$ as shown in Fig. 8, where (a) Q includes disconnected $Q_1'^1$ and $Q_1'^2$, which are isomorphic to Q'_1 in Example 6, and (b) $X'{=}$ {x_1.name = x_2.name}, and l is y_1.name = y_2.name.

Assume $k = 6$. ParDis maintains 6-sorted list $\mathcal{L}$. Once

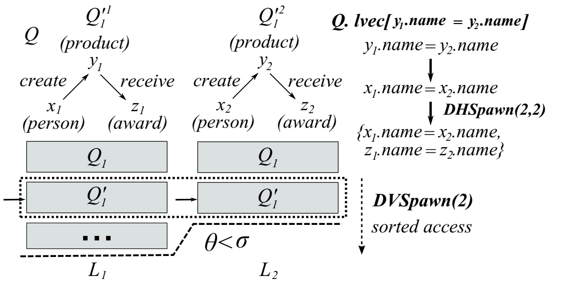


Figure 8: ParDis**: TA-style** GFD **discovery**

pattern Q'_1 in Fig. 2 is generated and verified by VSpawn(2), DVSpawn(2) inserts Q'_1 into all the 6 lists. It then follows a TA-style expansion to generate 6-bounded patterns, as shown in Fig. 8 (only two lists are shown). When the cursors move down to Q'_1 on both lists L_1 and L_2, DVSpawn(2) generates disconnected pattern Q, if supp(Q, G) $\geq \sigma$. □

Disconnected horizontal spawning. Algorithm ParDis next executes DHSpawn(i, j), which generates candidates $\varphi' = Q'(\bar{x})(X \to l)$ just like HSpawn(i, j), except that X and l include literals from different components of Q'. Indeed, if X and l consist of only literals from a single component, φ' is not reduced and is pruned (Lemma 4). After the candidates are generated, ParDis performs parallel validation as before. Upgrading GFDs with disconnected patterns is conducted in the same way as in HSpawn (Section 6.2).

Example 13: For Q and l of Example 12, DHSpawn(2, 2) expands $X'{=}$ {x_1.name = x_2.name} to X by adding z_1.name = z_2.name, if $G \not\models Q[\bar{x}](X' \to l)$ (by Lemma 4). Note that X' and l contain literals from both $Q_1'^1$ and $Q_1'^2$. □

One can verify that algorithm ParDis extended to GFDs with disconnected remains parallel scalable, along the same lines as the argument for negative GFDs (Section 6.2).

A.6 Parallel Scalability of ParDis (Section 6.2)

We 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 (communication cost), in time $O(\frac{|G|^{|\bar{x}|}}{n})$ (step (c) of the analysis of ParDis, Section 6.2).

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 $n \ll |G|$, for the first step of ParDis. 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), Section 6.2).

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, y are variables in $Q[\bar{x}]$. There are two cases. (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, 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 $|E_Q| = w + 1$.

A.7 Correctness of ParCover (Section 6.3)

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) $\Sigma_c \equiv \Sigma$.

We first show that Σ_c is minimal. Suppose by contradiction that there exists $\varphi \in \Sigma_c$ such that $\Sigma_c \setminus \{\varphi\} \models \varphi$. Let $\varphi = Q[\bar{x}](X \to Y)$. Then by the characterization of GFD implication [21], there exists a group Σ' of GFDs in Σ_c 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. The correctness is warranted by Lemma 6. Hence Σ_c includes only non-redundant GFDs of Σ.

We next show that $\Sigma_c \equiv \Sigma$. Since $\Sigma_c \subseteq \Sigma$, obviously $\Sigma \models \Sigma_c$. For the other direction, suppose that $\Sigma_c \not\models \Sigma$ by contradiction. Then we show that there must exist a GFD $\varphi = Q[\bar{x}](X \to Y)$ 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 now show that if $\Sigma_c \not\models \Sigma$, then there exists $\varphi = Q[\bar{x}](X \to Y)$ 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 φ_i. Then, no $\psi \in \Sigma_r$ is in $\Sigma_Q \setminus \Sigma^Q$. Hence, $\Sigma_c \models \Sigma_Q \setminus \Sigma^Q$, since by the definition of Σ_r, all GFDs in $\Sigma_Q \setminus \Sigma^Q$ are entailed by Σ_c.

A.8 Load Balancing in ParCover (Section 6.3)

By the characterization of [21], 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, it is to compute an assignment that sends each $\Sigma_{Q_j} \in W$ to a worker P_j, such that $\mathsf{load}(P_i) = \mathsf{load}(P_j)$ for $i, j \in [1, n]$. Here $\mathsf{load}(P_i) = \sum c(\Sigma_{Q_j})$ is the total load of P_j, for all work units Σ_{Q_j} sent to P_j. This problem is nontrivial. Nonetheless, there exists efficient approximation.

Proposition 7: *The load balancing problem for* GFD *implication is (1)* NP*-complete, and (2)* 2*-approximable.* □

Proof sketch: We first show Proposition 7(1). Given workload W, a set $\mathcal{P}$ of n workers, and a constant ϵ, the decision problem of load balancing is to decide whether there exists an assignment that balances the load of each worker, bounded by $\epsilon\frac{\sum_{w \in W} c(w)}{n}$. (1) 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 polynomial time. (2) To show the problem is NP-hard, we construct a reduction from the *number partition problem*, which is NP-complete [34]. The latter is to decide, given a set $S = \{n_1, \ldots, n_m\}$, whether there exists an n-partition of S such that the sum of the numbers in each set is equal.

Given S and n, we define 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 $\varphi_i = Q_i[x, y](x.A = 1 \to y.B = 1)$, where Q_i is a single-edge (x, y) labeled with i, and both x and y are labeled with i; and (b) Σ_{Q_i} also contains a set of GFDs $\varphi_i^j = Q_i'[x](x.B = j \to x.C = j)$ ($j \in [1, n_i - 1]$), where Q_i' consists of only one node labeled with i. We set (a) $c(\Sigma_{Q_i})$ as $|\Sigma_{Q_j}| * |Q_j|^{|Q_j|} = 4 \cdot n_i$, since the number of nodes in Q_j 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. Hence, the problem is NP-hard.

Approximation. Algorithm ParCover adopts a greedy strategy for a general load balancing problem [5] to iteratively assigns work units with the smallest cost to the workers with the (dynamically updated) least load. By developing an approximation-factor preserving reduction, one can verify that this algorithm is a 2-approximation [5], and is in $O(|W|n \log n)$ time, where $|W| \leq |\Sigma|$ as there are at most $|\Sigma|$ patterns, and each pattern yields one work unit. □

A.9 Expressing Axioms for Knowledge Enrichment

As an application, 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.*

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 [33, 45]. It can be enforced by GFDs of the form φ_2 given in Example 1.

A dual axiom states that if p is inverseFunctional, then for any object y, there exists at most one subject x such that (x, p, y) is a RDF triple [33]. For example, hasISBN is inverseFunctional. It can be expressed as GFDs similar to φ_2. It can catch errors such as (A_Thief_of_Time, hasISBN, 978-0-06-100004-1), (Listening_Woman, hasISBN, 978-0-06-100004-1) in YAGO3. This can also be expressed as GFDs.

Disjoint. The axiom states that two disjoint classes cannot have a common instance [7, 33, 45]. 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

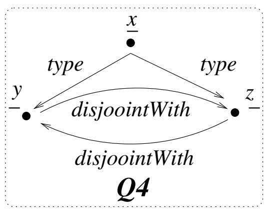
$$\varphi_4 = Q_4[x, y, z](\emptyset \to \mathsf{false})$$

where Q_4 is given in Fig. 9. 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 [33]. It can also be enforced by a GFD similar to φ_4 above.

Asymmetric. Property p is *asymmetric* if (x, p, y) cannot coexist with (y, p, x) [33]. This can be expressed as GFDs similar to φ_2, to catch *e.g.,* (Walter_Maule, child, Henry_Maule), (Henry_Maule, child, Walter_Maule) in DBpedia.

A related axiom is for irreflexive properties [33, 45]: 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



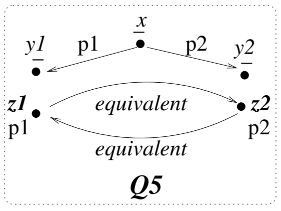


Figure 9: GFDs for knowledge enrichment axioms

(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 GFD

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

for Q_5 shown in Fig. 9. Similarly, synonym can be specified.

Inheritance. As shown in [21], general properties of subclass and is_a can be readily enforced by GFDs with wildcard '_'.