# Dependencies for Graphs: Challenges and Opportunities

WENFEI FAN, University of Edinburgh, Beihang University, and Shenzhen Institute of Computing Sciences

What are graph dependencies? What do we need them for? What new challenges do they introduce? This paper tackles these questions. It aims to incite curiosity and interest in this emerging area of research.

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

Additional Key Words and Phrases: Dependencies, graphs, satisfiability, implication, validation, dependency discovery, error detection, certain fixes

**ACM Reference Format:**
Wenfei Fan. 2019. Dependencies for Graphs: Challenges and Opportunities. *ACM J. Data Inform. Quality* 1, 1 (April 2019), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Relational dependencies specify a fundamental part of the semantics of relations, and are found in almost every database textbook. For example, among our familiar dependencies are functional dependencies (FDs), which help us design database schema, optimize queries, and clean relational data.

Do we need dependencies for graphs, *e.g.,* social networks, knowledge bases and finance transactions? Yes, the need for dependencies is even more evident for graphs. Unlike relational data, real-life graphs often do not come with a schema, and dependencies provide one of few means for us to specify the integrity and semantics of the data. They are useful in consistency checking, entity resolution, knowledge base expansion, spam and fraud detection, among other things.

*Example 1.1.* Consider knowledge bases and finance transactions, which are modeled as graphs.
*(1) Consistency checking*. It is common to find inconsistencies in real-life knowledge bases, *e.g.,*

- DBPedia: all birds can fly, and moa are birds, although moa are “flightless”;
- DBPedia: Philip Sclater is marked as both a child and a parent of William Lutley Sclater;
- Yago: psychologist Tony Gibson is credited for creating Ghetto Blaster, while the video game was actually created by programmer Tony ‘Gibbo’ Gibson;
- Yago: an institute BBC Trust was created in 2007 but destroyed in 1946; and
- Yago: a village Bhonpur in India has 600 females and 722 males; its total population is 1572.

As shown in [11, 12], such errors can be easily caught by functional dependencies (FDs) on graphs.
*(2) Knowledge base expansion.* When adding a newly extracted album to a knowledge base $G$, to avoid duplicates, we need rules to uniquely identify an album entity in $G$, specified in terms of

$\psi_1$: its title and the id of its primary artist, or
$\psi_2$: its title and the year of initial release.

Author’s address: Wenfei Fan, University of Edinburgh , Beihang University , Shenzhen Institute of Computing Sciences, wenfei@inf.ed.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1936-1955/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

As shown in [6], these rules can be expressed as keys on graphs. Note that the title of an album and the name of its artist cannot uniquely identify an album. For instance, an American band and a British band are both called "Bleach", and both bands had an album "Bleach".

To cope with $\psi_1$, we also need a key to identify artists:

$\psi_3$: the name of the artist, and the id of an album recorded by the artist.

As opposed to our familiar keys for relations, these keys are "recursively defined" on graphs: to identify an album, we may need to identify its primary artist, and vice versa.

*(3) Fraud detection.* Fraud causes U.S. banks to lose tens of billions of dollars every year, and 10%-20% of bad debt at leading US and European banks is actually fraud (cf. [23]). Fraud rings often operate as follows [23]: (a) a group of people organize a fraud ring; the ring shares a subset of legitimate contact information, *e.g.,* phone numbers and addresses, combining them to create a number of synthetic identities; (b) ring members open accounts using these "fake" identities; (c) the accounts are used normally, with regular purchases and timely payments; (d) banks increase the revolving credit lines over time, due to the observed responsible credit behavior; and (e) one day the ring "busts out", coordinating their activity, maxing out all of their credit lines, and disappearing.

For instance, 3 people each sharing 2 valid identifiers (phone numbers and addresses) lead to 9 interconnected synthetic identities. A ring of $n$ people sharing $m$ elements of data (such as name, date of birth, phone number, address, SSN, etc.) can create up to $n^m$ synthetic identities.

As will be seen shortly, we can detect such fraud by using graph dependencies that capture associations among accounts (synthetic identities), phone numbers and addresses. □

**Challenges**. No matter how useful, the study of graph dependencies is challenging. Graph dependencies are a departure from their relational counterparts. As an example, consider a traditional FD $R(X \rightarrow Y)$. It is defined on a relation schema $R$ with attributes $X$ and $Y$, where $R$ specifies the "scope" of the FD, *i.e.,* $X \rightarrow Y$ is to be applied to an instance $D$ of $R$ such that for any tuples $t_1$ and $t_2$ in $D$, if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$ [1]. In contrast, graphs are semistructured and are often schemaless. To define an FD on graphs $G$, we need a combination of (a) a topological constraint $Q$ to identify associated entities in $G$, specifying its "scope", *i.e.,* it identifies entities to which the FD is applied; and (b) a dependency $X \Rightarrow Y$ on the attributes of the entities identified.

As another example, relational keys are a simple special case of FDs and are based on value equality. In contrast, keys for graphs are often necessarily "id-based" and recursively defined, as shown by $\psi_1$–$\psi_3$ of Example 1.1. That is, they are based on node identity. Moreover, if two vertices are identified as the same entity, then they must have the same set of attributes and edges.

To make practical use of graph dependencies, several questions have to be answered. How should we define dependencies on schemaless graphs, to associate entities and specify their regularity? What is the complexity of reasoning about graph dependencies? What new challenges are introduced by such dependencies? How can we make practical use of the dependencies?

**A brief history**. In light of the practical need, graph dependencies are being investigated by W3C [19] and the industry (*e.g.,* [22]). While relational dependency theory makes an important part of database theory [1], the study of graph dependencies is still in its infancy. Even a "standard" form of FDs is not yet in place. This line of work started from [21]. It defines simple keys, foreign keys and FDs by extending relational methods to RDF, and treating relation names as class types. Also for RDF, FDs are defined with path patterns [24] and tree patterns [4] of entities. Using RDF triple patterns with variables, keys [6], equality-generating dependencies (EGDs) [2] and tuple-generating dependencies (TGDs) [5] are formulated. Over generic graphs, FDs and their extensions are defined with (possibly cyclic) graph patterns with variables [11, 16] and node identities [12].

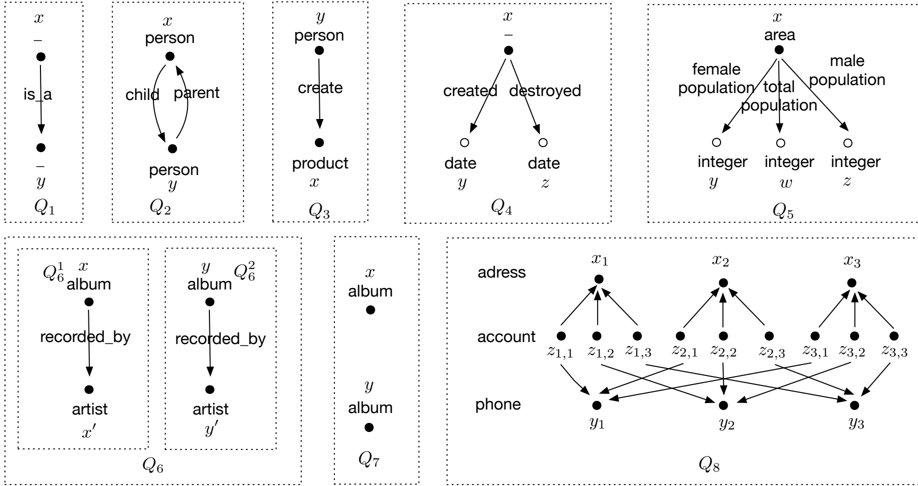


Fig. 1. Graph patterns

**Organization**. This paper aims to incite interest in the study of graph dependencies. As examples, we present graph dependencies introduced in [11, 12, 16] and their associated fundamental problems in Section 2, highlighting their differences from traditional relational dependencies. As a case study, we demonstrate how GEDs help us clean graph-structured data in Section 3, identifying technical problems underlying the application. We identify open problems in Section 4.

## 2 DEPENDENCIES FOR GRAPHS

As examples, we present graph functional dependencies of [16] and their extensions [11, 12].

### 2.1 Preliminaries

We first review some basic notations. We consider directed and (node and edge) labeled graphs.

Graphs. A *graph* $G$ is specified as $(V, E, L, F_A)$, where (a) $V$ is a finite set of nodes, in which each node $v$ has label $L(v)$; (b) $E$ is a finite set of edges, in which $(v, \iota, v')$ denotes an edge from node $v$ to $v'$ labeled with $\iota$; and (c) each node $v \in V$ carries a (finite) tuple $F_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$ of *attributes* for *e.g.*, properties, keywords and ratings, written as $v.A_i = a_i$, and $A_i \neq A_j$ if $i \neq j$. In particular, each node $v \in V$ carries a special attribute id denoting its *node identity*.

Unlike relational databases, there is typically no schema for graphs. Hence for an attribute $A$ and a node $v \in V$, $v.A$ may not exist, except that $v$ has a unique attribute $v.\mathrm{id}$ denoting its identity such that when two nodes have the same id, the two are the same node.

Graph patterns. A *graph pattern* is a directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q)$, where (1) $V_Q$ (resp. $E_Q$) is a finite set of pattern nodes (resp. edges); (2) $L_Q$ is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) to each node $u \in V_Q$ (resp. edge $e \in E_Q$); and (3) $\bar{x}$ is a list of distinct variables, each denoting a node in $V_Q$. We allow wildcard '_' as a special label for nodes or edges in $Q$ that matches any label.

For example, Figure 1 depicts a few graph patterns. Note that nodes in *e.g.*, $Q_1$, carry wildcard _.

Matches. A *match* of pattern $Q[\bar{x}]$ in graph $G$ is a homomorphism $h$ from $Q$ to $G$ such that (a) for each node $u \in V_Q$, $L_Q(u) \asymp L(h(u))$; and (b) for each edge $e = (u, \iota, u')$ in $Q$, there exists an edge $e' = (h(u), \iota', h(u'))$ in $G$ such that $\iota \asymp \iota'$. Here $\iota \asymp \iota'$ if either the two labels are identical or one of them is wildcard '_'. Distinct wildcards may match different label valuations. We denote the match as $h(\bar{x})$, where $h(\bar{x})$ consists of $h(x)$ for all variables $x \in \bar{x}$. Intuitively, $\bar{x}$ is a list of entities to be identified by pattern $Q$, and $h(\bar{x})$ is an instantiation of $\bar{x}$ in graph $G$, one node for each entity.

### 2.2 Graph Dependencies

A *graph functional dependency* (GFD) [16] $\varphi$ is defined as $Q[\bar{x}](X \Rightarrow Y)$, where $Q[\bar{x}]$ is a graph pattern, $X$ and $Y$ are conjunctions of literals of $\bar{x}$. A literal has the following form: for $x, y \in \bar{x}$,

(a) *constant literal* $x.A = c$, where $c$ is a constant, and $A$ is an attribute that is not id; or
(b) *variable literal* $x.A = y.B$, where $A$ and $B$ are attributes that are not id.

Intuitively, GFD $\varphi$ is a pair of (1) a *topological constraint* imposed by pattern $Q$, to identify entities in a graph, and (2) an attribute dependency $X \Rightarrow Y$ to be applied to the entities identified by $Q$.

*Example 2.1.* We can use GFDs as rules to detect some errors and fraud observed in Example 1.1.

(1) Consistency checking. The following GFDs are taken from [12], defined with patterns of Figure 1.

(a) $\varphi_1 = Q_1[x, y](x.A = x.A \Rightarrow y.A = x.A)$, where $A$ is an attribute of $x$. It says that if entity $y$ is_a $x$ and if $x$ has property $A$, then $y$ inherits $x.A$, *i.e.,* $y$ also has attribute $A$ and $y.A = x.A$. Note that $x$ and $y$ are labeled '_', representing generic entities regardless of their labels. When $A$ is can_fly, $\varphi_1$ catches the inconsistency between birds and moa in the classification of DBPedia.

(b) $\varphi_2 = Q_2[x, y](\text{true} \Rightarrow \text{false})$, where true and false are syntactic sugar for Boolean values. More specifically, true is empty $X$, *i.e.,* when $X$ imposes no pre-conditions, and false can be expressed as $y.A = c$ and $y.A = d$ for distinct constants $c$ and $d$. The GFD states that graph pattern $Q_2$ is "illegal", *i.e.,* no person can be both a child and a parent of another person.

(c) $\varphi_3 = Q_3[x, y](x.\text{type} = \text{"video game"} \Rightarrow y.\text{type} = \text{"programmer"})$. Here $Q_3[x, y]$ specifies a person $y$ and a product $x$, linked by a create edge; both entities carry attribute type. The GFD says that a video game is only created by programmers; it catches the error in Yago about Ghetto Blaster.

(2) Fraud detection. Recall behaviors of fraud rings observed in [23] and described in Example 1.1.

(d) $\phi = Q_8[\bar{x}, \bar{z}, \bar{y}](\text{true} \Rightarrow \text{false})$. Pattern $Q_8$ shows bank accounts opened with "fake" identities by a fraud ring of 3 people, where $\bar{x} = (x_1, x_2, x_3)$ of 3 legitimate addresses; $\bar{y} = (y_1, y_2, y_3)$, for 3 phone numbers; and $\bar{z} = (z_{1,1}, \ldots, z_{3,3})$; each $z_{i,j}$ denotes a bank account opened with a synthetic identity; it shares address with $z_{i,k}$ and phone number with $z_{l,j}$ for $i, k \in [1, 3]$. The GFD says that if a group of accounts match pattern $Q_8$, then these accounts need to be monitored and scrutinized. □

**Semantics**. We next interpret GFD $Q[\bar{x}](X \Rightarrow Y)$. Consider a match $h(\bar{x})$ of $Q$ in a graph $G$, and a literal $x.A = c$ of $\bar{x}$. We say that $h(\bar{x})$ *satisfies* the literal if attribute $A$ *exists* at node $v = h(x)$, and moreover, $v.A = c$; similarly for literals $x.A = y.B$. We denote by $h(\bar{x}) \models X$ if the match $h(\bar{x})$ satisfies *all* the literals in $X$. We write $h(\bar{x}) \models X \Rightarrow Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$.

A graph $G$ *satisfies* GFD $\varphi$, denoted by $G \models \varphi$, if *for all* matches $h(\bar{x})$ of $Q$ in $G$, $h(\bar{x}) \models X \Rightarrow Y$. We say that $G$ *satisfies* a set $\Sigma$ of GFDs if for all $\varphi \in \Sigma$, $G \models \varphi$, *i.e.,* $G$ satisfies each GFD in $\Sigma$.

Intuitively, if $h(\bar{x})$ *violates* $X \Rightarrow Y$, *i.e.,* $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$, then the subgraph induced by $h(\bar{x})$ is inconsistent. Hence we can detect errors using GFDs as demonstrated in Example 2.1.

Recall that for a relational FD $R(X \rightarrow Y)$, all attributes in $X \cup Y$ are assured to exist by schema $R$. In contrast, for a literal $x.A = c$ in a GFD $Q[\bar{x}](X \Rightarrow Y)$, node $h(x)$ does not necessarily have attribute $A$ in a schemaless graph. By the definition of satisfaction, (a) when $x.A = c$ is a literal in $X$, if $h(x)$ has *no* $A$-attribute, then $h(\bar{x})$ trivially satisfies $X \Rightarrow Y$. (b) In contrast, if $x.A = c$ is in $Y$, then for $h(\bar{x}) \models Y$, node $h(x)$ must have $A$-attribute; similarly for $x.A = y.B$. That is, graph dependencies have to cope with the semistructured nature of graphs, as opposed to traditional dependencies.

**Extensions**. GFDs extend conditional functional dependencies (CFDs) [8] to graphs, by supporting bindings of semantically related constants with constant literals. CFDs have proven effective in catching semantics inconsistencies in relational data [7]. However, there are still errors in real-life graphs that GFDs cannot capture. To cope with these, several extensions of GFDs have been studied.

*(1)* NGDs. Semantic inconsistencies often involve numeric values. To catch such errors, NGDs extend GFDs by supporting linear arithmetic expressions and built-in comparison predicates [11]. That is, for a graph pattern $Q[\bar{x}]$, we allow literals of $\bar{x}$ of the form $e_1 \otimes e_2$, where (a) $\otimes$ is one of $=, \neq, <, \leq, >$ and $\geq$, and (b) $e_1$ and $e_2$ are linear arithmetic expressions defined in terms of attributes $x.A$ and constants $c$ connected with $+, -, \times, \div$, with degree 1 by treating $x.A$ as variables.

*Example 2.2.* Continuing with Example 2.1, we use the NGDs below to catch inconsistencies with numeric values. These NGDs are defined with patterns in Figure 1 and are borrowed from [11].

(d) NGD $\varphi_4 = Q_4[x, y, z](\text{true} \Rightarrow z.\text{val} - y.\text{val} \geq c)$. From $Q_4$ of Figure 1, we can see that $x, y$ and $z$ denote an entity, the date when the entity was created and the date when it was destroyed, respectively; val is an attribute for the integer values of $y$ and $z$ in days; and $c$ is a constant. The NGD states that an entity cannot be destroyed within $c$ days of its creation.

(e) NGD $\varphi_5 = Q_5[w, x, y, z](\text{true} \Rightarrow y.\text{val} + z.\text{val} = w.\text{val})$. It says that in any area $x$, its total population $w.\text{val}$ should equal the sum of its female population $y.\text{val}$ and male population $z.\text{val}$. □

*(2)* GEDs. Relational keys are a special case of FDs. In contrast, GFDs cannot identify entities (nodes) in a graph. To uniformly express FDs and keys on graphs, GEDs extend GFDs by supporting id literals $x.\text{id} = y.\text{id}$, identifying entities $x$ and $y$ [12]. In particular, GEDs of the form $Q[\bar{z}](X \Rightarrow x_0.\text{id} = y_0.\text{id})$ are called *keys*, denoted as GKeys, where $x_0$ and $y_0$ are two designated nodes in $Q[\bar{x}]$.

*Example 2.3.* The GKeys below are taken from [12], defined with patterns in Figure 1.

For album: $\psi_1 = Q_6[x, x', y, y'](x.\text{title} = y.\text{title} \wedge x'.\text{id} = y'.\text{id} \Rightarrow x.\text{id} = y.\text{id})$,
$\psi_2 = Q_7[x, y](x.\text{title} = y.\text{title} \wedge x.\text{release} = y.\text{release} \Rightarrow x.\text{id} = y.\text{id})$.
For artist: $\psi_3 = Q_6[x, x', y, y'](x.\text{name} = y.\text{name} \wedge x.\text{id} = y.\text{id} \Rightarrow x'.\text{id} = y'.\text{id})$.

Note that pattern $Q_6[x, x', y, y']$ consists of a pattern $Q_6^1[x, x']$ and a "copy" $Q_6^2[y, y']$ of $Q_6^1$ by renaming variables $x \mapsto y$ and $x' \mapsto y'$. To identify albums $x$ and $y$, we check either their title attributes and the ids of their artists ($\psi_1$), or their title and release attributes ($\psi_2$). Similarly, to identify artists $x'$ and $y'$ as required by $\psi_1$, we check the ids of albums they recorded ($\psi_3$) in turn. □

*(3)* GED$^\vee$s. Real-life graphs are typically schemaless. Can we express a schema using graph dependencies, by enforcing a node to carry certain attributes of a particular type? As remarked earlier, we can enforce the existence of attribute $A$ at a node $x$ by using GFD $Q[\bar{x}](\text{true} \Rightarrow x.A = x.A)$. However, we cannot enforce attribute $x.A$ to have a finite domain, *e.g.,* Boolean.

To this end, GED$^\vee$s extend GEDs by adding limited disjunctions [12]. A GED$^\vee$ has the same syntactic form $\psi = Q[\bar{x}](X \Rightarrow Y)$ as GEDs, but $Y$ is interpreted as the disjunction of its literals. That is, for a match $h(\bar{x})$ of $Q$ in a graph $G$, $h(\bar{x}) \models Y$ if *there exists* a literal $l$ in $Y$ such that $h(\bar{x})$ satisfies $l$.

We can enforce each node of "type" $\tau$ to have an attribute with a finite domain, *e.g.,* Boolean:

$$Q_e[x](\text{true} \Rightarrow x.A = 0 \vee x.A = 1),$$

where $Q_e$ is a single node labeled $\tau$. Note that NGDs can also express the "domain constraint":

$\phi_1$: $Q_e[x](\text{true} \Rightarrow x.A = x.A)$, $\phi_2$: $Q_e[x](x.A \neq 0 \wedge x.A \neq 1 \Rightarrow \text{false})$.

That is, each $\tau$-node $x$ must have an $A$-attribute ($\phi_1$), and $x.A$ can only takes values 0 or 1 ($\phi_2$).

## 2.3 Classical Problems

What graph dependencies should we adopt in practice? A guideline is to strike a balance between the expressive power of the dependencies we need for our applications, and the complexity of reasoning about these dependencies. There are three classical problems for reasoning about graph dependencies, namely, the satisfiability, implication and validation problems, to be stated shortly. Table 1 shows the complexity bounds of these problems for the graph dependencies we have seen in Section 2.2, compared with their counterparts for relational FDs and CFDs.

| Dependencies | Satisfiability | Implication | Validation | Remark |
|---|---|---|---|---|
| GFDs [16] | coNP-complete | NP-complete | coNP-complete | CFDs on graphs: $Q[\bar{x}](X \Rightarrow Y)$ |
| GKeys [12] | coNP-complete | NP-complete | coNP-complete | $Q[\bar{x}](X \Rightarrow x.\text{id} = y.\text{id})$ |
| GEDs [12] | coNP-complete | NP-complete | coNP-complete | GFDs extended with id literals |
| NGDs [11] | $\Sigma_2^p$-complete | $\Pi_2^p$-complete | coNP-complete | (linear) arithmetic and comparison |
| GED$^\vee$s [12] | $\Sigma_2^p$-complete | $\Pi_2^p$-complete | coNP-complete | disjunctive $Y$ in $Q[\bar{x}](X \Rightarrow Y)$ |
| FDs (cf. [1]) | O(1) | linear-time | PTIME | $R(X \rightarrow Y)$ on relation $R$ |
| CFDs [8] | NP-complete | coNP-complete | PTIME | FDs extended with value patterns |

Table 1. Complexity of fundamental problems

**Satisfiability**. Consider a class $C$ of graph dependencies and a set $\Sigma$ of dependencies in $C$. A *model* of $\Sigma$ is a graph $G$ such that (a) $G \models \Sigma$, and (b) for each $Q[\bar{x}](X \Rightarrow Y)$ in $\Sigma$, $Q$ has a match in $G$.

Intuitively, if $\Sigma$ has a model, then each dependency in $\Sigma$ is sensible and the dependencies in $\Sigma$ do not conflict with each other. We say that $\Sigma$ is *satisfiable* if it has a model.

The *satisfiability problem for* $C$ is to decide whether a given set $\Sigma$ of $C$-dependencies is satisfiable.

For relational FDs, the satisfiability problem is trivial: for any set $\Sigma$ of FDs, there always exists a nonempty relation that satisfies $\Sigma$ (cf. [7]). In contrast, it is more intriguing for graph dependencies. As shown in Table 1, it is coNP-complete for GFDs, GKeys and GEDs, and is $\Sigma_2^p$-complete for NGDs and GED$^\vee$. The intractability is quite robust: it remains intractable even when $\Sigma$ consists of a fixed number of dependencies, and when each dependency in $\Sigma$ is defined with a tree pattern [12].

The satisfiability problem for relational CFDs is NP-complete [8], but it is in PTIME in the absence of attributes that have a finite domain, *e.g.,* Boolean [8]. GFDs cannot enforce an attribute to have a finite domain, and their satisfiability problem is intractable in the absence of finite-domain attributes. Hence its intractability is not inherited from CFDs, as indicated by the difference between coNP and NP (unless P = NP). While NGDs and GED$^\vee$ can specify finite-domain constraints, the satisfiability problem for NGDs and GED$^\vee$ is $\Sigma_2^p$-complete, as opposed to NP-complete for CFDs.

To see the need for striking a balance between the complexity and expressive power, the satisfiability problem for NGDs extended with non-linear arithmetic expressions is undecidable, even when no expressions have degree above 2 [11]. That is why NGDs support linear arithmetic expressions only.

**Implication**. A set $\Sigma$ of dependencies *implies* another dependency $\varphi$, denoted by $\Sigma \models \varphi$, if for all graphs $G$, if $G \models \Sigma$ then $G \models \varphi$. We consider finite implication, when graphs $G$ are finite.

The *implication problem* for a class $C$ of graph dependencies is to decide, given a finite set $\Sigma$ of dependencies in $C$ and another dependency $\varphi$ in $C$, whether $\Sigma \models \varphi$.

Intuitively, the implication analysis helps us optimize data quality rules, among other things.

As shown in Table 1, the implication problem is NP-complete for GFDs, GKeys and GEDs, and is $\Pi_2^p$-complete for NGDs and GED$^\vee$. The intractability remains intact when $\Sigma$ consists of a fixed number of dependencies, and when $\varphi$ and all dependencies in $\Sigma$ carry a tree pattern. In contrast, it is in linear time for relational FDs, and coNP-complete for CFDs with finite-domain attributes [8].

**Validation**. The *validation problem* for a class $C$ of graph dependencies is to decide, given a finite set $\Sigma$ of dependencies in $C$ and a graph $G$, whether $G \models \Sigma$.

The validation analysis is the basis of inconsistency, spam and fraud detection, to find violations of graph dependencies in a knowledge base, a social graph and finance transactions, respectively.

While the validation problem for relational FDs and CFDs is in PTIME, it is more intriguing for graph dependencies unless P = NP. As shown in Table 1, the validation problem is coNP-complete for GFDs, GKeys, GEDs, NGDs and GED$^\vee$, even when $\Sigma$ consists of a constant number of dependencies and when the dependencies in $\Sigma$ carry tree patterns only. The result is a bit surprising since it is in PTIME to decide, given graphs $Q$ and $G$, whether there exists a homomorphism from $Q$

to $G$ when $Q$ is a tree. These tell us that when graph pattern matching and attribute dependencies $X \Rightarrow Y$ (in graph dependencies) are put together, the analysis becomes harder [12].

**Finite axiomatization**. Recall Armstrong's axioms for relational FDs [3]: reflexivity, augmentation and transitivity. The axioms reveal insight of (finite) implication of FDs.

We naturally want a finite set $\mathcal{A}$ of inference rules to characterize graph dependencies, along the same lines as Armstrong's axioms. For a class $C$ of graph dependencies, we use $\Sigma \vdash_{\mathcal{A}} \varphi$ to denote that $\varphi$ is *provable from* $\Sigma$ *using* $\mathcal{A}$, where $\Sigma$ is a set of dependencies in $C$ and $\varphi$ is a dependency in $C$. That is, $\varphi$ can be deduced from $\Sigma$ by applying the rules in $\mathcal{A}$ (see [1] for the notion of proofs).

For a class $C$ of graph dependencies, we say that an inference system $\mathcal{A}$ is

- *sound* if $\Sigma \vdash_{\mathcal{A}} \varphi$ implies $\Sigma \models \varphi$, and
- *complete* if $\Sigma \models \varphi$ implies $\Sigma \vdash_{\mathcal{A}} \varphi$,

for all sets $\Sigma$ of dependencies in $C$ and all dependencies $\varphi$ in $C$. We say that $\mathcal{A}$ is

- *independent* if for any rule $\mathsf{r} \in \mathcal{A}$, there exist $\Sigma$ and $\varphi$ in $C$ such that $\Sigma \vdash_{\mathcal{A}} \varphi$ but $\Sigma \nvdash_{\mathcal{A} \setminus \mathsf{r}} \varphi$,

where $\mathcal{A} \setminus \mathsf{r}$ denotes $\mathcal{A}$ excluding $\mathsf{r}$, *i.e.,* removing any rule from $\mathcal{A}$ makes it no longer complete.

Despite the complications introduced by graph dependencies, there exists a set $\mathcal{A}$ of six inference rules that is sound, complete and independent for GEDs (and hence for GFDs and GKeys) [12]. That is, graph dependencies may retain the finite axiomatizability of relational FDs.

### 2.4 Graph Dependencies versus Relational Dependencies

We can see that graph dependencies depart from our familiar relational dependencies and introduce new challenges. We summarize their differences by considering the following dichotomies.

*(1) Definition.* Graph dependencies are defined on graphs that often do not come with a schema. To cope with schemaless graphs, a graph dependency is typically a combination of a topological constraint (graph pattern $Q$) and an attribute dependency $X \Rightarrow Y$. It is to be applied to matches of pattern $Q$ in a (possibly big) graph $G$, rather than to tuples. Moreover, $X$ and $Y$ may contain id literals $x.\mathrm{id} = y.\mathrm{id}$, such that when two entities (vertices) $x$ and $y$ are identified, they must carry the same set of attributes and adjacent edges. These are a departure from relational dependencies.

*(2) Expressive power.* To cope with the semistructured nature of schemaless graphs, graph dependencies such as GEDs are not a mere extension of equality-generating dependencies (EGDs); they can express limited tuple-generating dependencies (TGDs) and are able to "generate new attributes", not only by merging nodes with GKeys, but also by applying GFDs (see [12] for details).

*(3) Complexity.* As shown in Table 1, graph dependencies are harder to reason about than their relational counterparts (unless P = NP). This is not surprising: graph dependencies are interpreted in terms of graph homomorphism, which is already intractable (cf. [17]). Nonetheless, the complexity of the static analyses (satisfiability and implication) of, *e.g.,* GFDs and GEDs, is comparable to their counterparts for relational CFDs. That is, their analyses do not make our lives much harder.

*(4) Techniques.* We can make use of, *e.g.,* the data locality of graph homomorphism to check graph dependencies. More specifically, for any graph $G$ and any node $v$ in $G$, to decide whether a pattern $Q$ has a match in $G$ at node $v$, it suffices to inspect only those nodes of $G$ that are within $d_Q$ hops of $v$, where $d_Q$ is determined only by the size $|Q|$ of pattern $Q$. We do not have to scan the entire (possibly big) $G$. Such native graph techniques yield efficient implementation strategies for supporting graph dependencies, which are not offered by relational reasoning methods for EGDs and TGDs.

Better yet, graph patterns embedded in graph dependencies are able to explicitly characterize associations among entities. As demonstrated by Examples 2.1, 2.2 and 2.3, this property is useful in, *e.g.,* fraud detection, knowledge base expansion and social media marketing.

## 3 MAKE PRACTICAL USE OF GRAPH DEPENDENCIES

Graph dependencies find applications in knowledge acquisition, knowledge base enrichment, spam detection in social networks, and fraud detection in finance transactions, among other things. As an example, below we demonstrate how graph dependencies help us improve the quality of graph-structured data, highlighting technical challenges introduced by real-life graphs.

**Parallel scalability**. Real-life graphs easily have billions of nodes and trillions of edges, *e.g.,* the social graph at Facebook [18]. Add to the challenge that the analyses of graph dependencies are costly (Table 1). To support dependencies on real-life graphs, parallel processing is often a must. The assumption is that the more processors are used, the less time it takes to process dependencies. However, many parallel algorithms in the literature provide no such performance guarantee.

A criterion to characterize the effectiveness of parallel algorithms is the notion of parallel scalability proposed in [20], which has been widely used in practice. Consider a serial (single-machine) algorithm $\mathcal{M}$ for processing graph dependencies, with cost $t(|G|, |\Sigma|)$ measured in the sizes of graph $G$ and a set $\Sigma$ of graph dependencies. A parallel algorithm $\mathcal{M}_p$ for the same task is said to be *parallel scalable relative* to yardstick $\mathcal{M}$ if its parallel running time can be expressed as:

$$T(|G|, |\Sigma|, p) = O\big(\frac{t(|G|, |\Sigma|)}{p}\big),$$

where $p$ is the number of processors used. Intuitively, parallel scalability measures speedup over serial algorithms by parallelization. It is a relative measure *w.r.t.* a yardstick algorithm $\mathcal{M}$. A parallel scalable $\mathcal{M}_p$ "linearly" reduces the running time of $\mathcal{M}$ when $p$ increases. Hence a parallel scalable algorithm is able to scale with large graphs $G$ by adding processors as needed, to an extent.

Parallel scalability is within reach of graph dependency analyses, despite their intractability. Indeed, parallel scalable algorithms for GFD satisfiability and implication are developed in [10].

**Discovering graph dependencies**. No matter how useful, it is nontrivial to find interesting graph dependencies. This highlights the need to study discovery of graph dependencies.

- Input: A graph $G$, and a support threshold $\sigma > 0$.
- Output: A cover $\Sigma_c$ of the set $\Sigma$ of all graph dependencies $\varphi$ satisfied by $G$ with $\mathsf{supp}(\varphi, G) \geq \sigma$.

Here we measure the interestingness of $\varphi$ in terms of the support of $\varphi$ in $G$, denoted by $\mathsf{supp}(\varphi, G)$, indicating how often $\varphi$ can be applied and thus whether $\varphi$ captures regularity. The notion of support on graphs is rather different from its counterpart in data mining over itemsets (see [9] for details). We want to find *a cover* $\Sigma_c$ of $\Sigma$ such that (a) $G \models \Sigma_c$, *i.e.,* all dependencies in $\Sigma_c$ are sensible; (b) $\Sigma_c$ is "equivalent to" $\Sigma$, *i.e.,* for each $\varphi \in \Sigma$, $\Sigma_c \models \varphi$, and vice versa; (c) for each $\varphi = Q[\bar{x}](X \Rightarrow Y)$ in $\Sigma_c$, reducing its pattern $Q$ or dependency $X \Rightarrow Y$ makes it no longer satisfied by $G$; and (d) $\Sigma_c$ is minimal itself, *i.e.,* removing any dependency from $\Sigma_c$ makes it no longer a cover of $\Sigma$. That is, graph dependencies in $\Sigma_c$ are frequent, non-trivial, non-redundant and hence, interesting.

Parallel scalable algorithms have been developed for discovering interesting GFDs [9].

**Detecting errors**. After we discover a set $\Sigma_c$ of graph dependencies, we want to detect errors by using the dependencies of $\Sigma_c$ as data quality rules. Consider a graph dependency $\varphi = Q[\bar{x}](X \Rightarrow Y)$. A *violation* of $\varphi$ in graph $G$ is a match $h(\bar{x})$ of $Q$ in $G$ such that $h(\bar{x})$ violates $X \Rightarrow Y$, *i.e.,* there exist inconsistencies in the subgraph of $G$ induced from $h(\bar{x})$. The *error detection problem* is as follows.

- Input: A graph $G$, and a set $\Sigma_c$ of graph dependencies treated as data quality rules.
- Output: The set of all violations of the dependencies of $\Sigma_c$ in $G$, denoted by $\mathsf{Vio}(\Sigma_c, G)$.

We have seen how to detect errors using GFDs (Example 2.1), NGDs (Example 2.2) and GKeys (Example 2.3). However, detecting errors in graphs is nontrivial, since the validation problem we have seen in Section 2.3 is its decision problem and is known to be coNP-complete.

A related problem is *the incremental error detection problem*, stated as follows.

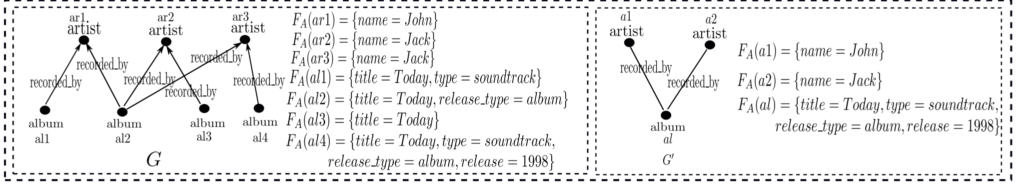


Fig. 2. Graph for repairing

- Input: A graph $G$, a set $\Sigma_c$ of graph dependencies, and updates $\Delta G$ to graph $G$.
- Output: The changes $\Delta\mathsf{Vio}(\Sigma_c, G, \Delta G)$ to $\mathsf{Vio}(\Sigma_c, G)$, where $\Delta\mathsf{Vio}(\Sigma_c, G, \Delta G)$ consists of new errors introduced by updates $\Delta G$ and errors removed from $\mathsf{Vio}(\Sigma_c, G)$ by $\Delta G$.

The need for incremental error detection is evident. As remarked earlier, real-life graphs $G$ are often big, and error detection is expensive (coNP-complete). Moreover, real-life graphs are constantly changed. It is often too costly to recompute $\mathsf{Vio}(\Sigma_c, G \oplus \Delta G)$ starting from scratch in response to frequent $\Delta G$. Hence we want to compute $\mathsf{Vio}(\Sigma_c, G)$ once, and then incrementally compute changes $\Delta\mathsf{Vio}$ in response to $\Delta G$. The rationale behind this is that in the real world, changes are typically small. Moreover, when $\Delta G$ is small, $\Delta\mathsf{Vio}$ is often small as well, and is much less costly to compute than $\mathsf{Vio}(\Sigma_c, G \oplus \Delta G)$ by making use of previous computation for $\mathsf{Vio}(\Sigma_c, G)$.

However useful, the incremental error detection problem is already coNP-complete for GFDs, even when both graph $G$ and updates $\Delta G$ have a constant size [11].

Not all is lost. Parallel scalable algorithms have been developed for error detection [16] and incremental error detection [11], when GFDs and NGDs are used as data quality rules.

**Correcting errors with certainty**. After we detect a set $\mathsf{Vio}(\Sigma_c, G)$ of inconsistencies, how do we fix the errors? A rule-based approach is to employ dependencies in $\Sigma_c$ as data quality rules.

*Example 3.1.* Consider graph $G$ depicted in Fig. 2, having three artists ($ar1$, $ar2$, and $ar3$) and four albums ($al1$, $al2$, $al3$, and $al4$), where artists $ar2$ and $ar3$ have the same name, and all albums have the same title. Consider $\Sigma_c$ including GKeys $\psi_1$–$\psi_3$ of Example 2.3, and a GFD $\varphi_4 = Q_7[x, y](x.\mathsf{title} = y.\mathsf{title} \wedge x.\mathsf{type} = y.\mathsf{type} \wedge x.\mathsf{release_type} = y.\mathsf{release_type} \Rightarrow x.\mathsf{release} = y.\mathsf{release})$, where $Q_7$ is shown in Figure 1. Observe that $\mathsf{Vio}(\Sigma, G)$ is nonempty, *e.g.,* $al1$, $al2$ and $al3$ violate GKey $\psi_1$.

We fix the errors using the dependencies in $\Sigma_c$ as follows: (1) merge $al1$, $al2$ and $al3$ using GKeys $\psi_1$; (2) add attribute $al3.\mathsf{release} = 1998$ to album $al3$ by applying GFD $\varphi_4$ to $al3$ and $al4$; (3) merge $al3$ and $al4$ with GKeys $\psi_2$; and (4) merge $ar2$ and $ar3$ using GKeys $\psi_3$. This yields a "repair" $G'$ of $G$ shown in Fig. 2. Note that the process interleaves object identification with GKeys (steps (1), (3) and (4)) and data repairing with GFDs (step (2)). Before step (1), node $al3$ has neither attribute type nor release_type, and GFD $\varphi_4$ cannot be applied to $al3$ and $al4$. After step (1), attributes type and release_type are added to $al3$, which are inherited from $al1$ and $al2$, respectively; this step "generates" new attributes of $al3$. Then we can apply $\varphi_4$, and further apply GKeys $\psi_3$ to merge $ar2$ and $ar3$. The process fixes all the violations of the dependencies in $\Sigma_c$, including the violation of $\psi_2$ by $al3$ and $al4$ that becomes obvious only after $\varphi_4$ is applied to $al3$ and $al4$. □

Formally, the graph cleaning process can be modeled as the chase on graphs [12], which has the Church-Rosser property: the chase converges at the same set of fixes regardless of the order of dependencies applied. Better yet, assume a block $\Gamma$ of ground truth, *i.e.,* attribute values and entity matches that are assured correct by, *e.g.,* domain experts or crowd-sourcing. Then if we apply a rule $Q[\bar{x}](X \Rightarrow Y)$ only when $X$ involves facts that are either in $\Gamma$ or deduced in the chase, the fixes generated are *certain*, *i.e.,* they are guaranteed correct as logical consequences of $\Sigma_c$ and $\Gamma$.

- Input: A graph $G$, a set $\Sigma_c$ of graph dependencies, and a block $\Gamma$ of ground truth.
- Output: Fixes to all errors in $\mathsf{Vio}(\Sigma_c, G)$ by chasing graph $G$ with $(\Sigma_c, \Gamma)$.

We refer to this as the *graph cleaning problem.* It is even more challenging than error detection.

## 4 CONCLUSION

This paper provides an overview of recent advances in the study of graph dependencies, from formalism and theory to practice. The study, however, has raised as many question as it has answered. Below we highlight several topics for future work in this line of research.

*(1) Other forms of graph dependencies*. We have only presented counterparts of relational FDs, CFDs and EGDs on graphs. Extensions of other well-studied relational dependencies, *e.g.,* TGDs, also deserve a full treatment over graphs. For instance, association rules on graphs [14, 15] are a special case of TGDs on graphs and have found applications in social media marketing.

*(2) Applications of graph dependencies*. Graph dependencies are expected to find a wide range of applications. We have only discussed graph cleaning. Practical techniques need to be developed for, *e.g.,* social media marketing and fraud detection, based on graph dependencies. Even for graph cleaning, scalable algorithms are not yet in place for (incrementally) fixing errors with certain.

*(3) Scalable techniques*. The analyses of graph dependencies are expensive. While parallel computation helps, we need other scalable processing techniques for effective applications of graph dependencies, especially approximation algorithms (*e.g.,* [13]) with provable accuracy guarantees.

*(4) Explainable AI*. Graph dependencies naturally characterize associations and regularity of real-world entities, and are promising to interpret, *e.g.,* the outcome of machine learning classifiers. An interesting topic is to combine machine learning and reasoning about graph dependencies.

## REFERENCES


[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.
[2] Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. 2010. Constraints in RDF. In *SDKB*. 23–39.
[3] William Ward Armstrong. 1974. Dependency Structures of Data Base Relationships. In *IFIP Congress*. 580–583.
[4] Diego Calvanese, Wolfgang Fischl, Reinhard Pichler, Emanuel Sallinger, and Mantas Simkus. 2014. Capturing Relational Schemas and Functional Dependencies in RDFS. In *AAAI*.
[5] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of Constraints in RDFS. In *AMW*. 75–90.
[6] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for Graphs. *PVLDB* 8, 12 (2015), 1590–1601.
[7] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management.* Morgan & Claypool Publishers.
[8] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional Functional Dependencies for Capturing Data Inconsistencies. *TODS* 33, 1 (2008).
[9] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2018. Discovering Graph Functional Dependencies. In *SIGMOD*.
[10] Wenfei Fan, Xueli Liu, and Yingjie Cao. 2018. Parallel Reasoning of Graph Functional Dependencies. In *ICDE*.
[11] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2018. Catching Numeric Inconsistencies in Graphs. In *SIGMOD*.
[12] Wenfei Fan and Ping Lu. 2017. Dependencies for Graphs. In *PODS*. 403–416.
[13] Wenfei Fan, Xin Wang, and Yinghui Wu. 2014. Querying Big Graphs within Bounded Resources. In *SIGMOD*. 301–312.
[14] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association Rules with Graph Patterns. *PVLDB* 8, 12 (2015).
[15] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Adding Counting Quantifiers to Graph Patterns. In *SIGMOD*.
[16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional Dependencies for Graphs. In *SIGMOD*.
[17] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.
[18] Ivana Grujic, Sanja Bogdanovic-Dinic, and Leonid Stoimenov. 2014. Collecting and Analyzing Data from E-Government Facebook Pages. In *ICT Innovations*.
[19] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). W3C Working Draft. (Feb. 2017). *https://www.w3.org/TR/shacl/#dfn-shacl-instance.*
[20] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1990. A Complexity Theory of Efficient Parallel Algorithms. *TCS* 71, 1 (1990), 95–132.
[21] Georg Lausen, Michael Meier, and Michael Schmidt. 2008. SPARQLing Constraints for RDF. In *EDBT*. 499–509.
[22] Neo4j Team. 2017. The Neo4j Developer Manual v3.1 (Chapter 3.5.2: Constraints). (2017). *http://neo4j.com/docs/developer-manual/current/.*
[23] Gorka Sadowksi and Philip Rathle. 2014. Fraud Detection: Discovering Connections with Graph Databases. *http://asiandatascience.com/wp-content/uploads/2018/01/Neo4j_WP-Fraud-Detection-with-Graph-Databases.pdf.* (2014).
[24] Yang Yu and Jeff Heflin. 2011. Extending Functional Dependency to Detect Abnormal Data in RDF Graphs. In *ISWC*.