Catching Numeric Inconsistencies in Graphs

Wenfei Fan[1,2] Xueli Liu[3] Ping Lu[2] Chao Tian[1,2]
[1]University of Edinburgh [2]Beihang University [3]Harbin Institue of Technology
{wenfei@inf., chao.tian@}ed.ac.uk, xueli.hit@gmail.com, luping@buaa.edu.cn

ABSTRACT

Numeric inconsistencies are common in real-life knowledge bases and social networks. To catch such errors, we propose to extend graph functional dependencies with linear arithmetic expressions and comparison predicates, referred to as NGDs. We study fundamental problems for NGDs. We show that their satisfiability, implication and validation problems are Σ_2^p-complete, Π_2^p-complete and coNP-complete, respectively. However, if we allow non-linear arithmetic expressions, even of degree at most 2, the satisfiability and implication problems become undecidable. In other words, NGDs strike a balance between expressivity and complexity.

To make practical use of NGDs, we develop an incremental algorithm IncDect to detect errors in a graph G using NGDs, in response to updates ΔG to G. We show that the incremental validation problem is coNP-complete. Nonetheless, algorithm IncDect is localizable, *i.e.,* its cost is determined by small neighbors of nodes in ΔG instead of the entire G. Moreover, we parallelize IncDect such that it guarantees to reduce running time with the increase of processors. Using real-life and synthetic graphs, we experimentally verify the scalability and efficiency of the algorithms.

KEYWORDS

numeric errors; graph dependencies; incremental validation

1 INTRODUCTION

A variety of dependencies have recently been studied for graphs [8, 15, 23, 24, 33, 56]. These dependencies are often defined in terms of graph patterns, and aim to capture inconsistencies among entities in a graph. They are useful in, *e.g.,* knowledge acquisition, knowledge base enrichment, and spam detection in social networks.

However, semantic inconsistencies in real-life graphs often involve numeric values. To catch such errors, arithmetic calculation and comparison predicates are often a must. These expressions are, unfortunately, not supported by existing graph dependencies.

Example 1: Consider the following inconsistencies taken from real-life knowledge bases and social graphs.

(1) Yago. It is recorded that an institute BBC Trust was created in 2007 but destroyed in 1946, as shown in graph G_1 of Fig. 1. To detect this, we need to check whether wasDestroyedOnDate - wasCreatedOndate $\geq c$ for a constant c. However, neither arithmetic operator $-$ nor comparison predicate $\geq$ is supported by existing proposals for graph dependencies.

(2) Yago. A village Bhonpur in India is claimed to have 600 females and 722 males, but its total population is 1572 (see G_2 of Fig. 1). To catch this, we need an arithmetic equation femalePopulation + malePopulation = populationTotal.

(3) DBpedia. There are two cities, Corona and Downey, in California. Based on the 2014 population census, it is known that Corona has a larger population than Downey. However, Downey is ranked ahead of Corona in population (11th vs. 33rd; see G_3 of Fig. 1). The inconsistency should be checked by using a condition that x.population $<$ y.population implies x.populationRank $>$ y.populationRank, where x and y denote places.

(4) Twitter. Suppose that two accounts refer to the same company. If the two substantially differ in the numbers of their followers and followings, then the one with less followers and followings is likely to be a fake account [44]. To specify this rule, we need a condition $a\times(x.\text{follower} - y.\text{follower}) + b\times(x.\text{following} - y.\text{following}) > c$, for accounts x and y, and constants a, b and c. The condition is specified by both arithmetic expressions and comparison predicate. It helps us find, *e.g.,* fake account NatWest_Help in G_4. □

The example raises several questions. How should we extend graph dependencies to catch numeric errors? Does the extension make it harder to reason about the dependencies? Can we strike a balance between the expressive power and complexity? Can we uniformly catch inconsistencies in real-life graphs, numeric or not?

Contributions. This paper tackles these questions.

(1) NGDs. We propose a class of numeric graph dependencies, referred to as NGDs (Section 3). NGDs are a combination of (a) a pattern Q to identify entities by graph homomorphism, and (b) an attribute dependency $X \rightarrow Y$ on the entities identified. They extend graph functional dependencies (GFDs [23, 24]) by supporting linear arithmetic expressions and built-in comparison predicates $=, \neq, <, \leq, >, \geq$. We show that NGDs are able to catch numeric inconsistencies commonly found in real-life graphs. Moreover, they subsume GFDs [23, 24] and relational conditional functional dependencies (CFDs [19]) as special cases. Thus they are able to capture all inconsistencies that can be detected by GFDs and CFDs, besides numeric errors that are beyond the capacity of GFDs and CFDs.

(2) Fundamental results. We study two classical problems for reasoning about NGDs (Section 4), stated as follows.

- The *satisfiability* problem is to decide whether a given set Σ of NGDs has a model *i.e.,* a graph satisfying Σ.
- The *implication* problem is to decide whether a set Σ of NGDs entails another NGD φ, *i.e.,* for all graphs G that satisfy Σ, G also satisfies φ.

These problems are not only of theoretical interest, but also find practical applications. The satisfiability analysis enables us to check whether a set of NGDs is consistent themselves before the NGDs are used as, *e.g.,* data quality rules. The implication analysis helps us remove redundant rules, and optimize the data cleaning process.

SIGMOD/PODS '18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183753

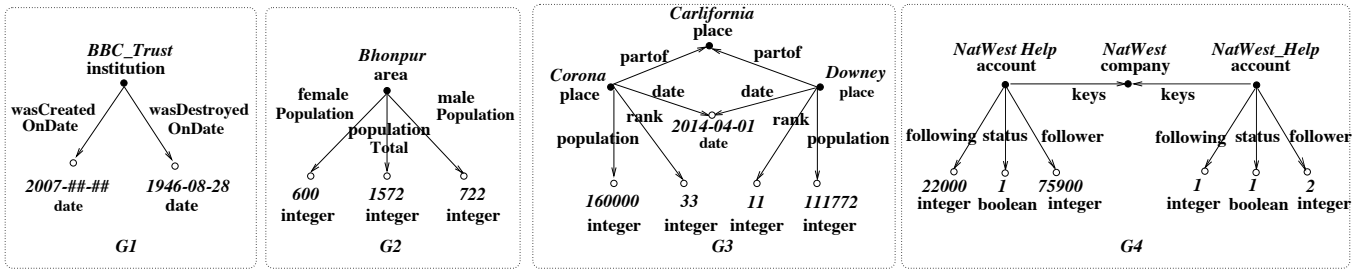

Figure 1: Numeric inconsistencies in real-life graphs

(a) We show that the increased expressive power of NGDs comes with a price. Their satisfiability and implication problems become Σ_2^p-complete and Π_2^p-complete, as opposed to coNP-complete and NP-complete for GFDs, respectively [23, 24]. The complexity bounds are robust: they remain Σ_2^p-hard and Π_2^p-hard, respectively, even when only equality = is used, in the absence of $\neq, <, \leq, >, \geq$, or when no arithmetic operations are used at all. These tell us that unless P = NP, it is harder to reason about NGDs than about GFDs.

(b) We show that if we expand NGDs by allowing non-linear arithmetic expressions, then both problems become undecidable, even when the degree of the arithmetic expressions is at most 2, and even in the absence of comparison predicates $\neq, <, \leq, >, \geq$.

The undecidability results justify the choice of linear arithmetic expressions. That is, NGDs strike a balance between expressivity and complexity when arithmetic and comparison are a must.

(3) Practical applications. We develop techniques for detecting inconsistencies in real-life graphs, numeric or not, by employing NGDs as data quality rules (Sections 5 and 6).

(a) We show that the *validation problem* is coNP-complete for NGDs, to decide whether a given graph satisfies a set of NGDs. The complexity is the same as for GFDs [23, 24]. That is, NGDs do not complicate the process of error detection. Better still, the parallel algorithms developed in [24] for detecting errors with GFDs can be readily extended to NGDs, retaining the same complexity.

(b) In light of this, we focus on incremental inconsistency detection in graphs, a problem that has not been studied by previous work, to the best of our knowledge (Section 5). Given a graph G and a set Σ of NGDs, suppose that we have already identified a set $\mathsf{Vio}(\Sigma, G)$ of violations of Σ in G, *i.e.,* entities in G that violate at least one NGD in Σ. We want to find *changes* $\Delta\mathsf{Vio}$ to $\mathsf{Vio}(\Sigma, G)$, such that

$$\mathsf{Vio}(\Sigma, G \oplus \Delta G) = \mathsf{Vio}(\Sigma, G) \oplus \Delta\mathsf{Vio},$$

where $X \oplus \Delta X$ denotes X updated by ΔX.

The need for incremental detection is evident. Real-life graphs G are often big, *e.g.,* the social graph of Facebook has billions of nodes and trillions of edges [31]. Error detection is expensive (coNP-complete). Moreover, real-life graphs are constantly changed. It is often too costly to recompute $\mathsf{Vio}(\Sigma, G \oplus \Delta G)$ starting from scratch in response to frequent ΔG. These highlight the need for incremental algorithms. We use (an extension of) the batch algorithms of [24] to compute $\mathsf{Vio}(\Sigma, G)$ once, and then incrementally compute changes $\Delta\mathsf{Vio}$ in response to ΔG. The rationale behind this is that in the real world, changes are typically small, *e.g.,* less than 5% on the entire Web in a week [45]. When ΔG is small, $\Delta\mathsf{Vio}$ is often small as well, and is much less costly to compute than $\mathsf{Vio}(\Sigma, G \oplus \Delta G)$ by making use of previous computation for $\mathsf{Vio}(\Sigma, G)$.

(c) While desirable, the incremental detection problem is nontrivial. We show that the problem is also coNP-complete, even when both graphs G and updates ΔG have *constant sizes* (Section 5).

(d) In response to the practical need, we develop two algorithms for incremental error detection with NGDs (Section 6), which make incremental error detection feasible in large-scale graphs.

One is a sequential *localizable* algorithm IncDect. It incrementalizes subgraph search by *update-driven evaluation*. Its cost is determined by the d_Σ-neighbors of nodes in ΔG, where d_Σ is the maximum diameter of the patterns in Σ [20]. In practice, Σ is much smaller than G, and so is d_Σ. It reduces the computations on (possibly big) graphs G to smaller d_Σ-neighbors of those nodes in ΔG.

The other one is a parallel algorithm PIncDect. We show that it is *parallel scalable* relative to IncDect: its cost is $O(t(|G|, |\Sigma|, |\Delta G|)/p)$, where p is the number of processors used, and $t(|G|, |\Sigma|, |\Delta G|)$ is the cost of IncDect. That is, PIncDect guarantees to reduce running time when more processors are used. We propose a *hybrid strategy* to split skewed work units and dynamically balance workload, based on cost estimation, to balance computation and communication.

(4) Experimental study. Using real-life and synthetic graphs, we empirically evaluate the scalability and efficiency of our algorithms (Section 7). We find the following. (a) Incremental error detection with NGDs is effective: IncDect is on average 6.7 times faster than its batch counterpart when $|\Delta G|$ accounts for 10% of $|G|$, and still does better even when $|\Delta G|$ is 33% of $|G|$. (b) The incremental algorithms scale well with G. (c) PIncDect is parallel scalable and efficient: it is on average 3.7 times faster when the number p of processors increases from 4 to 20. It takes 225s on graphs of 28 million nodes and 33.4 million edges when $p = 20$. (d) Hybrid workload balancing improves the performance by 1.73 times on average.

The novelty of the work consists of (1) NGDs to capture semantic inconsistencies in graphs, numeric or not; (2) fundamental results for NGDs, demonstrating the complications introduced by arithmetic expressions and comparison predicates (see below); and (3) the first incremental error detection algorithms for graphs.

Related work. We categorize the related work as follows.

Dependencies for graphs. Dependencies have been studied for RDF [8, 15, 17, 33, 41, 56], and for generic graphs [23, 24]. This line of work started from [41]. It extends RDF vocabulary to define keys, foreign keys and functional dependencies (FDs). Using triple patterns with variables, [8, 15] interpret FDs with triple embedding and homomorphism. A class of FDs was also formulated in [56] with path patterns; these FDs were extended in [33] to support CFDs. [13, 28] study a class of first-order Horn clause on binary predicates as soft constraints to facilitate knowledge base reasoning.

Closer to this work are GFDs on general graphs [24], defined in terms of (a) a graph pattern Q that is interpreted via subgraph isomorphism, and (b) an extension of an FD carrying constant and variable literals. GFDs are extended to graph entity dependencies (GEDs) in [23], by supporting literals with node identities to express keys of [17], interpreted via graph homomorphism.

This work defines NGDs by extending GFDs, and interprets pattern matching by graph homomorphism following [23]. It differs from [23, 24] in the following. (1) NGDs support both arithmetic operations and comparison predicates. (2) As shown by the fundamental results, the presence of either arithmetic or comparsion makes satisfiability and implication problems Σ_2^p-complete and Π_2^p-complete, respectively, as opposed to NP-complete and coNP-complete for GFDs and GEDs. The good news is that they do not complicate the validation problem. (3) We develop the first (parallel) incremental error detection algorithms for graphs with performance guarantees, which complement the batch ones for GFDs [24].

Dependencies on numeric data. Several dependency classes have been studied for detecting numeric errors in relations [16, 25, 27, 30, 38, 50]. Metric functional dependencies [38] and sequential dependencies [30] extend FDs by supporting (numeric) metrics and intervals on ordered data, respectively. Differential dependencies [50] constrain distances of numeric attribute values among different tuples. However, none of these supports arithmetic operations. There has also been work on repairing numeric data using constraints defined in terms of aggregate functions [25] and disjunctive logic programming [27]. Their satisfiability and implication problems are open, and the complexity is suspected high. Numeric functional dependencies (NFDs) [16] extend CFDs and support linear arithmetic expressions and built-in predicates like NGDs.

This work differs from the prior work as follows. (1) NGDs are defined on schemaless graphs with a graph pattern and an attribute dependency. They cannot be expressed as dependencies of [16, 30, 38, 50]. As shown in [23], GFDs, a special case of NGDs, are not expressible even as equality-generating dependencies with constants, which subsume CFDs. As an evidence, the validation problem is coNP-complete for NGDs and GFDs, but is in polynomial time (PTIME) for CFDs [19] and NFDs [16]. (2) The techniques for handling graph dependencies are quite different. For instance, we make use of the data locality of graph homomorphism to check NGDs (Section 6), a departure from relational dependencies. (3) To strike a balance between the complexity and expressivity, we do not consider aggregations; in fact, most numeric errors we encounter in real-life graphs can be caught without using aggregations.

Comparison predicates have been included in dependencies for data exchange [7] and views for query rewriting [5, 6]. However, (1) the comparisons in [5–7] are posed over dense orders, whereas we study linear arithmetic constraints over integers, whose satisfiability problem is NP-complete [46], as opposed to PTIME for densely ordered domains. (2) Chasing with NGDs, *e.g.,* testing satisfiability, always terminates [23], but the chase in [7] may not. (3) [5–7] do not consider any arithmetic operators supported by NGDs.

Algorithms for error detection. Error detection has been studied for relations [21, 47, 54] and RDF [37, 53, 55]. [21] studies (incremental) CFD validation in horizontally or vertically partitioned relations. A continuous framework is developed in [54] to clean relations that may change, using FDs that may also evolve. The method of [47] combines logical and quantitative data cleaning by using metric FDs. Consistency checking in RDF is conducted by logical reasoning on [53], or by unsupervised detection of numerical outliers [55]. [37] detects errors in RDF with SPARQL queries. On general graphs, [51] studies repairing of vertex labels to make graphs satisfy a form of neighborhood constraints. Batch algorithms are developed for catching violations of GFDs [24] or keys [17] in graphs, in parallel.

Different from the prior work, (1) we provide the first incremental error detection algorithms that are localizable [20] and relatively parallel scalable. As far as we know, none of the previous error detection algorithms is parallel scalable except the batch ones of [17, 24]. However, they cannot be directly incrementalized. Localizable algorithms have only been developed for graph queries [20], *e.g.,* keyword search. (2) We propose update-driven search and a hybrid dynamic strategy to achieve relative parallel scalability. The strategy balances the workload at run time, at two levels: (a) it makes use of cost estimation to split and distribute stragglers, and (b) it monitors the status of processors and reassigns work units from a busy processor to those with a light load. While (b) is along the same lines as work stealing and shedding [11, 32], we find that it does not work very well alone unless in combination with (a).

Incremental detection of NGD violations is more intriguing than conventional graph pattern matching: we have to compute violations that are newly introduced or removed by updates only. As a consequence, previous algorithms for parallel pattern matching, *e.g.,* [34, 40], cannot be applied directly in this context.

2 PRELIMINARIES

We first review basic notations. Assume alphabets Γ, Θ and U denoting labels, attributes and constant values, respectively.

Graphs. We consider directed *graphs* $G = (V, E, L, F_A)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; (3) each node v in V (resp. edge e in E) carries label $L(v)$ (resp. $L(e)$) in Γ, and (4) for each node v, $F_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$ such that $A_i \neq A_j$ if $i \neq j$, where a_i is a constant in U, and A_i is an *attribute* of v drawn from Θ, written as $v.A_i = a_i$, carrying the content of v such as keywords and blogs as found in social networks.

We use two notions of subgraphs. A graph $G' = (V', E', L', F'_A)$ is a *subgraph of* $G = (V, E, L, F_A)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$, $L'(v) = L(v)$ and $F'_A(v) = F_A(v)$; similarly for each edge $e \in E'$, $L'(e) = L(e)$.

A subgraph G' is *induced by* a set V' of nodes if $V' \subseteq V$ and E' consists of all the edges in E whose endpoints are 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. $L_Q(e)$) in Γ to each pattern node $u \in V_Q$ (resp. edge $e \in E_Q$), (3) $\bar{x}$ is a list of distinct variables; and (4) μ is a bijective mapping from $\bar{x}$ to V_Q, *i.e.,* it assigns a distinct variable to each node v in V_Q.

For $x \in \bar{x}$, we use $\mu(x)$ and x interchangeably when it is clear in the context. We allow wildcard '_' as a special label in $Q[\bar{x}]$.

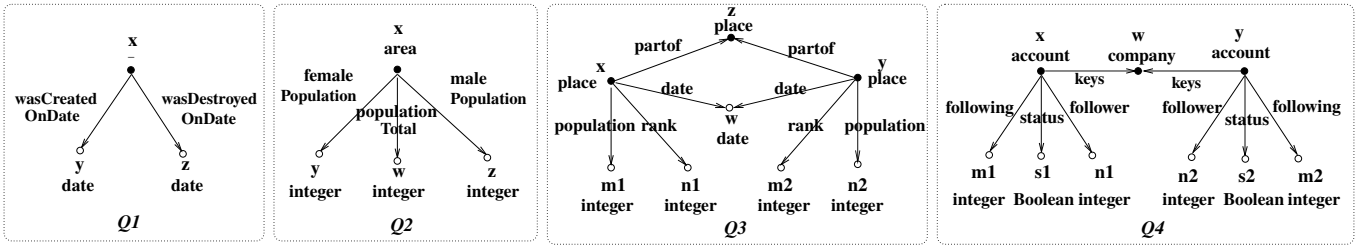

Figure 2: Graph patterns

Example 2: Four graph patterns are shown in Fig. 2. Here Q_1 depicts an entity x connected to date y and z with edges labeled wasCreatedOndate and wasDestroyedOndate, respectively. Node x is labeled '_', denoting arbitrary entities regardless of their labels. In G_1 of Fig.1, x is mapped to BBC_Trust. Similarly, Q_2–Q_4 can be interpreted by referencing their counterparts in Fig. 1. □

Pattern matching. We adopt the homomorphism semantics of matching following [8, 15, 23]. A *match* of pattern $Q[\bar{x}]$ in graph G is a mapping h from Q to G such that (a) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (b) for each $e = (u, u')$ in Q, $e' = (h(u), h(u'))$ is an edge in G and $L_Q(e) = L(e')$. Here $L_Q(u) = L(h(u))$ if $L_Q(u)$ is '_', *i.e.,* wildcard matches any label to indicate generic entities.

We denote the match as a vector $h(\bar{x})$, consisting of $h(x)$ for all $x \in \bar{x}$, in the same order as $\bar{x}$. Intuitively, $\bar{x}$ is a list of entities to be identified by Q, and $h(\bar{x})$ is such an instantiation in G.

3 NUMERIC GRAPH DEPENDENCIES

We extend the GFDs of [23, 24] to incorporate arithmetic expressions and built-in predicates. We start with basic notations.

Literals. Consider a graph pattern $Q[\bar{x}]$. A *term* of $Q[\bar{x}]$ is either an integer c in U or an integer "variable" $x.A$, where $x \in \bar{x}$ and A is an attribute in Θ (note that attributes are not specified in Q).

A *linear arithmetic expression* e *of* $Q[\bar{x}]$ is defined as

$$e ::= t \mid |e| \mid e + e \mid e - e \mid c \times e \mid e \div c$$

where t is a term, c is an integer, and $|e|$ is the absolute value of e. We consider *linear expression* e, *i.e.,* its degree is at most 1, where the *degree* of e is the sum of the exponents of its variables (*e.g.,* $x.A$).

For instance, all the arithmetic expressions given in Example 1 are linear. As will be seen in Section 4, we adopt linear e to strike a balance between the expressive power and complexity.

A *literal* l *of* $Q[\bar{x}]$ is of the form $e_1 \otimes e_2$, where e_1 and e_2 are linear arithmetic expressions of $Q[\bar{x}]$, and $\otimes$ is one of the built-in comparison operators $=, \neq, <, \leq, >$ and $\geq$.

NGDs. A *numeric graph dependency*, denoted by NGD, is of the form $Q[\bar{x}](X \rightarrow Y)$, where

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

Intuitively, NGD φ is a combination of (a) a *topological constraint* Q, to identify entities in a graph, and (b) an *attribute dependency* $X \rightarrow Y$, defined with linear arithmetic expressions connected with built-in predicates, to be enforced on the entities identified by Q.

NGDs extend GFDs of [23, 24] by supporting

(a) linear arithmetic expressions with $+, -, \times, \div$ and $|\cdot|$, and
(b) comparisons with built-in predicates $=, \neq, <, \leq, >, \geq$.

In other words, GFDs of [23, 24] are a special case of NGDs when literals are restricted to terms connected with equality '=' only, *i.e.,* literals of the form $x.A = c$ or $x.A = x.B$.

Example 3: To catch those errors spotted in Example 1, we define the following NGDs, in terms of the patterns depicted in Fig. 2.

(1) Yago. NGD $\varphi_1 = Q_1[x, y, z](\emptyset \rightarrow z.\text{val} - y.\text{val} \geq c)$. Here X is empty set $\emptyset$ and Y includes a single literal. From Q_1 of Fig. 2, we can see that x, y and z denote an entity, the date when it was created and the date when it was destroyed, respectively; val is an attribute for the integer values of y and z in days (not shown in Q_1); and c is a constant integer. It states that an entity cannot be destroyed within c days of its creation. It catches the error in G_1 of Fig. 1.

(2) Yago. NGD $\varphi_2 = Q_2[w, x, y, z](\emptyset \rightarrow y.\text{val} + z.\text{val} = w.\text{val})$. The NGD says that in any area x, its total population w.val should equal the sum of its female population y.val and its male population z.val. It catches the inconsistency in graph G_2.

(3) DBPedia. NGD $\varphi_3 = Q_3[\bar{x}](m_1.\text{val} < m_2.\text{val} \rightarrow n_1.\text{val} > n_2.\text{val})$, where $\bar{x}$ includes x and y in the same area z. It states that if the population m_1.val of x is less than the population m_2.val of y in the same census w, then the populationRank n_1.val of x is behind the populationRank n_2.val of y. It captures the inconsistency in G_3.

(4) Twitter. NGD $\varphi_4 = Q_4[\bar{x}](\{s_1.\text{val} = 1, a{\times}(m_1.\text{val} - m_2.\text{val}) + b{\times}(n_1.\text{val} - n_2.\text{val}) > c\} \rightarrow s_2.\text{val} = 0)$. Here $\bar{x}$ includes two accounts x and y about the same company w, where x (resp. y) has n_1.val (resp. n_2.val) followers and is following m_1.val (resp. m_2.val) people; and has status s_1.val (resp. s_2.val) indicating the realness. Integers a and b specify the weights of following and followers, respectively; and c is the threshold for their difference (see Example 1). It states that if the gap between the followers and followings of a real account x and account y exceeds c, then the chances are that y is fake. It catches NatWest_Help in G_4 as a fake account.

Note that φ_4 cannot be expressed as NFDs [16], since NFDs do not support preconditions with arithmetic operations. □

As shown in [23], GFDs can express (a) conditional functional dependencies (CFDs [19]) and (b) equality generating dependencies (EGDs [4]) when relational tuples are represented as vertices in a graph. Since NGDs subsume GFDs, NGDs can also express CFD and EGDs. In particular, NGDs support constant bindings of CFDs [19], which have proven useful in detecting errors in relations [18]. Hence, NGDs can catch non-numeric inconsistencies that GFDs and CFDs can detect, in addition to numeric errors.

Semantics. Consider a match $h(\bar{x})$ of Q in a graph G.

We say that match $h(\bar{x})$ *satisfies* a literal $l = e_1 \otimes e_2$ of $Q[\bar{x}]$ if (a) for each term $x.A$ in l, node $v = h(x)$ carries attribute A, and (b) $h(e_1) \otimes h(e_2)$, where $h(e_i)$ denotes the arithmetic expression obtained from e_i by substituting $h(x)$ for each x in e_i for $i \in [1, 2]$; here $h(e_1) \otimes h(e_2)$ is interpreted following the standard semantics of arithmetic operations and build-in predicates.

For instance, for $e_1 > e_2$, where e_1 is $x.A + x.B$ and e_2 is 3, $h(x)$ satisfies $e_1 > e_2$ if (a) node $v = h(x)$ carries attributes A and B, and (b) the value of $v.A + v.B$ is greater than 3.

For a set Z of literals, we write $h(\bar{x}) \models Z$ if $h(\bar{x})$ satisfies *all* literals in Z, *i.e.,* their conjunction. We write $h(\bar{x}) \models X \rightarrow Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$, *i.e.,* if $h(\bar{x}) \models X$, then $h(\bar{x}) \models Y$.

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

Intuitively, to check whether $G \models \varphi$, we need to examine all matches $h(\bar{x})$ of Q in G. We check whether $h(\bar{x}) \models Y$ if $h(\bar{x})$ is a match of Q and it satisfies the precondition X.

Example 4: Consider graph G_1 of Fig. 1 and NGD φ_1 of Example 3. Then $G_1 \not\models \varphi_1$, since there exists a match $h(x, y, z)$: $x \mapsto$ BBC_Trust, $y \mapsto$ 2007-#-# and $z \mapsto$ 1946-08-28, such that $h(y).\text{val} > h(z).\text{val}$, *i.e.,* $h(x, y, z) \not\models Y$. That is, $h(x, y, z)$ denotes entities that make a *violation* of φ_1 in G_1. Similarly, $G_2 \not\models \varphi_2$, $G_3 \not\models \varphi_3$ and $G_4 \not\models \varphi_4$. □

4 FUNDAMENTAL PROBLEMS FOR NGDS

We next study two fundamental problems associated with NGDs. The main conclusion of this section is that the presence of either linear arithmetic expressions or built-in predicates necessarily makes these problems harder unless P = NP. Nonetheless, NGDs pay a minimum price for supporting both arithmetic and comparison, striking a balance between the complexity and expressivity.

(1) Satisfiability. We consider two notions of satisfiability.

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

We say that Σ is *strongly satisfiable* if there exists a graph G such that (a) $G \models \Sigma$, and (b) for each NGD $Q[\bar{x}](X \rightarrow Y)$ in Σ, there exists a match $h_Q(\bar{x})$ of Q in G. Intuitively, condition (b) requires that all graph patterns in Σ find a model in G, to ensure that the NGDs in Σ do not conflict with each other.

The *satisfiability problem* for NGDs is to decide, given a set Σ of NGDs, whether Σ is satisfiable. The *strong satisfiability problem* is to decide whether Σ is strongly satisfiable.

Example 5: Consider a set Σ_0 consisting of two NGDs: $\varphi_5 = Q[x](\emptyset \rightarrow x.A = 7 \wedge x.B = 7)$ and $\varphi_6 = Q[x](\emptyset \rightarrow x.A + x.B = 11)$, where Q has a single node x labeled '_'. Then there exist nonempty graphs that satisfy φ_5 and φ_6 when taken separately. However, φ_5 and φ_6 are not satisfiable when put together. Indeed, the values of attributes A and B on each node must be 7 as required by φ_5, while their sum is required to be 11 by φ_6, which is impossible.

Suppose that pattern Q in φ_6 is replaced by Q' that has a single node labeled 'a'. Then Σ_0 becomes satisfiable. Indeed, consider graph G having a single node v labeled 'b' with $v.A = v.B = 7$. Then $G \models \Sigma_0$. However, Σ_0 is not strongly satisfiable, since for any graph G', if all patterns in Σ_0 find a match in G', then there must exist nodes labeled 'a', and the conflicts above arise again.

Similarly, one can verify that the NGDs below are not (strongly) satisfiable: $\varphi_7 = Q[x](x.A \leq 3 \rightarrow x.B > 6)$, $\varphi_8 = Q[x](x.A > 3 \rightarrow x.B > 6)$, and $\varphi_9 = Q[x](\emptyset \rightarrow x.B < 6 \wedge x.A \neq 0)$. □

These show that the presence of either linear arithmetic expressions or built-in comparison predicates beyond equality makes the satisfiability analysis more intriguing than that of GFDs [23, 24].

(2) Implication. A set Σ of NGDs *implies* another NGD φ, denoted by $\Sigma \models \varphi$, if for all graphs G, if $G \models \Sigma$, then $G \models \varphi$. That is, the NGD φ is a logical consequence of the set Σ of NGDs.

The *implication problem* for NGDs is to determine, given a set Σ of NGDs and another NGD φ, whether $\Sigma \models \varphi$.

As remarked in Section 1, the practical need for studying these problems is evident, besides theoretical interest, for determining whether data quality rules discovered from possibly dirty data are sensible, and for optimizing data quality rules, among other things.

Complexity. We next settle the complexity of these problems. The proofs of the results below are quite involved. For the lack of space, we provide proof sketches for all the results of the paper, and defer the detailed proofs for satisfiability of Theorems 1 and 3 to [**?**].

Recall that the satisfiability problem for relational functional dependencies (FDs) is trivial, *i.e.,* for any set Σ of FDs over a relation schema R, there always exists a nonempty database instance of R that satisfies Σ [18]. Moreover, the implication problem for FDs is in linear-time (cf. [4]). It is known that the satisfiability and implication problems for GFDs are coNP-complete and NP-complete [24], respectively. These are comparable to their counterparts for relational CFDs, which are NP-complete and coNP-complete, respectively [19]. In contrast, NGDs make our lives harder.

Theorem 1: *For* NGDs*, (a) the satisfiability problem and strong satisfiability problems are both Σ_2^p-complete, and (b) the implication problem is Π_2^p-complete.* □

Here Σ_2^p is the class of decision problems that are solvable in NP by calling an NP oracle, *i.e.,* $\Sigma_2^p = \text{NP}^{\text{NP}}$. It is considered more intriguing than NP unless P = NP. Similarly, $\Pi_2^p = \text{coNP}^{\text{NP}}$, which is also above NP in the polynomial hierarchy (see [46] for details).

Proof: (1) For the upper bound of the satisfiability problem, we first show a small model property: if a set Σ of NGDs is satisfiable, then there exists a graph G_Σ of size at most $3(|\Sigma| + 1)^5$ such that $G_\Sigma \models \Sigma$. The proof needs attribute normalization so that the total size of the attributes in G_Σ is also bounded by a function of $|\Sigma|$. This is verified by means of bounded-length solution to the linear programming problem [14] preserving the satisfiability of the conditions enforced by Σ, and by transforming all built-in predicates to inequality $\leq$.

Based on this property, we give an Σ_2^p algorithm to check whether a given set of NGDs is satisfiable: guess a graph G_m such that $|G_m| \leq 3(|\Sigma| + 1)^5$ and a pattern Q from Σ, and check whether (a) Q has a match in G_m in PTIME and (b) $G_m \models \Sigma$ in coNP.

We prove the lower bound by reduction from the generalized subset sum problem (GSSP) [49]. GSSP is to decide, given two vectors $\bar{u}_1$ and $\bar{u}_2$ of integers, and another integer w, whether $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$, where $\bar{v}_1$ and $\bar{v}_2$ refers to Boolean vectors, and $\bar{u}_1^T$ (resp. $\bar{u}_2^T$) is the transpose of $\bar{u}_1$ (resp. $\bar{u}_2$). It is known that GSSP is Σ_2^p-complete [49]. In the reduction, we use a set of three NGDs that share the same pattern Q to encode GSSP. We encode the existential semantic of vector $\bar{v}_1$ with one NGD, such that there are m nodes carrying A-attributes with Boolean values in its model. The universal semantic of vector $\bar{v}_2$ is encoded by another NGD and wildcards in the graph pattern to arbitrarily match two nodes with value 0 and 1, respectively, of another attribute B. The third NGD encodes $\bar{u}_1$ and $\bar{u}_2$, and checks the condition in GSSP.

(2) The proof above is extended to the strong satisfiability problem. In particular, to prove the small model property, we build such a model that all patterns in Σ find a match in it. The lower bound proof of the satisfiability problem in (1) carries over here, since all NGDs in the set Σ used there share the same pattern.

(3) We also show a small model property for the implication problem: if $\Sigma \not\models \varphi$, then there exists a graph $G_{(\Sigma,\varphi)}$ that "witnesses" $\Sigma \not\models \varphi$, *i.e.,* $G_{(\Sigma,\varphi)} \models \Sigma$ while $G_{(\Sigma,\varphi)} \not\models \varphi$, such that $|G_{(\Sigma,\varphi)}| \leq 3(|\Sigma| + |\varphi| + 1)^5$. Given this, we develop an Σ_2^p algorithm to check whether $\Sigma \not\models \varphi$. The Π_2^p-hardness is shown by reduction from the complement of GSSP, using NGD φ and Σ of a single NGD.

None of the lower bound proofs uses $\neq, <, \leq, >, \geq$. □

One might think that the complexity comes from interactions between arithmetic operations and comparison predicates. This is not the case: the lower bounds still hold when either arithmetic expressions or built-in predicates are present, not necessarily both.

Corollary 2: *For* NGDs*, the satisfiability, strong satisfiability and implication problems remain Σ_2^p-complete, Σ_2^p-complete and Π_2^p-complete, respectively, even in the absence of either (a) arithmetic operations, or (b) comparison predicates $\neq, <, \leq, >, \geq$.* □

Proof: The upper bounds inherit from Theorem 1, as well as the lower bounds in the absence of $\neq, <, \leq, >, \geq$, since the reductions given there use no such predicates. However, new lower bound proofs are required for NGDs that carry only comparison predicates. We show these by reductions from their counterparts for an extension of GEDs with built-in predicates only [23]. □

Undecidability. One might want to support arithmetic expressions that are not necessarily linear, defined as

$$e ::= t \mid |e| \mid e + e \mid e - e \mid e \times e \mid e \div e.$$

That is, e is built up from terms by closing them under arithmetic operators, *not necessarily of degree at most 1.* A literal is defined as $e_1 \otimes e_2$ as before, where e_1 and e_2 are arithmetic expressions of $Q[\bar{x}]$, and $\otimes$ is one of $=, \neq, <, \leq, >, \geq$.

This extension, however, makes the static analyses undecidable, even for NGDs with literals of a bounded degree. The undecidability justifies our choice of linear arithmetic expressions for NGDs.

Theorem 3: *The satisfiability, strong satisfiability and implication problems become undecidable for* NGDs *extended with non-linear arithmetic expressions, even when*

- *no arithmetic expressions in the* NGDs *have degree above 2,*
- *and none of $\neq, <, \leq, >, \geq$ predicate is present.* □

Proof: We show the undecidability for the extended NGDs by reductions from the Hilbert's 10th problem (HTP) [35, 43] and its complement, respectively. HTP is to decide, given a polynomial Diophantine equation $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$, where $a_1, \dots, a_n$ are integer coefficients and $n_{1,i}, \dots, n_{m,i}$ are non-negative integer exponents for each $i \in [1, n]$, whether there exists an integer solution for $(y_1, \dots, y_m)$. It is known that HTP is undecidable [43].

The reductions are a little complicated. For the (strong) satisfiability problem, we use a set Σ consisting of a single NGD to encode the computation of polynomials in the given Diophantine equation in a recursive manner, and to check the existence of feasible solutions. Moreover, the degree of each arithmetic expression in Σ is at most 2, and only built-in predicate $=$ is used. Especially, we decompose each polynomial $a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}}$ into coefficient and exponentiation that are represented by distinct nodes in the pattern of the NGD, and encode its computation recursively through the encoding of sub-expression $a_i y_1^{n_{1,i}} \dots y_{m-1}^{n_{m-1,i}}$ and suffix exponentiation $y_m^{n_{m,i}}$. For implication, we use a singleton set Σ and another NGD φ to encode the equation, again in a recursive manner. □

5 DETECTING ERRORS WITH NGDS

We have seen that NGDs provide uniform rules for capturing inconsistencies in graphs, numeric or not (Section 3). We next study error detection in graphs by using NGDs as data quality rules.

5.1 Detecting Inconsistencies in Graphs

To state the error detection problem, we borrow the following notations from [24]. Given an NGD $\varphi = Q[\bar{x}](X \to Y)$ and a graph G, we say that a match $h(\bar{x})$ of Q in G is a *violation* of φ if $G_h \not\models \varphi$, where G_h is the subgraph induced by $h(\bar{x})$. For a set Σ of NGDs, we denote by $\mathrm{Vio}(\Sigma, G)$ the set of all violations of NGDs in G, *i.e.,* $h(\bar{x}) \in \mathrm{Vio}(\Sigma, G)$ if there exists an NGD φ in Σ such that $h(\bar{x})$ is a violation of φ in G. That is, $h(\bar{x})$ violates at least one NGD in Σ.

The *error detection problem* is stated as follows.

- *Input*: A set Σ of NGDs and a graph G.
- *Output*: The set $\mathrm{Vio}(\Sigma, G)$ of violations.

That is, when NGDs in Σ are used as data quality rules, it is to find the set $\mathrm{Vio}(\Sigma, G)$ of all inconsistent entities in G.

Its decision version is the *validation problem* to decide, given a set Σ of NGDs and a graph G, whether $G \models \Sigma$, *i.e.,* $\mathrm{Vio}(\Sigma, G) = \emptyset$.

It is known that the validation problem for GFDs is coNP-complete [24]. The good news is that the problem gets no harder for NGDs, despite their increased expressive power.

Corollary 4: *The validation problem for* NGDs *remains* coNP*-complete.* □

Proof: We develop an NP algorithm to check whether $\mathrm{Vio}(\Sigma, G) \neq \emptyset$: guess an NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ and a mapping h from Q to G, and check whether (a) h is a match of Q in G; and (b) $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$. Checking (a) and (b) are in PTIME. Hence NGD validation is in coNP. The lower bound follows from [23], since NGDs subsume GFDs, and GFD validation is coNP-complete. □

Using GFDs as data quality rules, parallel algorithms have been developed for error detection [24]. The algorithms are *parallel scalable, i.e.,* they guarantee to reduce the running time of a yardstick sequential algorithm when more processors are used (see Section 6.1). Hence they can scale with real-life graphs by adding resources when the graphs grow big. The experimental study of [24] has validated the parallel scalability and efficiency of the algorithms.

A close examination of the algorithms of [24] reveals that the algorithms can be readily extended to NGDs. Indeed, for the algorithms to work with NGDs on a graph G that is fragmented and distributed across different processors, the only change involves local checking of NGDs in each fragment of G, by adding arithmetic and comparison calculations; the generation of matches of graph patterns, which dominates the cost of the algorithms, remains unchanged. The workload estimation and balancing strategies of [24] remain intact for NGDs. These strategies make the algorithms parallel scalable. As a result, the algorithms remain parallel scalable when they employ NGDs instead of GFDs.

Hence there are parallel scalable algorithms to uniformly detect semantic inconsistencies in graphs, numeric or not, with NGDs.

5.2 Incremental Error Detection

Error detection is costly in large G, and real-life graphs are frequently updated. This highlights the need for studying incremental error detection: we compute $\mathsf{Vio}(\Sigma, G)$ once, and then incrementally compute $\mathsf{Vio}(\Sigma, G \oplus \Delta G)$ in response to updates ΔG to G. This is more efficient than recomputing $\mathsf{Vio}(\Sigma, G \oplus \Delta G)$ starting from scratch when ΔG is small, as often found in practice.

We define a unit update as edge insertion (insert e) or deletion (delete e), which can simulate certain modifications. An insertion possibly introduces new nodes carrying labels, attributes, and values drawn from Γ, Θ and U, respectively, while deletions just remove the links, leaving the nodes otherwise unaffected. We formalize the problem as follows. We consider *batch update* ΔG consisting of a sequence of insertions and deletions of edges. Denote by

$\Delta\mathsf{Vio}^+(\Sigma, G, \Delta G) = \mathsf{Vio}(\Sigma, G \oplus \Delta G) \setminus \mathsf{Vio}(\Sigma, G)$,
$\Delta\mathsf{Vio}^-(\Sigma, G, \Delta G) = \mathsf{Vio}(\Sigma, G) \setminus \mathsf{Vio}(\Sigma, G \oplus \Delta G)$,
$\Delta\mathsf{Vio}(\Sigma, G, \Delta G) = (\Delta\mathsf{Vio}^+(\Sigma, G, \Delta G), \Delta\mathsf{Vio}^-(\Sigma, G, \Delta G))$,

new errors introduced by ΔG, removed by ΔG and their combination, respectively. The *incremental error detection problem* is:

- *Input*: Graph G, NGDs Σ, and batch update ΔG to G.
- *Output*: The changes $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ to $\mathsf{Vio}(\Sigma, G)$.

We do not require $\mathsf{Vio}(\Sigma, G)$ as part of the input, since the set may be exponential in size and is costly to store.

It is not surprising that the problem is nontrivial. Its decision problem is to decide whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G) = \emptyset$.

Theorem 5: *It is* coNP*-complete to decide, given a set* Σ *of* NGDs*, a graph* G *and a batch update* ΔG*, whether* $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ *is empty, even when both* G *and* ΔG *have constant sizes.* □

Proof: To prove the upper bound, we give an NP algorithm to check, given Σ, G and ΔG, whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G) \neq \emptyset$. The lower bound is verified by reduction from the complement of the 3-colorability problem [46]. The latter problem is to decide, given an undirected graph G, whether there exists a proper 3-coloring γ of G such that for each edge (u, v) in G, $\gamma(u) \neq \gamma(v)$. The problem is NP-complete [46]. The reduction uses a graph G' of 3-clique and a single NGD $Q[\bar{x}](\emptyset \rightarrow x_1.A = 3)$ to encode the 3 colors and the structure of graph G, respectively. The tricky part is to encode an undirected graph G using an NGD, which is defined on directed graphs (see Section 3) and also used for verifying possible 3-colorings. The reduction employs a constant-size graph G', and a batch update $\Delta G'$ of 3 edge insertions (constant size) with the same edge label. □

In the rest of the paper we focus on (parallel) algorithms for incrementally detecting inconsistencies in graphs, by using NGDs. The algorithms complement the batch algorithms of [24], for NGDs used as data quality rules. As remarked earlier, we are not aware of prior work on incremental error detection in graphs, and the static workload partitioning strategy of [24] hampers the parallel scalability of the batch algorithms when being incrementalized.

6 INCREMENTAL DETECTION ALGORITHMS

Despite the challenges noted in Theorem 5, we develop two practical algorithms to incrementally detect errors in graphs with NGDs. We show that the algorithms have certain performance guarantees.

We first review the performance guarantees (Section 6.1). We then present a sequential incremental error detection algorithm (Section 6.2), followed by a parallel algorithm (Section 6.3).

To simplify the discussion, we focus on NGDs defined with graph patterns Q that are connected, *i.e.,* there exists a path between any two vertices in Q when Q is treated as an undirected graph. The algorithms can be readily extended to process NGDs that are defined with possibly disconnected patterns. More specifically, one can first compute (candidate) partial violations, by finding matches of distinct connected components in Q following the same update-driven approach to be given shortly in this section. These partial matches are then combined to evaluate attribute dependencies. that go across multiple connected components, to identify violations.

6.1 Performance Guarantees

We first review two characterizations of the effectiveness of (parallel) incremental error detection algorithms.

(1) Locality. We first adapt a criterion from [20]. (a) In a graph G, a node v' is *within d hops* of v if $\mathsf{dist}(v, v') \leq d$ by taking G as an undirected graph, where $\mathsf{dist}(v, v')$ is the shortest distance between v and v' in G. (b) We denote by $V_d(v)$ the set of all nodes in G that are within d hops of v. (c) The *d-neighbor* of v, denoted by $G_d(v)$, is the subgraph of G induced by $V_d(v)$ (see Section 2).

The *diameter* d_Q of a pattern Q is the minimum $\mathsf{dist}(v, v')$ for all nodes v and v' in Q. For a set Σ of NGDs, *the diameter* d_Σ of Σ is the maximum diameter d_Q for all patterns Q that appear in Σ.

An incremental error detection algorithm $\mathcal{A}$ is *localizable* if given a set Σ of NGDs, a graph G, and a batch update ΔG to G, its cost is determined only by the size $|\Sigma|$ of NGDs and the sizes of the d_Σ-neighbors of those nodes on the edges of ΔG.

The notion of *localizable* was proposed in [20] and has proven effective for graph queries, by reducing computation on a (large) graph to small areas surrounding ΔG. We specialize it to incremental validation, to compute $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ by checking only $G_{d_\Sigma}(v)$ for nodes v that appear in ΔG. In practice, $G_{d_\Sigma}(v)$ is often small. Indeed, (a) Q is typically small, *e.g.,* 98% of real-life patterns have radius 1 [29], which also indicate patterns in rules [28]; and (b) G is sparse, *e.g.,* the average node degree is 14.3 in social graphs [12].

(2) Parallel scalability. The second criterion is adapted from [39], which has been widely used in practice to characterize the effectiveness of parallel algorithms. Consider a sequential algorithm $\mathcal{A}$ for incremental error detection, with cost $t(|G|, |\Sigma|, |\Delta G|)$ measured in the sizes of graph G, Σ of NGDs and batch update ΔG.

A parallel algorithm $\mathcal{A}_p$ for incremental error detection is said to be *parallel scalable relative* to yardstick $\mathcal{A}$ if its parallel running time by using p processors can be expressed as follows:

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

where $p \ll |G|$, *i.e.,* the number of processors is much smaller than real-life graphs G, as commonly found in the real world.

Intuitively, parallel scalability measures speedup over sequential algorithms by parallelization. It is a relative measure *w.r.t.* a yardstick algorithm $\mathcal{A}$. A parallel scalable $\mathcal{A}_p$ "linearly" reduces the running time of $\mathcal{A}$ when p increases. Hence a parallel scalable algorithm is able to scale with large G by adding processors as needed. It makes incremental detection feasible by increasing p.

6.2 A Sequential Localizable Algorithm

We first develop an exact localizable algorithm, denoted by IncDect. Given a set Σ of NGDs, a graph G and a batch update ΔG, IncDect computes $\Delta\text{Vio}(\Sigma, G, \Delta G)$ with a single processor. Algorithm IncDect incrementalizes subgraph matching by following *update-driven evaluation*, and checks dependencies with arithmetic.

Subgraph matching. We start by reviewing the general framework of subgraph matching, denoted as Match_n.

A number of subgraph matching algorithms have been developed for graphs, mostly following a backtracking-based procedure Match_n [42]. Given a pattern Q and a graph G, Match_n first identifies a set $C(u)$ of candidate matches for each node u in Q. Then its main subroutine SubMatch_n recursively expands partial solution M, by matching one pattern node of Q with a node of G in each round, where M is a set of node pairs (u, v) indicating that v matches pattern node u. Subgraph homomorphism algorithms [26, 48] can also be characterized by the generic Match_n and SubMatch_n.

More specifically, given a partial solution M, SubMatch_n selects a pattern node u that is not yet matched, and refines $C(u)$ following certain matching order selection and pruning strategies. For each refined candidate v in $C(u)$, it checks whether v can make a valid match of u by inspecting the correspondence between edges adjacent to u in Q and those edges connected to v in G. The qualified node pair (u, v) is added to M, and SubMatch_n is called recursively for further expansion, until all the pattern nodes are matched. The partial solution M is restored when SubMatch_n backtracks.

Algorithm. IncDect incrementalizes batch algorithm Match_n to process G, Σ and $\Delta G = (\Delta G^+, \Delta G^-)$, where ΔG^+ and ΔG^- include $\text{insert}(v, v')$ and $\text{delete}(v, v')$, respectively. (1) It starts with $\Delta\text{Vio}^+(\Sigma, G, \Delta G) = \emptyset$ and $\Delta\text{Vio}^-(\Sigma, G, \Delta G) = \emptyset$. (2) For each NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, it invokes a procedure IncMatch revised from Match_n to expand $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (resp. $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) with those matches $h(\bar{x})$ of Q in $G \oplus \Delta G$ (resp. G) such that (a) $h(u) = v$ and $h(u') = v'$ for some $(u, u') \in E_Q$ and $\text{insert}(v, v')$ in ΔG^+ (resp. $\text{delete}(v, v')$ in ΔG^-), and (b) $h(\bar{x}) \not\models X \to Y$.

Intuitively, edge insertions may introduce new violations and hence expand $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$, but do not remove existing ones; on the other hand, deletions expand $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$ only. IncMatch computes the newly added (resp. removed) violations; this is done by identifying those matches that have nodes connected by edges in ΔG^+ (resp. ΔG^-) and violating the attribute dependency.

Procedure IncMatch. We next give details of IncMatch and its subroutine IncSubMatch for processing NGD $Q[\bar{x}](X \to Y)$. Following *update-driven* evaluation, we extend Match_n and SubMatch_n to conduct (1) initial partial solution selection; (2) candidates filtering; and (3) arithmetic and comparison calculations.

(1) Given pattern Q, IncMatch first finds out whether each edge (v, v') in ΔG is a candidate match of some pattern edge (u, u') in Q by checking the labels. This is in contrast to Match_n that searches candidates in the entire graph. If (v, v') makes a candidate, it forms an initial partial solution $h_{\text{up}}(u, u') = (v, v')$, referred to as an *update pivot* of Q *triggered by* unit update of edge (v, v'). IncMatch then expands $h_{\text{up}}(u, u')$ by recursively invoking IncSubMatch as in Match_n to compute update-driven violations $h(\bar{x})$.

(2) In each call, IncSubMatch searches candidates from the neighbors of those nodes that are already in a partial solution, starting from the update pivot. Each time IncSubMatch picks a pattern node that is connected to some already matched ones. For a match $h(\bar{x})$ of Q to be included in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$, (a) it must be expanded from a pivot triggered by insertion, and (b) there exist no v and v' in $h(\bar{x})$ such that $h(u) = v$ and $h(u') = v'$ for any $(u, u') \in E_Q$ while $\text{delete}(v, v')$ is in ΔG^-. Therefore, it leaves out edges in ΔG^- when retrieving candidates to expand the solutions from update pivots triggered by edge insertions. Similarly, it does not consider edges $\text{insert}(v, v')$ in ΔG^+ when expanding $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$.

As an optimization strategy, IncMatch marks the combination of multiple update pivots in partial solutions to prevent the same match from being enumerated more than once.

(3) The validation of arithmetic expressions is performed by applying candidate pruning in IncSubMatch. More specifically, it evaluates a literal l in X as long as all the variables are instantiated, *i.e.,* every variable in l is already matched or is being matched by the candidates under process, and prunes those when l is evaluated to be false. Literals in Y are handled similarly except that candidates contributing to true evaluations are pruned. Indeed, only matches $h(\bar{x})$ that satisfy $h(\bar{x}) \models X$ and $h(\bar{x}) \not\models Y$ are returned as violation.

Finally, those matches expanded from update pivots triggered by edge insertions (resp. deletions) and violating $X \to Y$, referred to as *update-driven violations*, are returned by IncMatch and added to $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (resp. $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) by algorithm IncDect.

Example 6: Suppose that the edge (NatWest Help, 1) is deleted from G_4 of Fig. 1. Given NGD φ_4 of Example 3, IncDect calls IncMatch to detect update-driven violations. It first finds that the deleted edge is a candidate match of (x, s_1) in Q_4. That is, an update pivot $h_{\text{up}}(x, s_1)$ = (NatWest Help, 1) is built. IncMatch then expands $h_{\text{up}}(x, s_1)$ recursively by inspecting the neighbors of candidate matches until all pattern nodes of Q_4 are matched. For instance, node 22000 in G_4 is the only candidate match for m_1. Finally, it returns violation $h_{\text{up}}(\bar{x})$ to be removed, which includes all the nodes of G_4, and NatWest_Help is found a fake account.

Besides delete(NatWest Help, 1), suppose that four edges are inserted into G_4 to indicate that another account NatWest_Help_1 has 1 following and 2 followers, and refers to company NatWest with status 1. Given this batch update, IncDect computes the same violation to be removed as above. Indeed, there are no newly introduced violations since all matches expanded from update pivots triggered by edge insertions are pruned by literal validation. □

Analysis. The correctness of IncDect is warranted by the following. The violations in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (resp. $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) are matches of Q in $G \oplus \Delta G$ (resp. G) that contain inserted (resp. deleted) edges of ΔG and violate dependency $X \to Y$ for an NGD $Q[\bar{x}](X \to Y)$ in Σ, *i.e.,* update-driven violations found by IncMatch.

IncDect runs in $O(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|})$ time, where $G_{d_\Sigma}(\Delta G)$ denotes the union of d_Σ-neighbors of nodes involved in ΔG. Hence it is localizable. Indeed, (a) the computation performed by each invocation of IncMatch is confined in the d_Σ-neighbors of an unit update in ΔG. (b) The cost of checking linear arithmetic expressions is much less than the cost of candidate selection in matching.

6.3 A Parallel Scalable Algorithm

Algorithm IncDect takes exponential time in the worst case. It is costly if Σ or ΔG is large, or G is dense. This motivates us to develop algorithm PIncDect that is parallel scalable relative to IncDect, to reduce response time by adding more processors when needed.

Overview. Algorithm PIncDect works with p processors $S_1, \ldots, S_p$ on a graph G that is partitioned via edge-cut [9] or vertex-cut [36]. In a nutshell, PIncDect finds update pivots of patterns in Σ triggered by unit updates, and distributes these partial solutions as work units to p processors. Then each processor handles its workload and identifies violations *in parallel*, driven by updates like in IncDect.

However, there are two challenges. (1) The d_Σ-neighbor of a node may reside in different fragments. (2) The workloads of some processors may be skewed, since (a) the workload assignment may be unbalanced; and (b) some work unit may take much longer, *e.g.,* when accessing a large d_Σ-neighbor. Note that work stealing and shedding [11, 32] do not solve (b) by re-assigning work units.

To cope with this, PIncDect does the following. It finds and distributes the *candidate neighborhood* of each update pivot. Then all the processors interact with each other asynchronously to expand and verify partial solutions, by accessing the candidate neighborhoods only. To reduce skewness, PIncDect (a) splits and parallelizes the *work unit* of filtering and verifying a candidate, based on cost estimation, and (b) periodically redistributes the partial solutions (work units) to be expanded from busy processors to those with light loads. This makes PIncDect parallel scalable relative to IncDect.

Candidate neighborhood. Similar to IncDect, initially PIncDect checks whether each unit update of edge (v, v') in ΔG triggers an update pivot $h_{\text{up}}(u, u') = (v, v')$ of some pattern nodes u and u' in Q, at each processor. It then identifies the d_Q-neighbor of v in $G \oplus \Delta G^+$, *i.e.,* the *candidate neighborhood* $N_C(h_{\text{up}}(u, u'))$ for $h_{\text{up}}(u, u')$. When node v is involved in multiple update pivots, only the union of their neighborhoods is extracted. The processors coordinate to extract such an area when it is fragmented, by notifying each other the remaining size to be explored via messages passed through crossing edges in edge cut or entry and exit nodes in vertex cut.

All processors broadcast the data extracted such that the union $N_C(\Delta G, \Sigma)$ of candidate neighborhoods for update pivots is replicated at each processor. We find that $N_C(\Delta G, \Sigma)$ is often much smaller than G when ΔG and Σ are small, as found in practice.

Moreover, for each node v in $N_C(\Delta G, \Sigma)$, PIncDect evenly "partitions" its adjacency list $v.\text{adj}$ by annotating local partition (instead of physically breaking it up), such that each processor S_i holds a *partial copy* $v.\text{adj}_i$. The update pivots are also evenly partitioned into p disjoint sets. Each S_i maintains one set BVio_i as its *workload*. A partial solution to be expanded is a *work unit*.

Parallel validation. All processors expand partial solutions to find update-driven violations in parallel. For each partial solution in BVio_i, S_i expands it by matching a pattern node that is not matched yet, until a complete violation is found. This is done by *candidate filtering* followed by *verification*. It adopts a hybrid processing strategy to split and parallelize skewed work units. Algorithm PIncDect also periodically balances workloads across p processors, to reduce skewed workloads with a large number of work units.

We next give the insights of the two steps for expanding partial solutions, which dominate the cost of algorithm PIncDect.

Candidate filtering. Consider $h_{\text{up}}(u_0, \ldots, u_k)$ in BVio_i, a partial solution for Q to be expanded at processor S_i. The next pattern node to be matched is u_{k+1} such that it is connected to u_r in Q for $r \in [0, k]$. The candidates for u_{k+1} are selected from the neighbors of $h_{\text{up}}(u_r)$, just like in procedure IncSubMatch (Section 6.2). Here it estimates the *sequential cost* as $|h_{\text{up}}(u_r).\text{adj}|$, and the *parallel cost* as

$$C \cdot (k+1) + |h_{\text{up}}(u_r).\text{adj}|/p,$$

for expanding the partial solution by matching u_{k+1}, where C is a constant referred to as *communication latency*, and $C \cdot (k+1)$ denotes the broadcasting cost. It conducts expansion at processor S_i directly by inspecting candidates from $h_{\text{up}}(u_r).\text{adj}$, if the sequential cost is less than the parallel one. Otherwise, $h_{\text{up}}(u_0, \ldots, u_k)$ is broadcast to all the processors, and is expanded in parallel by checking the partial copy $h_{\text{up}}(u_r).\text{adj}_j$ reserved at each S_j for $j \in [1, p]$. This allows us to reduce a skewed work unit with large adjacency lists.

Verification. After $h_{\text{up}}(u_0, \ldots, u_k)$ is expanded with u_{k+1} at processor S_i, PIncDect checks the edges between the candidate $h_{\text{up}}(u_{k+1})$ and other matches $h_{\text{up}}(u_0), \ldots, h_{\text{up}}(u_k)$, to verify the validity of the expansion. It may split the verification work. Here the sequential cost is estimated as $|h_{\text{up}}(u_{k+1}).\text{adj}|$ and the parallel cost is

$$C \cdot (k+2) + |h_{\text{up}}(u_{k+1}).\text{adj}|/p.$$

If the parallel cost is smaller, it broadcasts $h_{\text{up}}(u_0, \ldots, u_k, u_{k+1})$ to check at each S_j by using its partial copy $h_{\text{up}}(u_{k+1}).\text{adj}_j$; the results are sent back to S_i to decide the qualification of the partial solution, which are added to BVio_i for further expansion if qualified.

Workload balancing. The workload of a processor S_i is *skewed* if BVio_i contains far more work units than the others at the same time. This happens even if we start with evenly distributed update pivots, as different partial solutions may trigger radically different number of new work units. We define *the skewness of* S_i as $\frac{\|\text{BVio}_i\|}{\text{avg}_{t \in [1,p]} \|\text{BVio}_t\|}$.

To cope with this, PIncDect checks the skewness of processors at a time interval intvl. If the skewness of S_i exceeds a threshold η (3 in experiments), it evenly distributes the work units in BVio_i to those S_j's having skewness below η' (0.7 in experiments), extending BVio_j's. We allow processors to send and receive work units at any time, without being blocked by synchronization barriers.

Algorithm. Putting these together, we present the main driver of algorithm PIncDect in Fig. 3. It first identifies the candidate neighborhood for each update pivot (lines 1-3), and replicates their union at all processors (line 4). The update pivots are also evenly distributed (line 5). Then PIncDect invokes procedure PIncMatch at each processor S_i with initial workload BVio_i, in parallel for $i \in [1, p]$ (line 6). It periodically balances workload (line 8), until all processors complete their work (line 9). At this point, PIncDect collects local violations Vio_i's from all processors. The union of all Vio_i's is $\Delta\text{Vio}(\Sigma, G, \Delta G)$ (line 10) and is returned (line 11).

At each processor S_i, procedure PIncMatch expands a partial solution by filtering candidate matches (lines 3-6), followed by verification (lines 7-10). Both steps split skewed work units by applying the hybrid processing strategy based on cost estimation, as described earlier. The local violations Vio_i and workload BVio_i are updated accordingly (lines 11-13). It returns Vio_i when no work units remain in BVio_i, *i.e.,* when S_i finishes its workload (line 14).

Example 7: Consider a graph G revised from G_4 of Fig. 1 by including additional 98 accounts NatWest_Help_i for $i \in [1, 98]$, where

Algorithm: PIncDect
Input: A fragmented graph G across p processors $S_1, \ldots S_p$, a set Σ of NGDs, and a batch update ΔG.
Output: The set $\Delta\text{Vio}(\Sigma, G, \Delta G)$ of violations.

1. **for each** unit update of (v, v') in ΔG and pattern edge (u, u') in Q of NGD $\psi = Q[\bar{x}](X \to Y) \in \Sigma$ **having** $L_Q(u) = L(v)$, $L_Q(u') = L(v)$, **and** $L_Q(u, u') = L(v, v')$ **do**
2. construct update pivot $h_{\text{up}}(u, u') = (v, v')$;
3. identify the d_{Q_u}-neighbor of v;
4. construct $N_C(\Delta G, \Sigma)$ in parallel and replicate it at all processors;
5. evenly partition adjacency lists and work units across p processors;
6. invoke PIncMatch(BVio_i) at processor S_i for all $i \in [1, p]$;
7. **repeat**
8. periodically balance workload across p processors at interval intvl;
9. **until** all S_i's return Vio_i;
10. $\Delta\text{Vio}(\Sigma, G, \Delta G) := \bigcup_i \text{Vio}_i$;
11. **return** $\Delta\text{Vio}(\Sigma, G, \Delta G)$;

Procedure PIncMatch /* *executed at each worker* S_i *in parallel* */
Input: Workload BVio_i.
Output: The set Vio_i of local violations.

1. $\text{Vio}_i := \emptyset$;
2. **while** there **exists** a partial solution to be expanded **do**
3. **for each** $h_{\text{up}}(u_0, \ldots, u_k) \in \text{BVio}_i$ by matching u_{k+1} with neighbors of $h_{\text{up}}(u_r)$ **do**
4. **if** $|h_{\text{up}}(u_r).\text{adj}| \le C(k+1) + |h_{\text{up}}(u_r).\text{adj}|/p$ **then**
5. expand $h_{\text{up}}(u_0, \ldots, u_k)$ at S_i;
6. **else** broadcast $h_{\text{up}}(u_0, \ldots, u_k)$ and expand it in parallel;
7. **for each** $h_{\text{up}}(u_0, \ldots, u_k, u_{k+1})$ to be verified at S_i **do**
8. **if** $|h_{\text{up}}(u_{k+1}).\text{adj}| \le C(k+2) + |h_{\text{up}}(u_{k+1}).\text{adj}|/p$ **then**
9. verify $h_{\text{up}}(u_0, \ldots, u_{k+1})$ at S_i;
10. **else** broadcast $h_{\text{up}}(u_0, \ldots, u_{k+1})$ and verify it in parallel;
11. **if** $h_{\text{up}}(u_0, \ldots, u_k, u_{k+1})$ is a valid partial solution **then**
12. **if** it is a complete match **then** add it to Vio_i;
13. **else** add it to BVio_i;
14. **return** Vio_i;

Figure 3: Algorithm PIncDect

each one has 1 following and 2 followers and refers to company NatWest with status 1. Assume that G is fragmented across 4 processors. Recall NGD φ_4 and delete(NatWest Help, 1) from Example 6. After generating update pivot $h_{\text{up}}(x, s_1)$ as in Example 6, algorithm PIncDect identifies in parallel $N_C(h_{\text{up}}(x, s_1))$, which is the 3-neighbor of node NatWest Help. This subgraph is replicated at all 4 processors. Moreover, the adjacency lists are evenly "partitioned" by annotating partial copies. For instance, each processor maintains a partial copy of 25 nodes (*i.e.,* accounts) for the adjacency list of the company node NatWest. Then it expands $h_{\text{up}}(x, s_1)$.

Suppose that $h_{\text{up}}(x, s_1, m_1, n_1, w)$ is to be expanded at S_j, where w is mapped to NatWest, and the next to be matched is y. Then it is broadcast by S_j, and PIncDect expands it in parallel at each processor by mapping y to NatWest_Help_i for some $i \in [1, 98]$ or NatWest_Help, using the partial copies maintained for the adjacency list of NatWest. Here the estimated parallel cost 30 is less than the sequential cost 100; thus parallel computation is favored.

Now consider a partial solution of $h_{\text{up}}(x, s_1, m_1, n_1, w, y)$ to be expanded at processor S_j. Algorithm PIncDect expands it locally at S_j with the entire adjacency list of $h_{\text{up}}(y)$, since the size of $h_{\text{up}}(y).\text{adj}$, *i.e.,* sequential cost of 4, is less than the estimated parallel cost.

Finally, a total of 99 violations are identified and added to $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$, in which NatWest_Help_i and NatWest_Help are validated to be fake for each $i \in [1, 98]$. □

Theorem 6: PIncDect *is parallel scalable relative to* IncDect. □

Proof: We show that with p processors, PIncDect runs in $O(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}/p)$ time, where $p < |G_{d_\Sigma}(\Delta G)|$. Obviously, identifying the candidate neighborhoods for update pivots triggered by ΔG and Σ takes $O(|G_{d_\Sigma}(\Delta G)|)$ time. We next analyze the cost for parallel expansion. The total time for candidate filtering in processing partial solutions of size k is at most $N_k(k+1)(Ck+|G_{d_\Sigma}(\Delta G)|/p)$, and their verification needs at most $N_{k+1}(C(k+1)+|G_{d_\Sigma}(\Delta G)|/p))$ time, where N_k denotes the number of partial matches of size k. Moreover, it inspects partial solutions with size less than $|V_\Sigma|$, where V_Σ denotes the set of all pattern nodes in Σ. Hence parallel expansion takes at most $\sum_{k=2}^{|V_\Sigma|-1}\Big(N_k(k+1)(Ck+\frac{|G_{d_\Sigma}(\Delta G)|}{p}) + N_{k+1}(C(k+1)+\frac{|G_{d_\Sigma}(\Delta G)|}{p})\Big) < \frac{4C|\Sigma|(1-|G_{d_\Sigma}(\Delta G)|^{|\Sigma|-1})|G_{d_\Sigma}(\Delta G)|^2}{(1-|G_{d_\Sigma}(\Delta G)|)p} = O(\frac{|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}}{p})$ time, which dominates the cost of PIncDect. This verifies the relative parallel scalability of PIncDect. □

7 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we evaluated the impact of (a) the size $|\Delta G|$ of updates; (b) the size $|G|$ of graphs; (c) the complexity of sets Σ of NGDs; (d) the number p of processors, and the parameters C and intvl for workload balancing on our (parallel) incremental algorithm; and (e) the effectiveness of NGDs.

Experimental setting. We used three real-life graphs: (a) DBpedia [1], a knowledge base with 28 million entities of 200 types and 33.4 million edges of 160 types; (b) YAGO2, an extended knowledge graph of YAGO [52] with 3.5 million nodes of 13 types and 7.35 million edges of 36 types; and (c) Pokec [3], a social network with 1.63 million nodes of 269 types and 30.6 million links of 11 types. The density (defined as $\frac{|E|}{|V|\cdot(|V|-1)}$) is 6.5×10^{-7}, 6×10^{-7} and 1.1×10^{-5}, and the average diameter of connected components is 4.8, 4.0 and 5.2, for DBpedia, YAGO2 and Pokec, respectively.

We also generated synthetic graphs G (Synthetic) with labels and attributes drawn from an alphabet $\mathcal{L}$ of 500 symbols and values from a set of 2000 integers. It is controlled by the numbers of nodes $|V|$ and edges $|E|$, up to 80 million and 100 million, respectively.

NGDs. We extended the algorithm of [22] to discover NGDs from the graphs. The algorithm interleaves "vertical levelwise expansion" for mining frequent patterns Q and "horizontal levelwise expansion" for mining literals in $X \to Y$, in a single process. Numeric attributes are identified and composed to form expressions up to a predefined bound on the lengths. The NGDs discovered from a graph G are strongly satisfied by its subgraphs. We picked a set Σ of 100 meaningful and diverse NGDs for each graph from the discovered ones, such that at least 90% of them have different patterns, including trees, DAGs (directed acyclic graphs) and cyclic graphs. They carry patterns of diameters from 1 to 6, and 1 to 4 literals, with linear arithmetic expressions of lengths from 1 to 10.

ΔG. Updates ΔG to graph G are randomly generated, controlled by the size $|\Delta G|$ and a ratio γ of edge insertions to deletions. The ratio γ is 1 unless stated otherwise, *i.e.,* the size $|G|$ remains unchanged.

Algorithms. In Java, we implemented (1) sequential IncDect (Section 6.2) vs. Dect, a batch error detection algorithm with NGDs, extended from the algorithm for GFDs [24]; (2) parallel PIncDect

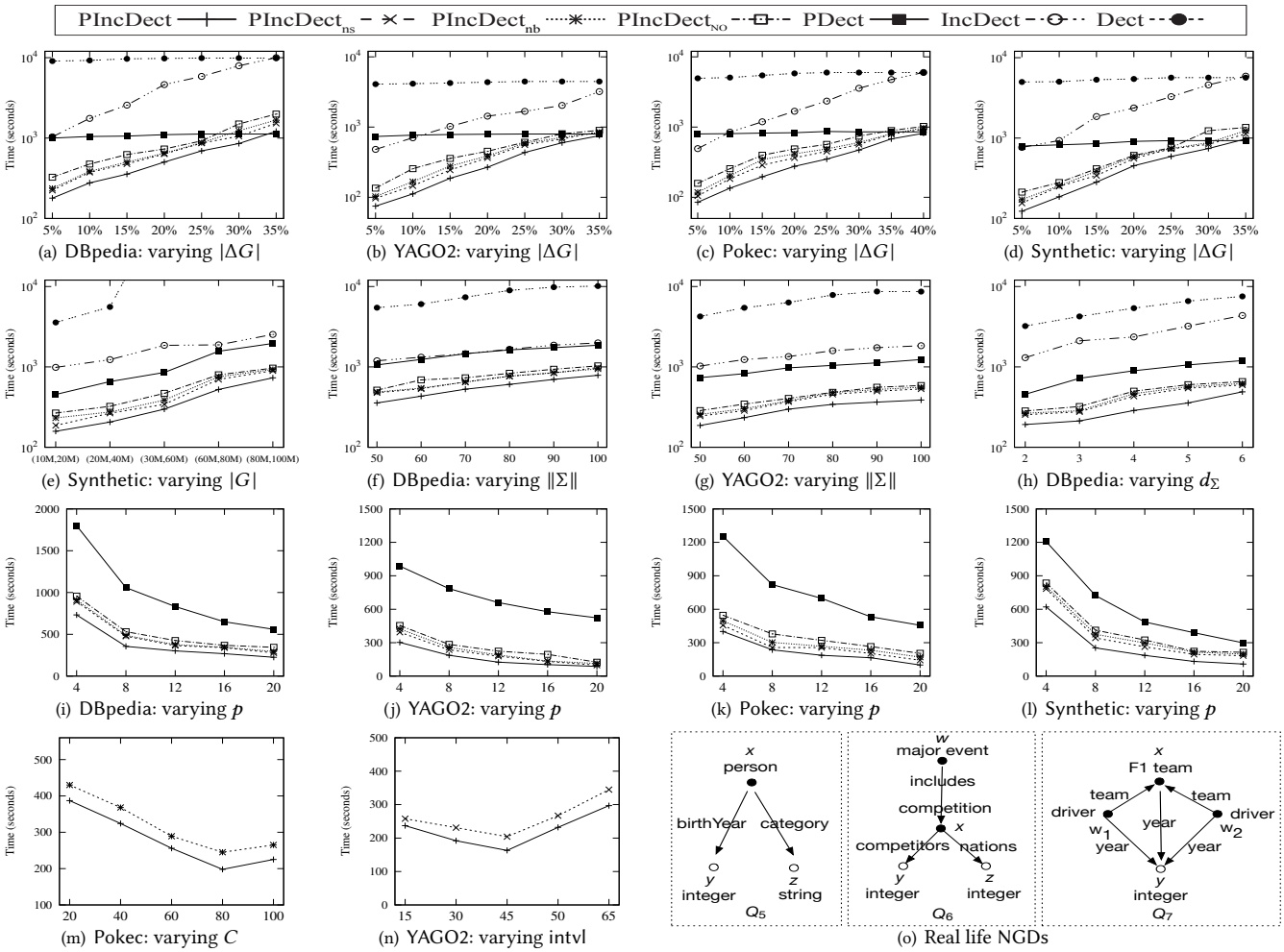

Figure 4: Performance evaluation

(Section 6.3) vs. PDect, an extension of the parallel batch detection algorithm in [24] to NGDs; and (3) parallel PIncDect$_{ns}$, PIncDect$_{nb}$ and PIncDect$_{NO}$, variants of PIncDect with no work unit splitting, no workload balancing, and neither of the two, respectively.

We deployed the algorithms on a cluster of up to 20 machines, each with 32GB DDR4 RAM and two 1.90GHz Intel(R) Xeon(R) E5-2609 CPU, running 64-bit CentOS7 with Linux kernel 3.10.0. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings. The graphs are fragmented using METIS [2]. We took Synthetic G with 40 million nodes and 60 million edges as default. We fixed the latency parameter $C = 60$, interval intvl = 45s, and the number of processors $p = 8$ for parallel algorithms unless stated otherwise.

Exp-1: Effectiveness of incremental error detection. We first evaluated the incremental algorithms against their batch counterparts. Fixing $\|\Sigma\| = 50$ and $d_\Sigma = 5$, we varied the size $|\Delta G|$ of updates from 5% up to 40% in 5% increments. The results are reported in Figures 4(a)–4(d) over DBpedia, YAGO2, Pokec and Synthetic G, respectively (y-axis in logarithmic scale). We find the following.

(a) When $|\Delta G|$ varies from 5% to 25% of $|G|$, IncDect is 8.8 to 1.7 (resp. 8.5 to 2.6, 9.8 to 2.6, and 6.6 to 1.7) times faster than Dect over the four graphs, respectively; PIncDect outperforms PDect by 5.6 to 1.6 (resp. 9.8 to 1.8, 9.4 to 2.5, and 5.6 to 1.6) times. PIncDect and IncDect beat their batch counterparts even when $|\Delta G|$ is 33% of $|G|$. These justify the need for incremental error detection.

(b) On average, PIncDect outperforms PIncDect$_{ns}$, PIncDect$_{nb}$ and PIncDect$_{NO}$ by 1.29, 1.33 and 1.61 times on DBpedia (resp. 1.31, 1.43, 1.81 on YAGO2, 1.33, 1.45, 1.81 on Pokec, and 1.27, 1.36, 1.5 on Synthetic) in the same setting. This verifies the effectiveness of our hybrid workload balancing strategy. It also suggests that workload balancing should be combined with work unit splitting.

(c) The larger $|\Delta G|$ is, the slower all incremental algorithms are, while the batch algorithms Dect and PDect are indifferent to $|\Delta G|$, as expected. In all cases, PIncDect performs the best.

(d) Incremental error detection is feasible in practice: PIncDect takes 693s on DBpedia when $|\Delta G|$ is 25% of $|G|$, and IncDect takes 5840s, as opposed to 1121s (resp. 9878s) by PDect (resp. Dect).

(e) All incremental algorithms are insensitive to the ratio γ of edge insertions to deletions, which is verified by varying γ (not shown).

(f) We found that the additional cost of checking linear arithmetic expressions is negligible (not shown). This confirms Corollary 4, *i.e.,* NGDs do not make the validation problem harder than GFDs.

Exp-2: Impact of $|G|$. We evaluated the impact of $|G|$ using synthetic graphs. Fixing $|\Delta G|$ as 15% of $|G|$ and using the same NGDs as

in Exp-1, we varied $|G|$ from (10M,20M) to (80M,100M). As shown in Fig. 4(e), (a) all the algorithms take longer on larger G, as expected, (b) incremental algorithms are less sensitive to $|G|$ than their batch counterparts, and (c) PIncDect does the best among all.

Exp-3: Complexity of NGDs. We also evaluated the impact of the complexity of sets Σ of NGDs. We fixed $|\Delta G| = 15\%|G|$.

Varying $\|\Sigma\|$. Fixing $d_\Sigma = 5$, we varied $\|\Sigma\|$ from 50 to 100 (our industry collaborator uses at most 95 rules [10]). As shown in Figures 4(f) and 4(g) on DBpedia and YAGO2, respectively, (a) the more NGDs are in Σ, the longer time is taken by all the algorithms, as expected, and (b) PIncDect and IncDect scale well with $\|\Sigma\|$. The results on Pokec and Synthetic are consistent (not shown).

Varying d_Σ. Fixing $\|\Sigma\| = 50$, we varied d_Σ from 2 to 6. Figure 4(h) shows that all algorithms take longer over larger d_Σ on DBpedia. This is consistent with our analysis that the costs of our incremental algorithms increase when d_Σ gets larger. Nonetheless, PIncDect is feasible with real-life NGDs, *e.g.,* it takes 489s on DBpedia when $d_\Sigma = 6$, as opposed to 1197s by PDect and 7532s by Dect. The results on YAGO2, Pokec and Synthetic are consistent.

Exp-4: Scalability of parallel algorithms. Using the same NGDs as in Exp-1 and fixing $|\Delta G| = 15\%|G|$ for all the graphs, we evaluated the scalability of parallel algorithm PIncDect vs. PDect, $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$ and $\text{PIncDect}_{\text{NO}}$, by varying the number p of processors, the parameter C of latency, and interval intvl.

Varying p. Fixing $C = 60$ and intvl = 45s, we varied p from 4 to 20. As shown in Figures 4(i), 4(j), 4(k) and 4(l) over DBpedia, YAGO2, Pokec and Synthetic, respectively, when p changes from 4 to 20, (a) PIncDect and PDect perform much better and are on average 3.7 and 3.8 times faster than IncDect and Dect, respectively, and (b) PIncDect consistently outperforms PDect, $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$ and $\text{PIncDect}_{\text{NO}}$: on average it is 2.47 to 3.14, 1.32 to 1.37, 1.44 to 1.53, and 1.53 to 1.72 times better, respectively.

These also verify the effectiveness of the hybrid workload partition strategy. It improves $\text{PIncDect}_{\text{NO}}$ from 1.53 to 1.72 times. Moreover, work unit splitting or workload balancing alone does not work very well, as verified by the gap between the performance of PIncDect and that of $\text{PIncDect}_{\text{ns}}$ and $\text{PIncDect}_{\text{nb}}$, respectively.

Varying C. Fixing $p = 8$ and intvl = 45s, we evaluated the impact of latency parameter on PIncDect and $\text{PIncDect}_{\text{nb}}$ by tuning C from 20 to 100 in 20 increments. As shown in Fig. 4(m) over Pokec, PIncDect performs the best when C is 80, taking 198s. On one hand, PIncDect favors parallel computation with smaller C to split work units; on the other hand, PIncDect has a bias towards local computation with larger latency C to reduce the communication cost. The results on DBpedia, YAGO2 and Synthetic are consistent.

Varying intvl. Fixing $p = 8$ and $C = 60$, we varied intvl from 15s to 65s in 15s increments, to evaluate impact of intervals for monitoring workloads on PIncDect and $\text{PIncDect}_{\text{ns}}$. As shown in Figure 4(n) on YAGO2, the "optimal" intvl is 45s for PIncDect. Similar to latency C, while smaller intvl helps workload balancing, it incurs more communication cost. Hence we need to strike a balance. The results on DBpedia, Pokec, and Synthetic are consistent.

Exp-5: Effectiveness study. The NGDs captured 415, 212, and 568 errors in DBpedia, YAGO2 and Pokec, respectively, ranging from wrong attribute values such as strings and numeric values, to structural errors, *i.e.,* incorrect relationship between entities. Among these, 92% can only be caught be NGDs, beyond the capacity of GFDs. Below are some errors caught, along with the NGDs.

NGD_1 is $Q_5[\bar{x}](y.\text{val} < 1800 \rightarrow z.\text{val} \neq \text{"living people"})$, stating that any person with birth year before 1800, *i.e.,* aged over 210, can no longer be categorized as living people. It identifies an error in DBpedia that a living person John Macpherson was born in 1713.

NGD_2 is $Q_6[\bar{x}](w.\text{type} = \text{"Olympic"} \rightarrow z.\text{val} \leq y.\text{val})$, which states that the number of participating nations in an Olympic event should not be larger than the number of competitors, *i.e.,* each athlete represents at most one nation. It detects that 24 athletes representing 34 countries participated in the Women's Sailboard Competition at the 1992 Summer Olympics, in DBpedia.

NGD_3 is $Q_7[\bar{x}](\emptyset \rightarrow x.\text{numberOfWins} \geq w_1.\text{numberOfWins} + w_2.\text{numberOfWins})$. This NGD states that in the Formula One racing, the total number of competitions won by two drivers is no larger than that of the team they represent during the same year. In DBpedia, it catches that Sebastian Vettel and Max Verstappen won 1 competition in 2016; however their team Scuderia Ferrari won none of the races. In fact, Max did not race for Ferrari in 2016. This shows that NGDs also help us detect the erroneous links.

Summary. We find the following. (1) Our incremental algorithms scale well with $|\Delta G|$, $|G|$, $\|\Sigma\|$ and d_Σ. IncDect and PIncDect outperform Dect from 6.7 to 2.1 times and from 52 to 13 times on average, respectively, when $|\Delta G|$ varies from 5% to 25% of $|G|$ over real-file and synthetic graphs. They perform better even when $|\Delta G|$ is up to 33% of $|G|$. (2) The incremental algorithms are much less sensitive to $|G|$ than the batch algorithms, and are able to deal with large-scale graphs. (3) Better still, parallel PIncDect scales well with the number p of processors used: its runtime is improved by 3.7 times on average when p increases from 4 to 20. (4) IncDect and PIncDect are feasible in practice: on real-life graphs, they take 1659s and 130s on average (with $p = 20$), respectively. (5) The hybrid workload balancing strategy is effective: it improves the performance of PIncDect by 1.73 times on average and works well with large p.

8 CONCLUSION

We have proposed a class of NGDs with arithmetic and comparison expressions to catch semantic inconsistencies in graphs. We have justified NGDs by establishing the complexity of the satisfiability and implication analyses of NGDs and their extensions. We have developed the first incremental algorithms to detect errors in graphs, with provable performance guarantees. We have empirically verified that NGDs and the algorithms yield a promising tool for detecting errors in graph-structured data, *numeric or not.*

One topic for future work is to extend NGDs by supporting aggregations. Another topic is to study graph repairing with NGDs.

Acknowledgments. The authors are supported in part by 973 2014CB340302, ERC 652976, NSFC 61421003, EPSRC EP/M025268/1, Beijing Advanced Innovation Center for Big Data and Brain Computing, and Joint Lab between Edinburgh and Huawei. Tian is also supported in part by NSFC 61602023. Liu is also supported by 973 2012CB316200, and is the corresponding author.

REFERENCES

[1] Dbpedia. *http://wiki.dbpedia.org/Datasets.*
[2] Metis. *http://glaros.dtc.umn.edu/gkhome/.*
[3] Pokec social network. *http://snap.stanford.edu/data/soc-pokec.html.*
[4] *S. Abiteboul, R. Hull, and V. Vianu.* Foundations of Databases. *Addison-Wesley, 1995.*
[5] *F. N. Afrati, C. Li, and P. Mitra. Answering queries using views with arithmetic comparisons. In* PODS, *pages 209–220, 2002.*
[6] *F. N. Afrati, C. Li, and P. Mitra. Rewriting queries using views in the presence of arithmetic comparisons.* Theor. Comput. Sci., *368(1-2):88–123, 2006.*
[7] *F. N. Afrati, C. Li, and V. Pavlaki. Data exchange in the presence of arithmetic comparisons. In* EDBT, *2008.*
[8] *W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in RDF. In* SDKB, *2010.*
[9] *K. Andreev and H. Racke. Balanced graph partitioning.* Theory of Computing Systems, *39(6):929–939, 2006.*
[10] *Baidu. Personal communication, 2017.*
[11] *R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing.* J. ACM, *46(5):720–748, 1999.*
[12] *P. Burkhardt and C. Waring. An NSA big graph experiment. Technical Report NSA-RD-2013-056002v1, U.S. National Security Agency, 2013.*
[13] *Y. Chen, S. L. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In* SIGMOD, *2016.*
[14] *W. Cook, A. M. Gerards, A. Schrijver, and E. Tardos. Sensitivity theorems in integer linear programming.* Mathematical Programming, *34(3):251–264, 1986.*
[15] *A. Cortés-Calabuig and J. Paredaens. Semantics of constraints in RDFS. In* AMW, *2012.*
[16] *G. Fan, W. Fan, and F. Geerts. Detecting errors in numeric attributes. In* WAIM, *2014.*
[17] *W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs.* PVLDB, *8(12), 2015.*
[18] *W. Fan and F. Geerts.* Foundations of Data Quality Management. *Morgan & Claypool Publishers, 2012.*
[19] *W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies.* TODS, *33(1), 2008.*
[20] *W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In* SIGMOD, *2017.*
[21] *W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. In* ICDE, *2012.*
[22] *W. Fan, X. Liu, P. Lu, C. Hu, and Y. Cao. Discovering graph functional dependencies. In* SIGMOD, *2018.*
[23] *W. Fan and P. Lu. Dependencies for graphs. In* PODS, *2017.*
[24] *W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In* SIGMOD, *2016.*
[25] *S. Flesca, F. Furfaro, and F. Parisi. Querying and repairing inconsistent numerical databases.* TODS, *35(2), 2010.*
[26] *F. V. Fomin, P. Heggernes, and D. Kratsch. Exact algorithms for graph homomorphisms.* Theory Comput. Syst., *41(2):381–393, 2007.*
[27] *E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: a challenging application of disjunctive logic programming. In* LPAR, *2001.*
[28] *L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In* WWW, *2013.*
[29] *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.*
[30] *L. Golab, H. J. Karloff, F. Korn, A. Saha, and D. Srivastava. Sequential dependencies.* PVLDB, *21(1), 2009.*
[31] *I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from e-government Facebook pages. In* ICT Innovations, *2014.*
[32] *A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In* SoCC, *2016.*
[33] *B. He, L. Zou, and D. Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In* SWIM, *2014.*
[34] *J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs.* PVLDB, *4(11):1123–1134, 2011.*
[35] *J. P. Jones. Undecidable Diophantine equations.* Bull. Amer. Math. Soc., *3(2), 1980.*
[36] *M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices.* Data Knowl. Eng., *72:285–303, 2012.*
[37] *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.*
[38] *N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In* ICDE, *2009.*
[39] *C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms.* TCS, *71(1):95–132, 1990.*
[40] *L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce.* PVLDB, *8(10):974–985, 2015.*
[41] *G. Lausen, M. Meier, and M. Schmidt. SPARQLing constraints for RDF. In* EDBT, *2008.*
[42] *J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases.* PVLDB, *6(2):133–144, 2012.*
[43] *Y. Matiyasevich.* Hilbert's 10th Problem. *The MIT Press, 1993.*
[44] *A. Murray. Fake natwest twitter account targets customers to steal bank details. http://www.telegraph.co.uk/money/consumer-affairs/fake-natwest-twitter-account- targets-customers-to-steal-bank-det.*
[45] *A. Ntoulas, J. Cho, and C. Olston. What's new on the Web? The evolution of the Web from a search engine perspective. In* WWW, *2004.*
[46] *C. H. Papadimitriou.* Computational Complexity. *Addison-Wesley, 1994.*
[47] *N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining quantitative and logical data cleaning.* PVLDB, *9(4):300–311, 2015.*
[48] *P. Rzażewski. Exact algorithm for graph homomorphism and locally injective graph homomorphism.* Inf. Process. Lett., *114(7):387–391, 2014.*
[49] *M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: A compendium.* SIGACT news, *33(3):32–49, 2002.*
[50] *S. Song and L. Chen. Differential dependencies: Reasoning and discovery.* TODS, *36(3):16, 2011.*
[51] *S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints.* PVLDB, *2014.*
[52] *F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In* WWW, *pages 697–706, 2007.*
[53] *F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: a self-organizing framework for information extraction. In* WWW, *2009.*
[54] *M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In* ICDE, *2014.*
[55] *D. Wienand and H. Paulheim. Detecting incorrect numerical data in dbpedia. In* ESWC, *2014.*
[56] *Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In* ISWC, *2011.*