Functional Dependencies for Graphs

Wenfei Fan[1,2] Yinghui Wu[3] Jingbo Xu[1,2]
[1]University of Edinburgh [2]RCBD and SKLSDE Lab, Beihang University [3]Washington State University
{wenfei@inf, jingbo.xu@}.ed.ac.uk, yinghui@eecs.wsu.edu

ABSTRACT

We propose a class of functional dependencies for graphs, referred to as GFDs. GFDs capture both attribute-value dependencies and topological structures of entities, and subsume conditional functional dependencies (CFDs) as a special case. We show that the satisfiability and implication problems for GFDs are coNP-complete and NP-complete, respectively, no worse than their CFD counterparts. We also show that the validation problem for GFDs is coNP-complete. Despite the intractability, we develop parallel scalable algorithms for catching violations of GFDs in large-scale graphs. Using real-life and synthetic data, we experimentally verify that GFDs provide an effective approach to detecting inconsistencies in knowledge and social graphs.

Keywords

Functional dependencies; graphs; implication; satisfiability; validation

1. INTRODUCTION

Data dependencies have been well studied for relational data. In particular, our familiar functional dependencies (FDs) are found in every database textbook, and have been extended to XML [9]. Their revisions, such as conditional functional dependencies (CFDs) [16], have proven effective in capturing semantic inconsistencies in relations [15].

The need for FDs is also evident in graphs, a common source of data. Unlike relational databases, real-life graphs typically do not come with a schema. FDs specify a fundamental part of the semantics of the data, and are hence particularly important to graphs. Moreover, (1) FDs help us detect inconsistencies in knowledge bases [45], which need to be identified as violations of dependencies [15]. (2) For social networks, FDs help us catch spams and manage blogs [11].

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

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915232

(1) *Knowledge bases*, where inconsistencies are common [45]:

- Flight A123 has two entries with the same departure time 14:50 and arrival time 22:35, but one is from Paris to NYC, while the other from Paris to Singapore [40].
- Both Canberra and Melbourne are labeled as the capital of Australia [13].
- It is marked that all birds can fly, and that penguins are birds [26], despite their evolved wing structures.

We will see that all these inconsistencies can be easily captured by FDs defined on entities with a graph structure.

(2) *Social graphs*. When a blog Z with photo Y is posed, a social network company defines a status X with attachment Y. It is required that the annotation X.text of X must match the description Y.desc of Y. That is,

- Blog: if Z has_status X, Z has_photo Y, and if X has_attachment Y, then $X.\text{text} = Y.\text{desc}$.

This is essentially an FD on graph-structured data.

Functional dependencies are also useful in catching spams.

- Fake account [11]: If account x' is confirmed fake, both accounts x and x' like blogs $P_1, \ldots, P_k$, x posts blog y, x' posts y', and if y and y' have a particular keyword c, then x is also identified as a fake account.

This rule to identify fake accounts is an FD on graphs. □

No matter how important, however, the study of FDs for graphs is still in its infancy, from formulation to classical problems to applications. It is more challenging to define FDs for graphs than for relations, since real-life graphs are semi-structured and typically do not have a schema. Moreover, for entities represented by vertices in a graph, FDs have to specify not only regularity between attribute values of the entities, but also the topological structures of the entities.

Contributions. We study functional dependencies for graphs, from their fundamental problems to applications.

(1) We propose a class of functional dependencies for graphs, referred to as GFDs (Section 3). As opposed to relational FDs, a GFD specifies two constraints: (a) a topological constraint in terms of a graph pattern (Section 2), to identify entities on which the dependency is defined, and (b) an extension of CFDs to specify the dependencies of the attribute values of the entities. We show that GFDs subsume FDs and CFDs as special cases, and capture inconsistencies between attributes of the same entity and across different entities.

(2) We settle two classical problems for GFDs (Section 4). For a set Σ of GFDs, we study (a) its satisfiability, to decide

whether there exists a non-empty graph that satisfies all the GFDs in Σ, and (b) its implication, to decide whether a GFD is entailed by Σ. We show that the satisfiability and implication problems for GFDs are coNP-complete and NP-complete, respectively. The results tell us that reasoning about GFDs is no harder than their relational counterparts such as CFDs, which are also intractable [16].

(3) As one of applications of GFDs, we study the validation problem, to detect errors in graphs by using GFDs as data quality rules (Section 5). We show that it is coNP-complete to decide whether a graph contains no violation of a set of GFDs. Despite the intractability, we develop algorithms that are *parallel scalable*, *i.e.,* they guarantee to take less time when more processors are used. They are 2-approximation algorithms for a bi-criteria optimization problem, to balance workload and minimize communication costs (Section 6). These make it feasible to detect errors in large-scale graphs.

(4) Using real-life and synthetic graphs, we experimentally verify the effectiveness and efficiency of our GFD techniques (Section 7). We find the following. (a) Inconsistency detection with GFDs is feasible in real-life graphs. It takes 156 (resp. 326) seconds over replicated (resp. partitioned) YAGO [44] with 20 processors, for a set of 50 GFDs. (b) Our algorithms are parallel scalable: they are on average 2.4 and 3.7 times faster on fragmented and replicated real-life graphs, respectively, when processors increase from 4 to 20. (c) Our optimization techniques are effective: they improve performance by up to 1.9 times. (d) GFDs catch a variety of inconsistencies in real-life graphs, validating the need for combining topological constraints and value dependencies.

We contend that GFDs are a natural extension of traditional FDs, by incorporating graph topological structures. GFDs provide us with primitive dependencies for graphs, to specify fundamental semantics and to detect inconsistencies. This work also provides the first complexity bounds for reasoning about GFDs. Moreover, we develop the first parallel scalable algorithms to make practical use of GFDs.

Related work. We categorize related work as follows.

FDs *on graphs.* Extensions of FDs and CFDs have been studied for RDF [8, 10, 12, 23, 24, 49]. The definitions of FDs in [8, 12, 24] are based on RDF triple embedding and the coincidence of variable valuations. FDs are extended [49] to specify value dependencies on clustered values via, *e.g.,* path patterns; similarly for extensions of CFDs [23]. A schema matching framework is proposed in [10], for transformations between RDF and relations. It defines FDs as trees in which each node denotes an attribute in a corresponding relation.

Our work differs from the prior work in the following. (1) We define GFDs with graph patterns to express topological constraints of (property) graphs, beyond RDF. (2) GFDs capture inconsistencies in graph-structured entities identified by patterns. In contrast, the FDs of [8, 12, 24] are value-based regardless of what entities carry the values, and the reasoning techniques of [24] are based on relational encoding of RDF data. Moreover, these FDs cannot express equality with constants (semantic value binding) as in CFDs, *e.g.,* x.city = "Edi", while GFDs subsume CFDs. The FDs of [10] are defined as trees and assume a relational schema. They do not support general topological constraints; similarly for [23,49]. (3) We provide complexity bounds for GFD analyses and parallel scalable algorithms for error detection in graphs, which were not studied by the prior work.

Closer to this work is [14] on keys for graphs [14], which differ from GFDs in the following. (1) Keys are defined simply as a graph pattern $Q[x]$, with a designated variable x denoting an entity. In contrast, GFDs have the form $(Q[\bar{x}], X \rightarrow Y)$, where $\bar{x}$ is a list of variables, and X and Y are conjunctions of equality atoms with constants and variables in $\bar{x}$. GFDs cannot be expressed as keys, just like that relational FDs are not expressible as keys. Moreover, keys of [14] are recursively defined to identify entities, while GFDs are an extension of conventional FDs and are not recursively defined. (2) Keys are defined on RDF triples (s, p, o), while GFDs are defined on property graphs, *e.g.,* social networks. (3) Keys are interpreted in terms of three isomorphic mappings: two from subgraphs to Q, and one between the two subgraphs. In contrast, GFDs needs a single isomorphic mapping from a subgraph to Q. In light of the different semantics, algorithms for GFDs and keys are radically different. (4) We study the satisfiability and implication for GFDs; these classical problems were not studied for keys [14].

Inconsistency detection has been studied for relations (see [15] for a survey), and recently for knowledge bases (linked data) [23,32,35,37,42,45]. The methods for knowledge bases employ either rules [23, 32, 35, 42, 45], or probabilistic inferences [37]. (1) Datalog rules are used [42] to extract entities and detect inconsistent "facts". SOFIE [45] maintains the consistency of extracted facts by using rules expressed as first-order logic (FO) formulas along with textual patterns, existing ontology and semantic constraints. Pellet [35] checks inconsistencies by using inference rules in description logic (*e.g.,* OWL-DL). Dependency rules are used to detect inconsistencies in attribute values in semantic Web [32] and RDF [23]. BigDansing [28] supports user-defined rules for repairing relational data. To clean graph-structured entities, it needs to represent graphs as tables and encode isomorphic functions beyond relational query languages. (2) The inference method of [37] uses Markov logic to combine FO and probabilistic graphical models, and detects errors by learning and computing joint probability over structures.

Our work differs from the prior work as follows. (1) GFDs are among the first data-quality rules on (property) graphs, not limited to RDF, by supporting topological constraints with graph patterns. (2) GFDs aim to strike a balance between complexity and expressivity. Reasoning about GFDs is much cheaper than analyzing FO formulas. (3) We provide the complexity and characterizations for satisfiability and implication of GFDs; these are among the first results for reasoning about graph dependencies in general, and about data quality rules for graphs in particular. (4) We develop parallel scalable algorithms for error detection and new strategies for workload assignment, instead of expensive large-scale inference and logic programming. These make error detection feasible in large graphs with provable performance guarantees, which are not offered by the prior work.

Parallel algorithms related to GFD validation algorithms are (1) algorithms for detecting errors in distributed data [17, 18], and (2) algorithms for subgraph enumeration, subgraph isomorphism and SPARQL [5,20,22,25,30,31,39,41,46].

(1) Algorithms of [17, 18] (incrementally) detect errors in (horizontally or vertically) partitioned relations based on CFDs. The methods work on relations, but do not help GFDs

that require subgraph isomorphism computation. Indeed, our algorithms are radically different from those of [17, 18].

(2) Closer to this work are parallel algorithms for subgraph enumeration [5, 30, 36, 41]. (a) MapReduce algorithms are proposed via conjunctive multi-way join operations [5] and decomposed edge joins [36]. The strategy is effective for triangle counting [47]. (b) To reduce excessive partial answers for general patterns, a MapReduce solution in [30] decomposes a pattern into twin twigs (single edge or two incident edges), and adopts a left-deep-join strategy to join multiple edges as stars. To cope with skewed nodes, the neighborhoods of high-degree nodes are partitioned, replicated and distributed. Decomposition strategies are used to reduce MapReduce rounds and I/O cost. (c) A BSP framework is developed in [41] via vertex-centric programming. It adopts an online greedy strategy to assign partial subgraphs to workers that incur minimum overall workload, and optimization strategies to reduce subgraph instances.

(3) A number of parallel algorithms are developed for subgraph isomorphism [39, 46] and SPARQL queries [20, 22, 25, 31]. Twig decomposition is used to prune the intermediate results and reduce the latency in Trinity memory cloud [46]. The in-memory algorithm of [39] parallelizes a backtracking procedure by (a) evenly distributing partial answers among threads for local expansion, and (b) copying the partial answers to a global storage for balanced distribution in the next round. Hash-based partitioning, query decomposition and load balancing strategies are introduced for parallel SPARQL on RDF [20, 25]. Query decomposition and plan generation techniques are studied in [22], which avoid communication cost by replicating graphs. Optimization techniques for multi-pattern matching are provided in [31], by extracting common sub-patterns. Many of these techniques leverage RDF schema and SPARQL query semantics, which are not available for GFDs and general property graphs.

This work differs from the prior work in the following. (a) GFD validation in distributed graphs is a bi-criteria optimization problem, to balance workload and minimize communication cost, with combined complexity from subgraph enumeration of *disconnected* patterns and dependency checking in fragmented graphs. It is more challenging than graph queries studied in the prior work. (b) We introduce a workload assignment strategy for the intractable optimization problem, with approximation bounds, instead of treating workload balancing and communication cost minimization separately [30, 41]. (c) We warrant parallel scalability, which is not guaranteed by the prior algorithms.

On the other hand, this work can benefit from prior techniques for fast parallel subgraph matching and listing, *e.g.,* query decomposition strategies [22, 30, 46] and multi-thread in-memory algorithm [39], for local error detection at each worker. We have adopted the optimization techniques of [31], and will incorporate others into GFD tools.

(4) There has also been work on characterizing the effectiveness of parallel algorithms, in terms of communication costs of MapReduce algorithms [6], constraints on MapReduce computation/communication cost (MRC [27], MMC [48] and SGC [38]), and the polynomial fringe property of recursive programs [4]. We adopt the notion of parallel scalability [29], which measures speedup by parallelization over multiple processors, in terms of both computation and communication costs. It is for generic parallel algorithms not

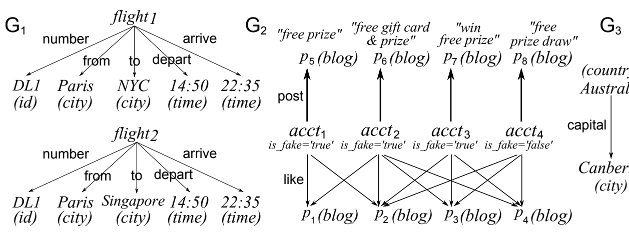


Figure 1: Graphs

limited to MapReduce. A parallel scalable algorithm guarantees to scale with large graphs by adding processors. However, parallel scalability is beyond reach for certain graph computations [19]. We show that GFD validation is parallel scalable, by providing such algorithms.

Static analyses. Over relations, the satisfiability and implication problems are known to be in $O(1)$ and linear time for FDs, NP-complete and coNP-complete for CFDs, $O(1)$ time and PSPACE-complete for inclusion dependencies (INDs), respectively. The validation problem is in PTIME for FDs, CFDs and INDs (cf. [3, 15]). We show that for GFDs on graphs, validation, satisfiability and implication for GFDs are coNP-complete, coNP-complete and NP-complete, respectively. As will be seen in Section 4, the complexity of GFDs comes from the interactions between graph patterns (subgraph isomorphism); it is not inherited from CFDs.

2. PRELIMINARIES

We start with a review of basic notations.

Graphs. We consider directed *graphs* $G = (V, E, L, F_A)$ with labeled nodes and edges, and attributes on its nodes. Here (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; (3) each node v in V (resp. edge e in E) carries label $L(v)$ (resp. $L(e)$), and (4) for each node v, $F_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where a_i is a constant, A_i is an *attribute* of v written as $v.A_i = a_i$, carrying the content of v such as properties, keywords, blogs and rating, as found in social networks, knowledge bases and property graphs.

Example 2: Three graphs are depicted in Fig. 2: (a) G_1 is a fragment of a knowledge graph, where each flight entity (*e.g.,* flight$_1$) has id (with value val = DL1), departure city (Paris), destination (NYC), and departure and arrival time; each node has attribute val (not shown) for its value; (b) G_2 records fake accounts; each account has an attribute is_fake that is "true" if the account is fake, and "false" otherwise; an account may post blogs that contain keywords (*e.g.,* blog p_5 has attribute keyword = "free prize"), and may like other blogs; and (c) G_3 depicts a country entity and its capital, carrying attribute val (not shown) for their values. □

We review 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 $e \in E'$, $L'(e) = L(e)$.
- We say that G' is *a subgraph induced by* a set V' of nodes if $G' \subseteq G$ and E' consists of all the edges in G whose endpoints are both in V'.

Graph patterns. A *graph pattern* is defined as a directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (resp. E_Q) is a

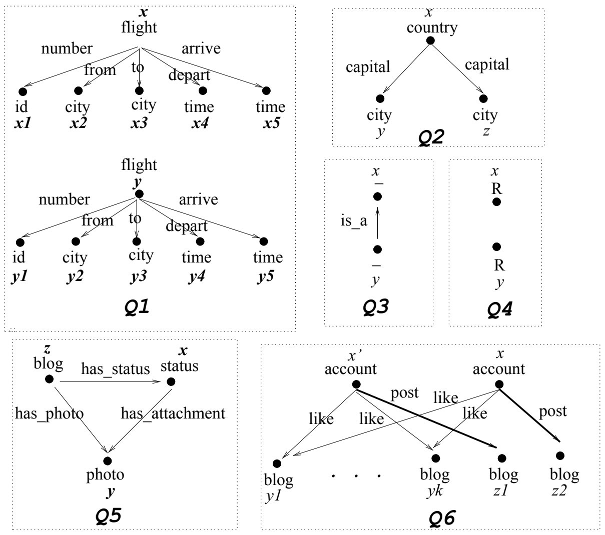


Figure 2: Graph patterns

set of pattern nodes (resp. edges), (2) L_Q is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) to each pattern node $u \in V_Q$ (resp. edge $e \in E_Q$), (3) $\bar{x}$ is a list of variables such that its arity $\|\bar{x}\|$ is equal to the number $|V_Q|$ of nodes, 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.

In particular, we allow wildcard '_' as a special label.

Example 3: Figure 2 depicts six graph patterns Q_1–Q_6: (1) Q_1 specifies two flight entities, where μ maps x to a flight, x_1–x_4 to its id, departure city, destination, departure time and arrival time, respectively; similarly for y and y_1–y_5; (2) Q_2 depicts a country entity with two distinct capitals; (3) Q_3 shows a generic is_a relationship, in which two nodes are labeled wildcard '_'; (4) Q_4 depicts two tuples of relation R represented as vertices in a graph, labeled with R; (5) Q_5 shows a blog entity z including photo y, and z is described by a status x; and (6) Q_6 specifies relationships between accounts x, x' and blogs $y_1, \ldots, y_k$ and z_1, z_2, where x and x' both like k blogs, x' posts a blog z_1 and x posts z_2. $\square$

Graph pattern matching. We adopt the conventional semantics of matching via subgraph isomorphism. A *match* of pattern Q in graph G is a subgraph $G' = (V', E', L', F'_A)$ of G that is isomorphic to Q, *i.e.,* there exists a *bijective function* h from V_Q to V' such that (1) for each node $u \in V_Q$, $L_Q(u) = L'(h(u))$; and (2) $e = (u, u')$ is an edge in Q if and only if $e' = (h(u), h(u'))$ is an edge in G' and $L_Q(e) = L'(e')$. In particular, $L_Q(u) = L'(h(u))$ always holds if $L_Q(u)$ is '_', *i.e.,* wildcard matches any label to indicate generic entities, *e.g.,* is_a in Q_3 of Example 3; similarly for edge labels.

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

Example 4: A match of Q_1 of Example 3 in G_1 of Fig. 2 is h_1: $x \mapsto \text{flight}_1$, $y \mapsto \text{flight}_2$, $x_3 \mapsto$ NYC, $y_3 \mapsto$ Singapore, and similarly for the other variables in Q_1.

When $k = 2$, a match of Q_6 in G_2 is h_2: ($x' \mapsto \text{acct}_3, x \mapsto \text{acct}_4$, $y_1 \mapsto \text{p}_3$, $y_2 \mapsto \text{p}_4$, $z_1 \mapsto \text{p}_7, z_2 \mapsto \text{p}_8$). $\square$

The notations of this paper are summarized in Table 1.

symbols	notations
G	graph (V, E, L, F_A)
$Q[\bar{x}]$	graph pattern $(V_Q,\ E_Q,\ L_Q,\ \mu)$
φ, Σ	GFD $\varphi = (Q[\bar{x}], X \to Y)$, Σ is a set of GFDs
$h(\bar{x}) \models X \to Y$	a match $h(\bar{x})$ of Q satisfies $X \to Y$
Σ_Q	a set of GFDs of Σ embedded in pattern Q
$\text{Vio}(\Sigma, G)$	all the violations of GFDs Σ in graph G
$t(\|\Sigma\|, \|G\|)$	sequential time for computing $\text{Vio}(\Sigma, G)$
$T(\|\Sigma\|, \|G\|, n)$	parallel time for $\text{Vio}(\Sigma, G)$, using n processors
$W(\Sigma, G)$	workload for computing $\text{Vio}(\Sigma, G)$
$\text{PV}(\varphi)$	a pivot vector $(\bar{z}, \bar{c}_Q)$ of GFD φ
$w = \langle \bar{v}_z, G_{\bar{z}} \rangle$	work unit ($\bar{v}_z$: candidate; $G_{\bar{z}}$: neighbors of $\bar{v}_z$)

Table 1: Notations

3. GFDS: SYNTAX AND SEMANTICS

We now define *functional dependencies for graphs* (GFDs).

GFDs. A GFD φ is a pair $(Q[\bar{x}], X \to Y)$, where

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

Here a *literal* of $\bar{x}$ has the form of either $x.A = c$ or $x.A = y.B$, where $x, y \in \bar{x}$, A and B denote attributes (not specified in Q), and c is a constant. We refer to $x.A = c$ as a *constant literal*, and $x.A = y.B$ as a *variable literal*.

Intuitively, GFD φ specifies two constraints:

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

Recall that the "scope" of a relational FD $R(X \to Y)$ is specified by a relation schema R: the FD is applied only to instances of R. Unlike relational databases, graphs do not have a schema. Here Q specifies the scope of the GFD, such that the dependency $X \to Y$ is imposed only on the attributes of the vertices in each subgraph identified by Q. Constant literals $x.A = c$ enforce bindings of semantically related constants, along the same lines as CFDs [16].

Example 5: To catch the inconsistencies described in Example 1, we define GFDs with patterns Q_1–Q_6 of Fig. 2.

(1) *Flight*: GFD $\varphi_1 = (Q_1[x, x_1\text{-}x_5, y, y_1\text{-}y_5],\ X_1 \to Y_1)$, where X_1 is $x_1.\text{val} = y_1.\text{val}$, and Y_1 consists of $x_2.\text{val} = y_2.\text{val}$ and $x_3.\text{val} = y_3.\text{val}$. Here val is an attribute for the content of a node. By Q_1, x_1, x_2 and x_3 denote the flight id, departing city and destination of a flight x, respectively; similarly for y_1, y_2 and y_3 of entity y. Hence GFD φ_1 states that for all flight entities x and y, if they share the same flight id, then they must have the same departing city and destination.

(2) *Capital*: GFD $\varphi_2 = (Q_2[x, y, z],\ \emptyset \to y.\text{val} = z.\text{val})$. It is to ensure that for all country entities x, if x has two capital entities y and z, then y and z share the same name.

(3) *Generic* is_a: GFD $\varphi_3 = (Q_3[x, y],\ \emptyset \to x.A = y.A)$. It enforces a general property of is_a relationship: if entity y is_a x, then for any property A of x (denoted by attribute A), $x.A = y.A$. Observe that x and y in Q_3 are labeled with wildcard '_', to match arbitrary entities. Along the same lines, GFDs can enforce inheritance relationship subclass.

In particular, if x is labeled with bird, y with penguin, and A is can_fly, then φ_3 catches the inconsistency described in Example 1: penguins cannot fly but are classified as bird.

(4) FDs *and* CFDs. Consider an FD $R(X \to Y)$ over a relation schema R [3]. When an instance of R is represented as a graph in which each tuple is denoted by a node labeled R, we write $\varphi_4 = (Q_4[x, y],\ X' \to Y')$. Here Q_4 consists of two vertices x and y denoting two tuples of R, X' consists of

$x.A = y.A$ for all $A \in X$, and Y' includes $x.B = y.B$ for all $B \in Y$. Note that φ_4 is defined with variable literals only.

Using constant literals, GFDs can express CFDs [16]. For instance, $R(\textsf{country} = 44, \textsf{zip} \rightarrow \textsf{street})$ is a CFD defined on relation R, stating that in the UK, zip code uniquely determines street [16]. It can be written as GFD $\varphi'_4 = (Q_4[x, y], X' \rightarrow Y')$, where X' consists of $x.\textsf{country} = 44$, $y.\textsf{country} = 44$, and $x.\textsf{zip} = y.\textsf{zip}$, and Y' is $x.\textsf{street} = y.\textsf{street}$.

As another example, CFD $R(\textsf{country} = 44, \textsf{area\_code} = 131 \rightarrow \textsf{city} = \text{Edi})$ states that in the UK, if the area code of a city is 131, then the city is Edi [16]. It can be expressed as a GFD $\varphi''_4 = (Q''_4[x], X'' \rightarrow Y'')$, where Q''_4 consists of a single node x labeled R, and X'' includes $x.\textsf{country} = 44$ and $x.\textsf{area\_code} = 131$, while Y'' is $x.\textsf{city} = \text{Edi}$.

(5) Blogs: $\varphi_5 = (Q_5[x, y, z], \emptyset \rightarrow x.\textsf{text} = y.\textsf{desc})$. It states that if entities x, y and z satisfy the topological constraint of Q_5 depicted in Fig. 3, then the annotation of status x of blog z must match the description of photo y included in z.

(6) Fake account: $\varphi_6 = (Q_6[x, x', y_1, \ldots, y_k, z_1, z_2], X_6 \rightarrow Y_6)$, where X_6 includes $x'.\textsf{is\_fake} = \textsf{true}$, $z_1.\textsf{keyword} = c$, $z_2.\textsf{keyword} = c$, and Y_6 is $x.\textsf{is\_fake} = \textsf{true}$; here c is a constant indicating a peculiar keyword. It states that for accounts x and x', if the conditions in X_6 are satisfied, including that x' is confirmed fake, then x is also a fake account. □

Semantics. To interpret GFDs, we use the following notations. Consider a GFD $\varphi = (Q[\bar{x}], X \rightarrow Y)$. Consider a match $h(\bar{x})$ of Q in a graph G, and a literal $x.A = c$. We say that $h(\bar{x})$ *satisfies* the literal if *there exists* attribute A at the node $v = h(x)$ *and* $v.A = c$; similarly for literal $x.A = y.B$. We denote by $h(\bar{x}) \models X$ if $h(\bar{x})$ satisfies *all* the literals in X; similarly for $h(\bar{x}) \models Y$. Here we write $h(\mu(x))$ as $h(x)$, where μ is the mapping in Q from $\bar{x}$ to nodes in Q.

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

Observe the following. (1) For a literal $x.A = c$ in X, node $h(x)$ does not necessarily have attribute A. If $h(x)$ has *no* attribute A, $h(\bar{x})$ trivially satisfies $X \rightarrow Y$. This allows us to accommodate the semi-structured nature of graphs. (2) In contrast, when $x.A = c$ is in Y and $h(\bar{x}) \models Y$, then $h(x)$ must have attribute A by the definition of satisfaction above; similarly for $x.A = y.B$. (3) When X is $\emptyset$, $h(\bar{x}) \models X$ for any match $h(\bar{x})$ of Q in G; similarly for $Y = \emptyset$.

Example 6: Consider GFDs φ_1, φ_2 and φ_6 of Example 5 and G_1, G_2, G_3 of Fig. 2. One can verify the following.

(a) $G_1 \not\models \varphi_1$. Indeed, the match h_1 given in Example 4 satisfies X_1 since $h_1(x_1).\textsf{val} = h_1(y_1).\textsf{val}$, but it does not satisfy Y_1 since $h_1(x_3).\textsf{val} \neq h_1(y_3).\textsf{val}$. Similarly, $G_2 \not\models \varphi_6$, as witnessed by match h_2 of Example 4. Note that there are other matches of Q_6 in G_2 that satisfy $X_6 \rightarrow Y_6$, *e.g.,* when we map $x' \mapsto \textsf{acct}_1$ and $x \mapsto \textsf{acct}_2$, However, $G_2 \models \varphi_6$ only if *all* matches of Q_6 in G_2 satisfy $X_6 \rightarrow Y_6$.

(b) $G_3 \models \varphi_2$ since there exists *no* match of Q_2 in G_3: the country in G_3 has a unique capital, and trivially satisfies φ_2.

Observe the following: (a) entities in the same match of Q may be far apart; *e.g.,* $\textsf{flight}_1$ and $\textsf{flight}_2$ are *disconnected* from each other; and (b) $X \rightarrow Y$ is imposed only on matches of Q (satisfying its topological constraint), *e.g.,* φ_2. □

We say that a graph G *satisfies* a set Σ of GFDs if for all $\varphi \in \Sigma$, $G \models \varphi$, *i.e.,* G satisfies every GFD in Σ.

Special cases. GFDs subsume the following special cases.

(1) As shown by φ_4, φ'_4 and φ''_4 in Example 5, relational FDs and CFDs are special cases of GFDs, when tuples in a relation are represented as nodes in a graph. In fact, GFDs are able to express equality-generating dependencies (EGDs) [3].

(2) A GFD $(Q[\bar{x}], X \rightarrow Y)$ is called a *constant* GFD if X and Y consist of constant literals of $\bar{x}$ only. It is called a *variable* GFD if X and Y consist of variable literals only. Intuitively, constant GFDs subsume constant CFDs [16], and variable GFDs are analogous to traditional FDs [3].

In Example 5, φ_1-φ_5 are variable GFDs, φ''_4 and φ_6 are constant GFDs, while φ'_4 is neither constant nor variable.

(3) GFDs can specify certain type information. For an entity x of type τ, GFD $(Q[x], \emptyset \rightarrow x.A = x.A)$ enforces that x must have an A attribute, where Q consists of a single vertex labeled τ and denoted by variable x. However, GFDs *cannot* enforce that x has a finite domain, *e.g.,* Boolean.

4. REASONING ABOUT GFDS

We next study the satisfiability and implication problems for GFDs. These are classical problems associated with any class of data dependencies. Our main conclusion is that these problems for GFDs are no harder than for CFDs.

4.1 The Satisfiability Problem for GFDs

A set Σ of GFDs is *satisfiable* if Σ has a *model*; that is, a graph G such that (a) $G \models \Sigma$, and (b) for each GFD $(Q[\bar{x}], X \rightarrow Y)$ in Σ, there exists a match of Q in G.

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

Intuitively, it is to check whether the GFDs are "dirty" themselves when used as data quality rules. A model G of Σ requires all patterns in the GFDs of Σ to find a match in G, to ensure that the GFDs do not conflict with each other.

Over relational data, a set Σ of CFDs may not be satisfiable [16]. The same happens to GFDs on graphs.

Example 7: Consider two GFDs defined with the same pattern Q_7 depicted in Fig. 3: $\varphi_7 = (Q_7[x], \emptyset \rightarrow x.A = c)$ and $\varphi'_7 = (Q_7[x], \emptyset \rightarrow x.A = d)$, where c and d are distinct constants. Then there exists no graph G that includes a τ entity v and satisfies both φ_7 and φ'_7. For if such a node v exists, then by φ_7, v has an attribute A with value c, while by φ'_7, $v.A$ must take a different value d, which is impossible.

As another example, consider GFDs $\varphi_8 = (Q_8[x, y, z], \emptyset \rightarrow x.A = c)$ and $\varphi_9 = (Q_9[x, y, z, w], \emptyset \rightarrow x.A = d)$ for distinct c and d, where Q_8 and Q_9 are shown in Fig. 3. One can verify that each of φ_8 and φ_9 has a model, when taken alone. However, they are not satisfiable when put together. Indeed, if they have a model G, then there must exist isomorphic mappings h and h' from Q_8 and Q_9 to G, respectively, such that $h(x) = h'(x) = v$ for some node v in G. Then again, v is required to have attribute A with distinct values. □

As shown in Example 7, GFDs defined with different graph patterns may interact with each other. Indeed, Q_8 and Q_9 are different, but φ_8 and φ_9 can be enforced on the same node, since Q_8 is isomorphic to a subgraph of Q_9. This tells us that the satisfiability analysis has to check subgraph

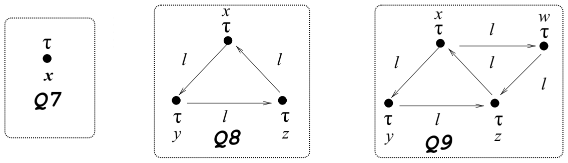


Figure 3: Graph patterns in GFDs

isomorphism among the patterns of the GFDs, which is NP-complete (cf. [34]). In light of this, we have the following.

Theorem 1: *The satisfiability problem is* coNP*-complete for* GFDs. □

One might think that the problem would become simpler if Σ consists of constant GFDs only (see Section 3), or when all patterns in Σ are acyclic directed graphs (DAGs). However, the complexity bound is rather robust.

Corollary 2: *The satisfiability problem is* coNP*-complete for constant* GFDs *that are defined with DAG patterns.* □

The complexity of GFDs is not inherited from CFDs. Indeed, the satisfiability analysis of CFDs is NP-hard only under a schema that enforces attributes to have a *finite domain* [16], *e.g.,* Boolean, *i.e.,* when CFDs and finite domains are put together. In contrast, graphs do not come with a schema; while GFDs subsume CFDs, they cannot specify finite domains. That is, the satisfiability problem for GFDs is already coNP-hard in the absence of a schema.

The upper bound proofs are nontrivial. It needs the following notations and a lemma.

(1) A pattern $Q' = (V'_Q,\, E'_Q,\, L'_Q,\, \mu')$ is *embeddable* in $Q = (V_Q,\, E_Q,\, L_Q,\, \mu)$ if there exists an isomorphic mapping f from (V'_Q, E'_Q) to a subgraph of (V_Q, E_Q), preserving node and edge labels. If Q' is embeddable in Q via f, then for any GFD $\varphi' = (Q'[\bar{x}'], X' \to Y')$ defined with Q', $(Q[\bar{x}],\, f(X') \to f(Y'))$ is an *embedded* GFD of φ' in Q, where $f(X')$ substitutes $f(x')$ for each x' in X'; similarly for $f(Y')$. Here again we use variable x and node $\mu(x)$ interchangeably.

(2) For a pattern Q and a set Σ of GFDs, a set Σ_Q of GFDs is said to be *embedded in* Q *and derived from* Σ if for each $\phi \in \Sigma_Q$, the pattern of ϕ is Q, and moreover, there exists $\varphi \in \Sigma$ such that ϕ is an embedded GFD of φ in Q.

(3) For a set Σ_Q of GFDs embedded in the same pattern Q, we define a set $\mathsf{enforced}(\Sigma_Q)$ of literals inductively as follows:

- if $(Q[\bar{x}], \emptyset \to Y)$ is in Σ_Q, then $Y \subseteq \mathsf{enforced}(\Sigma_Q)$, *i.e.,* all literals of Y are included in $\mathsf{enforced}(\Sigma_Q)$; and
- if $(Q[\bar{x}], X \to Y)$ is in Σ_Q and if all literals of X can be derived from $\mathsf{enforced}(\Sigma_Q)$ via the transitivity of equality atoms, then $Y \subseteq \mathsf{enforced}(\Sigma_Q)$.

As an example of transitivity, if $x.A = c$ and $y.B = c$ are in $\mathsf{enforced}(\Sigma_Q)$, then $X.A = y.B \in \mathsf{enforced}(\Sigma_Q)$. Intuitively, $\mathsf{enforced}(\Sigma_Q)$ is a set of equality atoms that have to be enforced on a graph G that satisfies Σ (and hence Σ_Q).

One can verify that given Σ_Q, $\mathsf{enforced}(\Sigma_Q)$ can be computed in polynomial time (PTIME) along the same lines as how closures for traditional FDs are computed (see, *e.g.,* [3]).

We say that Σ_Q is *conflicting* if there exist $(x.A, a)$ and $(x.A, b)$ in $\mathsf{enforced}(\Sigma_Q)$ such that $a \neq b$.

(4) A set Σ of GFDs is *conflicting* if *there exist* a pattern Q and a set Σ_Q of GFDs that are embedded in Q and derived from Σ, such that Σ_Q is *conflicting*.

Conflicting GFDs characterizes the satisfiability of GFDs.

Lemma 3: *A set* Σ *of* GFDs *is satisfiable if and only if* Σ *is not conflicting.* □

Proof of Theorem 1. Based on the lemma, we develop an algorithm that, given a set Σ of GFDs, returns "yes" if Σ is *not* satisfiable, *i.e.,* the complement of GFD satisfiability. (a) Guess (i) a set $\Sigma' \subseteq \Sigma$, (ii) a pattern Q such that Q carries labels that appear in Σ and $|Q|$ is at most the size of the largest pattern in Σ, and (iii) a mapping from the pattern of each GFD in Σ' to Q. (b) Check whether the mappings are isomorphic to subgraphs of Q. (c) If so, derive the set Σ_Q of GFDs embedded in Q from Σ' and the guessed mappings. (d) Check whether Σ_Q is conflicting; if so, return "yes". The algorithm is correct by Lemma 3. It is in NP as steps (b), (c) and (d) are in PTIME. Thus GFD satisfiability is in coNP.

The lower bound is verified by reduction from subgraph isomorphism to the complement of the satisfiability problem. The reduction uses constant GFDs defined with DAG patterns only, and hence proves Corollary 2 as well. □

Tractable cases. We next identify special cases when the satisfiability analysis can be carried out efficiently.

Corollary 4: *A set* Σ *of* GFDs *is always satisfiable if one of the following conditions is satisfied:*

- Σ *consists of variable* GFDs *only, or*
- Σ *includes no* GFDs *of the form* $(Q[\bar{x}], \emptyset \to Y)$.

It is in PTIME *to check whether* Σ *is satisfiable if* Σ *consists of* GFDs *defined with tree-structured patterns only,* i.e., *if for each* GFD $(Q[\bar{x}], X \to Y)$ *in* Σ*,* Q *is a tree.* □

4.2 The Implication Problem for GFDs

We say that a set Σ of GFDs *implies* another GFD φ, denoted by $\Sigma \models \varphi$, if for all graphs G such that $G \models \Sigma$, we have that $G \models \varphi$, *i.e.,* φ is a logical consequence of Σ.

We assume *w.l.o.g.* the following: (a) Σ is satisfiable, since otherwise it makes no sense to consider $\Sigma \models \varphi$; and (b) X is a satisfiable set of literals, where $\varphi = (Q[\bar{x}],\, X \to Y)$, since otherwise φ trivially holds. We will see that these do not increase the complexity of the implication problem.

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

In practice, the implication analysis helps us eliminate redundant data quality rules defined as GFDs, and hence, optimize our error detection process by minimizing rules.

Example 8: Consider a set Σ of two GFDs $(Q_8[x, y, z],\, x.A = y.A \to x.B = y.B)$ and $(Q_9[x, y, z, w],\, x.B = y.B \to z.C = w.C)$. Consider GFD $\varphi_{11} = (Q_9[x, y, z, w],\, x.A = y.A \to z.C = w.C)$, where patterns Q_8 and Q_9 are given in Fig. 3. One can verify that $\Sigma \models \varphi_{11}$. □

The implication analysis of GFDs is NP-complete. In contrast, the problem is coNP-complete for CFDs [16].

Theorem 5: *The implication problem for* GFDs *is* NP*-complete.* □

As suggested by Example 8, to decide whether $\Sigma \models \varphi$, we have to consider the interaction between their graph patterns even when φ and all GFDs in Σ are variable GFDs, and when none of them has the form $(Q[\bar{x}], \emptyset \to Y)$. Thus the implication analysis of GFDs is more intriguing than their satisfiability analysis, in contrast to Corollary 2.

Corollary 6: *The implication problem is* NP*-complete for constant* GFDs *alone, and for variable* CFDs *alone, even* *when all the* GFDs *are defined with* DAG *patterns and when none of them has the form* $(Q[\bar{x}], \emptyset \to Y)$. □

To prove these, consider a set Σ of GFDs and a GFD $\varphi = (Q[\bar{x}], X \to Y)$. We define the following notations.

(1) We assume that φ is in *the normal form*, *i.e.,* when Y consists of a single literal $x.A = y.B$ or $y.B = c$ that is not a tautology $x.A = x.A$. This does not lose generality. Indeed, if Y consists of multiple literals, then φ is equivalent to a set of GFDs $(Q[\bar{x}], X \to l)$, one for each literal $l \in Y$. If Y is $\emptyset$ or a tautology, then $\Sigma \models \varphi$ trivially holds.

(2) For a set Σ_Q of GFDs embedded in Q, we define a set $\mathsf{closure}(\Sigma_Q, X)$ of literals inductively as follows:

- $X \subseteq \mathsf{closure}(\Sigma_Q, X)$, *i.e.,* all literals of X are in it; and
- if $(Q[\bar{x}'], X' \to Y')$ is in Σ_Q and if all literals of X' can be derived from $\mathsf{closure}(\Sigma_Q, X)$ via the transitivity of equality atoms, then $Y' \subseteq \mathsf{closure}(\Sigma_Q, X)$.

Note that $\mathsf{closure}(\Sigma_Q, X)$ differs from $\mathsf{enforced}(\Sigma_Q)$ only in the base case: the former starts with a given set X of literals, while the latter uses X from GFDs with $\emptyset \to X$.

Along the same lines as closures of relational FDs [3], one can verify that $\mathsf{closure}(\Sigma_Q, X)$ can be computed in PTIME.

(3) Recall that Y is a literal by the normal form defined above. We say that Y is *deducible* from Σ and X if *there exists* a set Σ_Q of GFDs that are embedded in Q and derived from Σ, such that $Y \in \mathsf{closure}(\Sigma_Q, X)$.

We characterize the implication analysis as follows.

Lemma 7: *For* $\varphi = (Q[\bar{x}], X \to Y)$ *and a set* Σ *of* GFDs, $\Sigma \models \varphi$ *if and only if* Y *is deducible from* Σ *and* X. □

The proof of the lemma is an extension of its relational FD counterpart (see [3] for relational FDs).

Proof of Theorem 5. For the upper bound, we give an algorithm for deciding $\Sigma \models \varphi$ as follows. (a) Guess a set $\Sigma' \subseteq \Sigma$, and a mapping from the pattern of each GFD in Σ' to the pattern Q of φ. (b) Check whether the mappings are isomorphic to subgraphs of Q. (c) If so, derive the set Σ_Q of GFDs embedded in Q from Σ' and the guessed mappings. (d) Check whether $Y \in \mathsf{closure}(\Sigma_Q, X)$; if so, return "yes". The algorithm is in NP since steps (b), (c) and (d) are in PTIME. Its correctness follows from Lemma 7.

When the assumption about the satisfiability of Σ and X in φ is lifted, the algorithm can be extended with two initial steps: (i) check whether Σ is not satisfiable in NP; if so, return "invalid", and otherwise continue; (ii) check whether X is satisfiable, in PTIME; if so, continue; otherwise return "yes". The extended algorithm is still in NP. That is, the assumption does not increase the complexity bound.

The lower bound is verified by reduction from a variant of subgraph isomorphism, which is shown NP-complete. The reduction uses constant GFDs only or variable CFDs only, all defined with DAGs. Thus it also proves Corollary 2. □

<u>Tractable cases</u>. An efficient special case is as follows.

Corollary 8: *The implication problem is in* PTIME *for* GFDs *defined with tree-structured patterns.* □

5. INCONSISTENCY DETECTION

As an application of GFDs, we detect inconsistencies in graphs based on the validation analysis of GFDs. Our main conclusion is that while the validation problem for GFDs is intractable, it is feasible to efficiently detect errors in real-life graphs by means of parallel scalable algorithms.

5.1 GFD Validation and Error Detection

Given a GFD $\varphi = (Q[\bar{x}], X \to Y)$ 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 GFDs, we denote by $\mathsf{Vio}(\Sigma, G)$ the set of all violations of GFDs in G, *i.e.,* $h(\bar{x}) \in \mathsf{Vio}(\Sigma, G)$ if and only if there exists a GFD φ in Σ such that $h(\bar{x})$ is a violation of φ in G. That is, $\mathsf{Vio}(\Sigma, G)$ collects all entities of G that are inconsistent when the set Σ of GFDs is used as data quality rules.

The *error detection problem* is stated as follows:

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

Its decision problem, referred to as *the validation problem* for GFDs, is to decide whether $G \models \Sigma$, *i.e.,* whether $\mathsf{Vio}(\Sigma, G)$ is empty. The problem is nontrivial.

Proposition 9: *Validation of* GFDs *is* coNP*-complete.* □

Proof: We show that it is NP-hard to check, given G and Σ, whether $G \not\models \Sigma$, by reduction from subgraph isomorphism. For the upper bound, we give an algorithm that returns "yes" if $G \not\models \Sigma$: (a) guess a GFD $(Q[\bar{x}], X \to Y)$ from Σ and a mapping h from Q to a subgraph of G; (b) check whether h is isomorphic; (c) if so, check whether $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$; if so, return "yes". This is in NP. □

In contrast, validation is in PTIME for FDs and CFDs, and errors can be detected in relations by two SQL queries that can be automatically generated from FDs and CFDs [16]. That is, error detection is more challenging in graphs.

A sequential algorithm. We give an algorithm that, given a set Σ of GFDs and a graph G, computes $\mathsf{Vio}(\Sigma, G)$ with a single processor. It is denoted as detVio and works as follows. (1) It starts with $\mathsf{Vio}(\Sigma, G) = \emptyset$. (2) For each $(Q[\bar{x}], X \to Y)$ in Σ, it enumerates all matches $h(\bar{x})$ of Q in G, and checks whether $h(\bar{x}) \not\models X \to Y$; if so, it adds $h(\bar{x})$ to $\mathsf{Vio}(\Sigma, G)$.

The cost of detVio is dominated by enumerating matches $h(\bar{x})$ of $Q[\bar{x}]$ in Σ. It is exponential and prohibitive for big G.

5.2 Parallel Scalability

Is error detection feasible in large-scale graphs? Our answer is affirmative, by using parallel algorithms to compute $\mathsf{Vio}(\Sigma, G)$. To characterize the effectiveness of parallelization, we adopt a notion of *parallel scalability* [29]. Denote by

- $W(\Sigma, G)$ the *workload*, *i.e.,* the necessary amount of work needed to compute $\mathsf{Vio}(\Sigma, G)$ for any algorithm;
- $t(|\Sigma|, |G|)$ the running time of a "best" *sequential algorithm* to compute $\mathsf{Vio}(\Sigma, G)$, *i.e.,* among all such algorithms, it has the least worst-case complexity; and
- $T(|\Sigma|, |G|, n)$ the time taken by a parallel algorithm to compute $\mathsf{Vio}(\Sigma, G)$ by using n processors.

An error detection algorithm is *parallel scalable* if

$$T(|\Sigma|, |G|, n) = \frac{c * t(|\Sigma|, |G|)}{n} + (n|\Sigma||W(\Sigma, G)|)^l,$$

such that $\frac{c*t(|\Sigma|,|G|)}{n} \geq (n(|\Sigma||W(\Sigma, G)|)^l$ when $n \leq |G|$ as found in practice, where c and l are constants. It reduces running time when n gets larger. Intuitively, such an algorithm guarantees that for a (possibly large) graph G, the more processors are used, the less time it takes to compute $\mathsf{Vio}(\Sigma, G)$. Hence it makes error detection feasible.

Workload model. To characterize the cost of error detection, we first introduce a model to quantify its workload.

We start with notions. Consider a GFD $\varphi = (Q[\bar{x}], X \rightarrow Y)$, where $(Q_1, \ldots, Q_k)$ are (maximum) connected components of Q. Consider $\bar{z} = (z_1, \ldots, z_k)$, where for $i \in [1, k]$, z_i is a variable in $\bar{x}$ such that $\mu(z_i)$ is a node in Q_i, where μ is the mapping from variables to nodes in Q (see Section 2). We fix a $\bar{z}$, referred to as the *pivot* of φ, by picking z_i with the minimum radius in Q_i, where the *radius* is the longest shortest distance between $\mu(z_i)$ and any node in Q_i. We use $\mathsf{PV}(\varphi)$ to denote $((z_1, c_Q^1), \ldots, (z_k, c_Q^k))$, referred to as the *pivot vector* of φ, where c_Q^i is the radius of Q_i at $\mu(z_i)$.

Observe the following. (a) By the locality of subgraph isomorphism, for any graph G, match $h(\bar{x})$ of Q in G, and any node $v = h(x)$ for $x \in \bar{x}$, v is within c_Q^i hops of some $h(z_i)$. (b) Vector $\mathsf{PV}(\varphi)$ can be computed in $O(|Q|^2)$ time, where Q is much smaller than G in real life. (c) Pattern Q typically has 1 or 2 connected components, and 99% of the components have radius at most 2 [21]. Hence in $\mathsf{PV}(\varphi)$, the arity $\|\bar{z}\|$ and each radius c_Q^i are typically 1 or 2.

Example 9: For GFDs of Example 5, $\mathsf{PV}(\varphi_1)$, $\mathsf{PV}(\varphi_2)$, $\mathsf{PV}(\varphi_4)$ and $\mathsf{PV}(\varphi_6)$ are $((x,1),(y,1))$, $((x,1))$, $((x,0),(y,0))$ and $((x,3))$, respectively (see Fig. 2); in particular, we take account x as a pivot of Q_6; similarly for φ_3 for φ_5. □

A *work unit* w for checking φ in a graph G is characterized by an one-to-one mapping σ from $\bar{z}$ to nodes in G, where $\bar{z}$ is the pivot in $\mathsf{PV}(\varphi)$, such that for each $z_i \in \bar{z}$, $\sigma(z_i)$ and $\mu(z_i)$ share the same label, *i.e.,* $\sigma(z_i)$ is a *candidate* of $\mu(z_i)$. More specifically, $w = \langle \bar{v}_z, G_{\bar{z}} \rangle$, where (a) $\bar{v}_z = \sigma(\bar{z})$; and (b) $G_{\bar{z}}$ is the fragment of G that includes, for each $z_i \in \bar{z}$, the c_Q^i*-neighbor* of $\sigma(z_i)$, *i.e.,* the subgraph of G induced by all the nodes within c_Q^i hops of $\sigma(z_i)$. Intuitively, $G_{\bar{z}}$ is a data block in G that has to be checked to validate φ.

We refer to $\bar{v}_z$ as a *pivot candidate* for φ in G.

The *workload* $W(\varphi, G)$ for checking φ in G, denoted by $W(\varphi, G)$, is the set of work units $\langle \bar{v}_z, G_{\bar{z}} \rangle$ when $\bar{v}_z$ ranges over all pivot candidates of φ in G. The *workload* $W(\Sigma, G)$ of a set Σ of GFDs in G is $\bigcup_{\varphi \in \Sigma} W(\varphi, G)$.

Observe the following. (a) To validate GFD φ in a graph G, it suffices to enumerate matches $h(\bar{x})$ of Q in data block $G_{\bar{z}}$ of each work unit of φ, by the locality of subgraph isomorphism. That is, *we enumerate in small* $G_{\bar{z}}$ *instead of in big* G. (b) The sequential cost $t(|\Sigma|, |G|)$ is the sum of $|G_{\bar{z}}|^{|\Sigma|}$ for all $G_{\bar{z}}$'s that appear in $W(\Sigma, G)$. (c) The size $|W(\Sigma, G)|$ is at most $|G|^k$, where k is the maximum arity of $\bar{z}$ in all $\mathsf{PV}(\varphi)$ of $\varphi \in \Sigma$. As argued earlier, typically $k \leq 2$. Hence $|W(\Sigma, G)|$ is *exponentially smaller* than $t(|\Sigma|, |G|)$. (d) For a match $h(\bar{x})$, checking whether $h(\bar{x}) \models X \rightarrow Y$ takes $O((|X| + |Y|)\mathsf{log}(|X| + |Y|))$ time, and $|X| + |Y| \leq |\varphi|$. Since the size $|\varphi|$ of φ is much smaller than $|G|$, $W(\varphi, G)$ suffices to assess the amount of work for checking φ in G.

Challenges. Computing $\mathsf{Vio}(\Sigma, G)$ is a bi-criteria optimization problem. (a) *Workload balancing*, to evenly partition $W(\Sigma, G)$ over n processors; it is to avoid "skewed" partitions, *i.e.,* when a processor gets far more work units than others, and hence, to maximize parallelism. (b) *Minimizing data shipment*, to reduce communication cost, which is often a bottleneck [4]. When a graph G is fragmented and distributed across processors, to process a work unit $w = \langle \bar{v}_z, G_{\bar{z}} \rangle$, we need to ship data from one processor to another to assemble $G_{\bar{z}}$. The cost, denoted by $\mathsf{CC}(w)$, is measured by $c_s * |M|$, where c_s is a constant and M is the data shipped.

Parallel scalable error detection. We tackle these challenges in the following two settings, which are practical parallel paradigms as demonstrated by [22]. We show that parallel scalability is within reach in these settings.

Replicated G. Graph G is replicated at each processor [22]. We study *error detection with replicated* G (Section 6.1), to balance workload $W(\Sigma, G)$ over n processors such that the overall parallel time for computing $\mathsf{Vio}(\Sigma, G)$ is minimized.

Theorem 10: *There exists a parallel scalable algorithm that given a set* Σ *of* GFDs *and a graph* G *replicated at* n *processors, computes* $\mathsf{Vio}(\Sigma, G)$ *in* $O(\frac{t(|\Sigma|, |G|)}{n} + |W(\Sigma, G)|(n + \log |W(\Sigma, G)|))$ *parallel time.* □

Partitioned G. When G is partitioned across processors, data shipment in inevitable. We study *error detection with partitioned* G (Section 6.2), with *bi-criteria* objective to (a) minimize data shipment and (2) balance the workload.

Theorem 11: *There exists a parallel scalable algorithm that given a set* Σ *of* GFDs, *a partitioned graph* G *and* n *processors, computes* $\mathsf{Vio}(\Sigma, G)$ *in* $O(\frac{t(|\Sigma|, |G|)}{n} + n|W(\Sigma, G)|^2$ $\mathsf{log}|W(\Sigma, G)| + |\Sigma||W(\Sigma, G)|)$ *parallel time.* □

6. PARALLEL ALGORITHMS

We next develop parallel scalable algorithms for error detection in the settings given above, as proofs of Theorems 10 and 11 in Sections 6.1 and 6.2, respectively. Such algorithms make it feasible to detect errors in large-scale graphs. We should remark that there exist other criteria for measuring the effectiveness of parallel algorithms (see Section 1).

6.1 Parallel Algorithm for Replicated Graphs

We start with an algorithm in the setting when G is replicated at each processor. In this setting, the major challenge is to balance the workload for each processor. The idea is to partition workload $W(\Sigma, G)$ in parallel, and assign (approximately) equal amount of work units to n processors.

Algorithm. The algorithm is denoted as repVal and shown in Fig. 4. Working with a coordinator S_c and n processors $S_1, \ldots, S_n$, it takes the following steps. (1) It first estimates workload $W(\Sigma, G)$, and creates a balanced partition $W_i(\Sigma, G)$ of $W(\Sigma, G)$ for $i \in [1, n]$, by invoking a parallel procedure bPar (line 1). It then sends $W_i(\Sigma, G)$ to processor S_i (line 2). (2) Each processor S_i detects its set of local violations, denoted by $\mathsf{Vio}_i(\Sigma, G)$, by a procedure localVio in parallel (line 3), which only visits the data blocks specified in $W_i(\Sigma, G)$. (3) When all processors S_i return $\mathsf{Vio}_i(\Sigma, G)$, S_c computes $\mathsf{Vio}(\Sigma, G)$ by taking a union of all $\mathsf{Vio}_i(\Sigma, G)$ (lines 4-5). It then returns $\mathsf{Vio}(\Sigma, G)$ (line 6).

We next present procedures bPar and localVio.

Workload balancing. Procedure bPar balances workload in two phases: estimation and partition, in parallel.

Workload estimation. Procedure bPar first estimates workload $W(\Sigma, G)$ in parallel, following the three steps below.

(1) At coordinator S_c, for each GFD $\varphi \in \Sigma$, bPar constructs a pivot vector $\mathsf{PV}(\varphi) = (\bar{z}, \bar{c}_Q)$. It then balances the computation for workload estimation at n processors as follows.

```
Algorithm repVal
Input: A set $\Sigma$ of GFDs, coordinator $S_c$, $n$ processors $S_1, \ldots, S_n$,
       a graph $G$ replicated at each processor
Output: Violation set $\mathsf{Vio}(\Sigma, G)$.
1.  $\mathsf{bPar}(\Sigma, G)$;    /*balance workload in parallel*/
/*executed at coordinator $S_c$*/
2.  send $W_i(\Sigma, G)$ to processor $S_i$;
3.  invoke $\mathsf{localVio}(\Sigma, W_i(\Sigma, G))$ at each processor $S_i$ for $i \in [1, n]$;
4.  if every processor $S_i$ returns answer $\mathsf{Vio}_i(\Sigma, G)$ then
5.      $\mathsf{Vio}(\Sigma, G) := \bigcup_{i\in[1,n]} \mathsf{Vio}_i(\Sigma, G))$;
6.  return $\mathsf{Vio}(\Sigma, G)$;

Procedure $\mathsf{localVio}(\Sigma, W_i(\Sigma, G))$
/*executed at each processor $S_i$ in parallel*/
1.  set $\mathsf{Vio}_i(\Sigma, G) := \emptyset$;
2.  for each $w = \langle v_{\bar z}, |G_{\bar z}|\rangle \in W_i(\Sigma, G)$ for GFD $\varphi \in \Sigma$ do
3.      enumerate matches $h(\bar x)$ by accessing $G_{\bar z}$;
4.      for each $h(\bar x)$ such that $h(\bar x) \not\models X \to Y$ do
5.          $\mathsf{Vio}_i(\Sigma, G) := \mathsf{Vio}_i(\Sigma, G) \cup \{h(\bar x)\}$;
6.  return $\mathsf{Vio}_i(\Sigma, G)$;
```

Figure 4: Algorithm repVal

(a) For each variable z in the pivot $\bar z$, it extracts the frequency distribution of *candidates* $C(\mu(z))$, *i.e.,* those nodes in G that have the same label as $\mu(z)$. This can be supported by statistics of G locally stored at S_c.

(b) For each $\mathsf{PV}(\varphi) = ((z_1, c_Q^1), \ldots, (z_k, c_Q^k))$ and each z_i, it evenly partitions candidates $C(\mu(z_i))$ into m sets, for a predefined number m. More specifically, it derives an m-balanced partition $R_{\mu(z_i)} = \{r_1, \ldots, r_m\}$ of value ranges of a selected attribute of $C(\mu(z_i))$, such that the number of candidates in $C(\mu(z_i))$ whose attribute values fall in each range r_j is even. This is done by using *e.g.,* precomputed equi-depth histogram (*e.g.,* [33]). It then constructs a set M of messages of the form $\langle \mathsf{PV}(\varphi), \overline{r}_z\rangle$, where φ is a GFD, $\overline{r}_z = \langle r_{z_1}, \ldots r_{z_k}\rangle$, and each $r_{z_i} \in R_{\mu(z_i)}$ is a *range* of $C(\mu(z_i))$ for z_i. Removing duplicates, M contains at most m^k messages for φ, where $k \leq 2$ in practice (see Section 5).

(c) The set M is evenly distributed to n processors; each processor S_i receives a subset M_i of about $\frac{|M|}{n}$ messages.

Example 10: Consider GFD φ_1 of Example 5, where $\mathsf{PV}(\varphi_1) = ((x, 1), (y, 1))$ (*i.e.,* $k = 2$). Consider graph G including 9 flights flight_1–flight_9. For $n = 3 = m$, procedure bPar balances the estimation $W(\varphi_1, G)$ as follows.

(1) It determines a 3-range partition R_{flight} for flight entities as *e.g.,* $\{[\mathsf{flight}_1, \mathsf{flight}_3], [\mathsf{flight}_4, \mathsf{flight}_6], [\mathsf{flight}_7, \mathsf{flight}_9]\}$, for both $\mu(x)$ and $\mu(y)$, based on attribute $\mu(x).\mathsf{val}$ and $\mu(y).\mathsf{val}$.

(2) It yields a set M of 6 messages $\langle \mathsf{PV}(\varphi_1), (r_{\mathsf{flight}}, r'_{\mathsf{flight}})\rangle$ after removing duplicates (since the two connected components in Q_1 (Fig. 2) of φ_1 are isomorphic, $(\mathsf{PV}(\varphi_1), r_i, r_j)$ and $(\mathsf{PV}(\varphi_1), r_j, r_i)$ are duplicates for ranges r_i and r_j).

It then evenly distributes M to 3 processors, *e.g.,* S_1 receives $M_1 = \{\langle \mathsf{PV}(\varphi_1), ([\mathsf{flight}_1, \mathsf{flight}_3], [\mathsf{flight}_1, \mathsf{flight}_3])\rangle$, $\langle \mathsf{PV}(\varphi_1), ([\mathsf{flight}_1, \mathsf{flight}_3], [\mathsf{flight}_4, \mathsf{flight}_6])\rangle\}$. □

(2) Procedure bPar then identifies work units at each processor S_i, in parallel. For each message $\langle \mathsf{PV}(\varphi), \overline{r}_z\rangle$ in M_i, S_i finds (a) all pivot candidates $v_{\bar z}$ of $\bar z$ such that for each $z_i \in \bar z$, *its candidate* $v_{\bar z}[z_i]$ in $v_{\bar z}$ has attribute value in the range $r_{z_i} \in \overline{r}_z$; and (b) the c_Q^i-neighbors $G_{\bar z}$ for each $v_{\bar z}$.

Each processor S_i then sends a message M'_i to the coordinator S_c. Here M'_i is a set of $\langle v_{\bar z}, |G_{\bar z}|\rangle$, each encoding a pivot candidate and *the size* of the data block for a unit. Note that $|G_{\bar z}|$ is sent, not $G_{\bar z}$. Moreover, S_i keeps track of $G_{\bar z}$ to facilitate local error detection (to be seen shortly).

Example 11: For $\langle \mathsf{PV}(\varphi_1), ([\mathsf{flight}_1, \mathsf{flight}_3], [\mathsf{flight}_1, \mathsf{flight}_3])\rangle$, processor S_1 finds 3 candidates $\{\mathsf{flight}_1, \mathsf{flight}_2, \mathsf{flight}_3\}$ in the range $[\mathsf{flight}_1, \mathsf{flight}_3]$, and their 1-hop neighbors. These yield $v_{\bar z}[x]$ as $(\mathsf{flight}_i, \mathsf{flight}_j)$ ($i \in [1, 3]$, $j \in [1, 3]$, and $i < j$ to remove duplicates) and correspondingly, 3 work units encoded with $|G_{\bar z}|$, where $|G_{\bar z}|$ is the total size of the 1-hop neighbors of flight_i and flight_j in $v_{\bar z}[x]$. For example, a unit w_1 is $\langle(\mathsf{flight}_1, \mathsf{flight}_2), 22\rangle$, where $G_{\bar z}$ for w_1 is graph G_1 in Fig. 2, which has 22 nodes and edges in total. □

(3) Procedure bPar, at the coordinator S_c, collects a set of messages $\langle v_{\bar z}, |G_{\bar z}|\rangle$ from all the processors, denoted by $W(\Sigma, G)$. It encodes the set of work units to be partitioned.

Workload partition. This gives rise to a load balancing problem. An n-partition $\mathcal{W}$ of $W(\Sigma, G)$ is a set of n pair-wisely disjoint work unit sets $\{W_1(\Sigma, G), \ldots, W_n(\Sigma, G)\}$, such that $W(\Sigma, G) = \bigcup_{i\in[1,n]} W_i(\Sigma, G)$. It is *balanced* if the cost $t(|\Sigma|, W_i(\Sigma, G))$, estimated as the sum of $|G_{\bar z}|^{|\Sigma|}$ for all $G_{\bar z}$ in $W_i(\Sigma, G))$, is approximately equal. The *load balancing problem* is to find a balanced n-partition $\mathcal{W}$ for a given $W(\Sigma, G)$.

Refer to the largest cost incurred at a processor as the *makespan* of the parallel processing. The load balancing problem is "equivalent to" makespan minimization [7], by setting the capacity of each processor as $\frac{t(|\Sigma|, |G|)}{n}$, via PTIME reductions. The problem is intractable, but approximable.

Proposition 12: *(1) The load balancing problem is* NP*-complete. (2) There is a 2-approximation algorithm to find a balanced workload partition in $O(n|W(\Sigma, G)| + |W(\Sigma, G)| \log |W(\Sigma, G)|)$ parallel time for given Σ, n and $W(\Sigma, G)$.* □

Given $W(\Sigma, G)$, procedure bPar computes a balanced n-partition with a greedy strategy, following an approximation algorithm of [7] for makespan minimization. (1) It first associates a weight $|G(\bar z)|$ with each work unit $w = \langle v_{\bar z}, |G_{\bar z}|\rangle$. It then sorts all the work units, in descending order of the weights. With each processor it associates a load, initially 0. (2) It greedily picks a work unit w with the smallest weight and a processor S_i with the minimum load, assigns w to S_i and updates the load of S_i by adding the weight of w. (3) The process proceeds until all work units are distributed. This yields a 2-approximation algorithm, by approximation-factor preserving reduction to its counterpart of [7].

Example 12: Suppose that coordinator S_c receives 9 work units $\{w_1, \ldots, w_9\}$ in total, with estimated size $\{22, 22, 26, 26, 30, 30, 24, 28, 28\}$, respectively. The greedy assignment strategy of bPar generates a 3-partition of the work units as $\{\{w_1, w_3, w_9\}, \{w_2, w_4, w_5\}, \{w_6, w_7, w_8\}\}$, with balanced block sizes as 76, 78, 82, respectively. Then S_c assigns the 3 partitions to processors S_1, S_2, S_3, respectively. □

Local error detection. Upon receiving the assigned $W_i(\Sigma)$, procedure localVio computes the local violation set $\mathsf{Vio}_i(\Sigma, G)$ at each processor S_i in parallel. For each work unit $\langle v_{\bar z}, |G_{\bar z}|\rangle \in W_i(\Sigma, G)$ for GFD φ, it (a) enumerates matches $h(\bar x)$ of the pattern in φ such that $h(\bar x)$ includes $v_{\bar z}$, by only accessing $G_{\bar z}$, and (b) checks whether $h(\bar x) \models X \to Y$ of φ. It collects in $\mathsf{Vio}_i(\Sigma, G)$ all violations detected, and sends $\mathsf{Vio}_i(\Sigma, G)$ to coordinator S_c at the end of the process.

Example 13: Consider GFD $\varphi_1 = (Q_1[\bar x], X_1 \to Y_1)$ (Example 5) and work unit w_1 (Example 11) assigned to processor S_1. Procedure localVio inspects G_1 (Fig. 2) for w_1, and finds a match $h_1(\bar{x})$ of Q_1 in G_1, where h_1 is given in Example 4. As shown there, $h_1(\bar{x}) \not\models X_1 \rightarrow Y_1$. Thus localVio adds $h_1(\bar{x})$ to $\mathsf{Vio}_1(\Sigma, G)$. Similarly, S_1 processes w_3 and w_9 assigned to it, and finally returns $\mathsf{Vio}_1(\Sigma, G)$ to S_c. □

Proof of Theorem 10. By the locality of subgraph isomorphism, procedure bPar identifies all work units, and localVio computes all violations. From these the correctness of repVal follows. For the complexity, one can verify the following: (a) procedure bPar estimates $W(\Sigma, G)$ in $O(\frac{|W(\Sigma,G)|}{n})$ parallel time, by using a balanced partition; the partitioning takes $O(n|W(\Sigma, G)| + |W(\Sigma, G)| \log |W(\Sigma, G)|)$ time [7]; and (b) procedure localVio takes $O(\frac{t(|\Sigma|,|G|)}{n})$ parallel time, via a balanced workload partition. Thus repVal has the complexity stated in Theorem 10 and is parallel scalable. □

6.2 Algorithm for Fragmented Graphs

Graph G may have already been fragmented and distributed across n processor, especially when it is too costly to replicate G at each processor. In this setting, we have a *bi-criteria error* detection problem. Given a set Σ of GFDs and a fragmented graph G, it is to compute $\mathsf{Vio}(\Sigma, G)$ in parallel, such that (1) the communication cost is minimized, and (2) the workload for n processors is balanced.

Consider a fragmentation $(F_1, \ldots, F_n)$ of $G(V, E, L, F_A)$ such that (a) each $F_i(V_i, E_i, L, F_A)$ is a subgraph of G, (b) $\bigcup E_i = E$ and $\bigcup V_i = V$, and (c) F_i resides at processor S_i ($i \in [1, n]$). Assume *w.l.o.g.* that the sizes of F_i's are approximately equal. Moreover, F_i keeps track of (a) *in-nodes $F_i.I$*, *i.e.,* nodes in V_i to which there exists an edge from another fragment, and (b) *out-nodes $F_i.O$*, *i.e.,* nodes in another fragment to which there is an edge from a node in V_i. We refer to nodes in $F_i.I$ or $F_i.O$ as *border nodes*.

Algorithm. We provide an error detection algorithm for fragmented G, denoted as disVal. It differs from repVal in workload estimation and assignment, and in local error detection, to minimize communication and computation costs.

Algorithm disVal works with a coordinator S_c and n processors $S_1, \ldots, S_n$. (1) It first estimates and partitions workload $W(\Sigma, G)$ via a procedure disPar, such that the workload $W_i(\Sigma, G)$ at each S_i is balanced, with minimum communication cost. (2) Each processor S_i uses a procedure dlovalVio to detect local violation $\mathsf{Vio}_i(\Sigma, G)$, in parallel, with data exchange. (3) Finally, $\mathsf{Vio}(\Sigma, G) = \bigcup_{i \in [1,n]} \mathsf{Vio}(\Sigma_i, G)$.

We next present procedures disPar and dlovalVio.

Bi-criteria assignment. Procedure disPar extends its counterpart bPar by supporting (a) workload estimation with communication cost, and (b) bi-criteria assignment.

Workload estimation. Procedure disPar estimates $W(\Sigma, G)$ at each S_i in parallel. For each pivot vector $\mathsf{PV}(\varphi) = ((z_1, c_Q^1), \ldots, (z_k, c_Q^k))$ and each z_l in $\bar{z}$, it finds (a) local candidates $C(\mu(z_l))$ of $\mu(z_l)$ in F_i, (b) the c_Q^l-neighbors $G_{\bar{z}}[z_l]$ for each candidate of $C(\mu(z_l))$, and (c) border nodes $B_{\bar{z}}[z_l]$ from $G_{\bar{z}}[z_l]$ to some nodes in $G_{\bar{z}}[z_l]$. It encodes *partial work unit* w_φ as $\langle v_{\bar{z}}, \overline{|G_{\bar{z}}|}, \overline{B_{\bar{z}}} \rangle$, where (i) $v_{\bar{z}}$ is a pivot candidate of $\bar{z}$ in F_i; if $C(\mu(z_l))=\emptyset$, $v_{\bar{z}}[z_j]$ takes a placeholder $\perp$; (ii) $\overline{|G_{\bar{z}}|}$ is the list of $|G_{\bar{z}}[z_l]|$; and (iii) $\overline{B_{\bar{z}}}$ is the list of border nodes $B_{\bar{z}}[z_l]$, for all $z_l \in \bar{z}$, indicating "missing data". Each S_i then sends a message M_i to coordinator S_c, with all units, along with the sizes of c-neighbors of border nodes in $F_i.I$, where c ranges over the radius of patterns Q in Σ.

Upon receiving M_i's, disPar builds $W(\varphi, G)$, the set of complete work units at S_c. A work unit $\langle v_{\bar{z}}, |G_{\bar{z}}|, B_{\bar{z}} \rangle$ is added to $W(\varphi, G)$ if for each $z_l \in \bar{z}$, $v_{\bar{z}}[z_l]$ is a candidate $v_{\bar{z}}^i[z_l]$ from a unit w_φ^i of M_i such that $v_{\bar{z}}^i[z_l] \neq \perp$, $|G_{\bar{z}}|$ is the sum of $|G_{\bar{z}}^i[z_l]|$ (extracted from $\overline{|G_{\bar{z}}^i|}$), and $B_{\bar{z}}$ is the union of $B_{\bar{z}}^i[z_l]$ (extracted from $\overline{B_{\bar{z}}^i}$), for all $i \in [1, n]$ and $\varphi \in \Sigma$. That is, disPar assembles $v_{\bar{z}}^i[z_l]$ into work units. It also marks $|G_{\bar{z}}^i[z_l]|$ and $B_{\bar{z}}^i[z_l]$ with its source w_φ^i.

Workload assignment. The *bi-criteria assignment problem* is to find an n-partition of $W(\Sigma, G)$ into $W_i(\Sigma, G)$ for $i \in [1, n]$, such that (a) $W_i(\Sigma, G)$ is balanced, and (b) its communication cost CC_i is minimized, where CC_i denotes the amount of data that needs to be shipped to processor S_i if $W_i(\Sigma, G)$ is assigned to S_i. It should ensure that for each pivot candidate $v_{\bar{z}}$, there exists a unique unit $\langle v_{\bar{z}}, |G_{\bar{z}}|, B_{\bar{z}} \rangle$ in all of $W_i(\Sigma, G)$, *i.e.,* the candidate is checked only once.

Cost CC_i is estimated as follows. For each $\langle v_{\bar{z}}, |G_{\bar{z}}|, B_{\bar{z}} \rangle$ in $W_i(\Sigma, G)$ and each $z_l \in \bar{z}$, define $\mathsf{CC}_{v_{\bar{z}}}[z_l]$ to be the sum of (a) $|G_{\bar{z}}^j[z_l]|$ if $j \neq i$, *i.e.,* $G_{\bar{z}}^j[z_l]$ has to be fetched from fragment j; (b) $|G_{(c_Q^l, b)}|$ for each border node $v_b \in B_{\bar{z}}[z_l]$, which also demands data fetching. These are identified by using the sources w_φ^i recorded above. Let $\mathsf{CC}_{v_{\bar{z}}}$ be the sum of $\mathsf{CC}_{v_{\bar{z}}}[z_l]$ for all $z_l \in \bar{z}$. Then CC_i is the sum of all $\mathsf{CC}_{v_{\bar{z}}}$ for candidates $v_{\bar{z}}$ in $W_i(\Sigma, G)$. Care is taken so that each data block is counted only once for CC_i.

While bi-criteria assignment is more intriguing than load balancing, it is within reach in practice via approximation.

Proposition 13: *(1) The bi-criteria assignment problem is* NP*-complete. (2) There exists a 2-approximation algorithm to find a balanced workload assignment with minimized communication cost in $O(n|W(\Sigma, G)|^2 \log(|W(\Sigma, G)|))$ time.* □

Extending a strategy for makespan minimization [43], procedure disPar computes an n-partition of $W(\Sigma)$ (after unit grouping) into $W_i(\Sigma, G)$, sent to processor S_i for $i \in [1, n]$.

Local error detection. Upon receiving $W_i(\Sigma, G)$, procedure dlovalVio computes local violations $\mathsf{Vio}_i(\Sigma, F_i)$ at processor S_i, by selecting the following evaluation schemes.

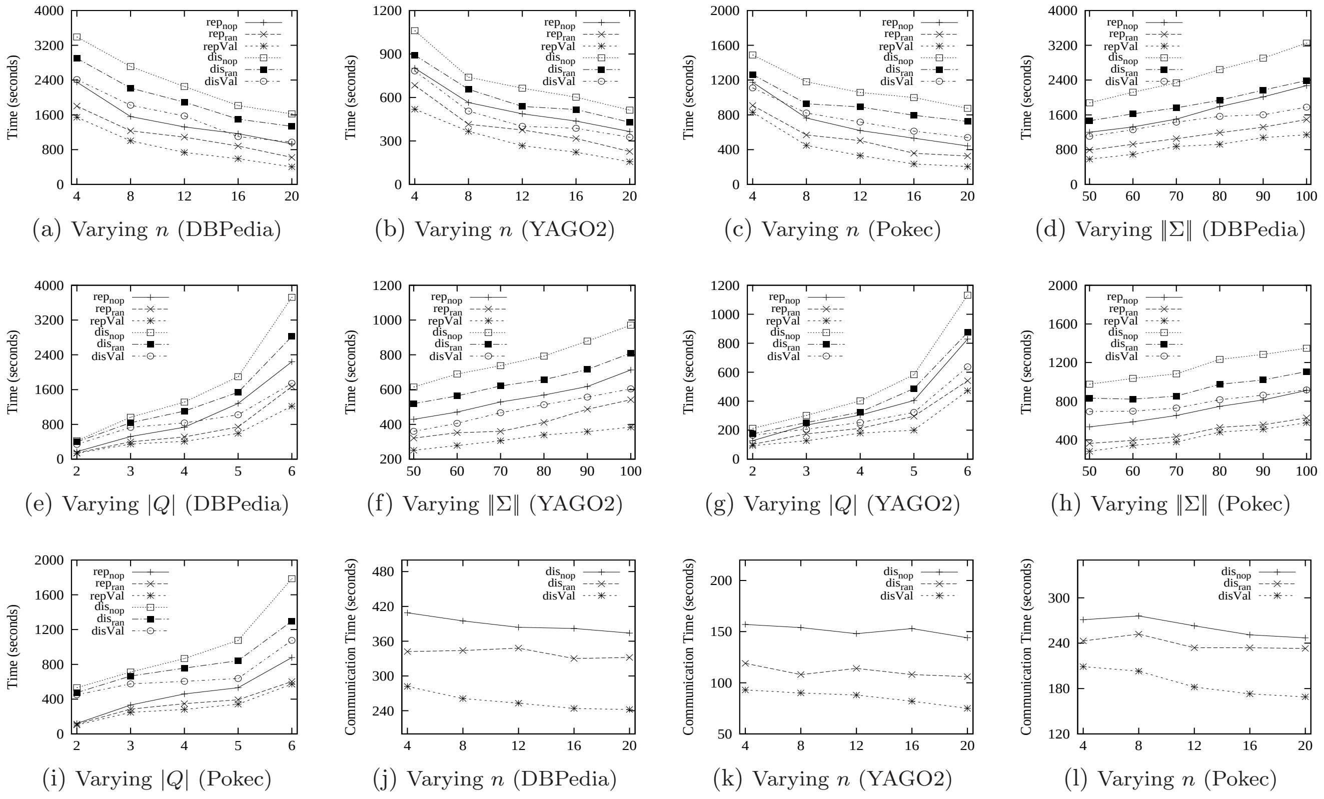
Prefetching. For a work unit $w = \langle v_{\bar{z}}, |G_{\bar{z}}|, B_{\bar{z}} \rangle$, it first fetches $G_{\bar{z}}$ and $G_{(c,b)}$ for $F_i.O$ nodes in $B_{\bar{z}}$ from other fragments. It ensures that each node (edge) is retrieved only once. After the data is in place, it detects errors locally as in localVio to compute $\mathsf{Vio}_i(\Sigma, F_i)$.

Partial detection. We can also ship partial matches instead of data blocks. The idea is to estimate the size of partial matches via *graph simulation* [19] from pattern $Q[\bar{x}]$ in a GFD φ to F_i. If the number of partial matches is not large, S_i exchanges such matches with other processors in a pipelined fashion, and updates $\mathsf{Vio}_i(\Sigma, F_i)$ as soon as a complete match can be formed from partial ones.

For a unit $w \in W_i(\Sigma, G)$ for GFD φ at S_i, procedure dlovalVio selects a strategy that incurs smaller (estimated) communication cost $\mathsf{CC}(w)$ (see Appendix for the estimation of $\mathsf{CC}(w)$). Intuitively, dlovalVio decides to process each unit either locally or at a remote processor, whichever incurs smaller data shipment.

Our algorithms also support optimization strategies for skewed graphs and workload reduction (see Appendix).

We verify Theorem 11 by showing that disVal is correct and has the desired complexity, similar to Theorem 10.



(a) Varying n (DBPedia) (b) Varying n (YAGO2) (c) Varying n (Pokec) (d) Varying $\|\Sigma\|$ (DBPedia)

(e) Varying $|Q|$ (DBPedia) (f) Varying $\|\Sigma\|$ (YAGO2) (g) Varying $|Q|$ (YAGO2) (h) Varying $\|\Sigma\|$ (Pokec)

(i) Varying $|Q|$ (Pokec) (j) Varying n (DBPedia) (k) Varying n (YAGO2) (l) Varying n (Pokec)

Figure 5: Performance evaluation

7. EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we experimentally evaluated (1) the parallel scalability, (2) workload partition, (3) communication costs, (4) scalability of our algorithms, and (5) the effectiveness of GFDs for error detection.

Experimental setting. We used three real-life graphs: (a) *DBpedia*, a knowledge graph [1] with 28 million entities of 200 types and 33.4 million edges of 160 types, (b) *YAGO2*, an extended knowledge base of YAGO [44] with 3.5 million nodes of 13 types and 7.35 million edges of 36 types, (c) *Pokec* [2], a social network with 1.63 million nodes of 269 different types, and 30.6 million edges of 11 types. We removed meaningless nodes and labels for a compact representation. We then inserted new edges by repeatedly dereferencing HTTP URIs over a set of sampled entities to further enlarge *DBpedia* (resp. *YAGO2*), to 12.3 million (resp. 3.2 million) entities and 32.7 million (resp. 7.1 million) edges.

We also developed a generator to produce synthetic graphs $G = (V, E, L, F_A)$ following the power-law degree distribution. It is controlled by the numbers of nodes $|V|$ (up to 50 million) and edges $|E|$ (up to 100 million), with L drawn from an alphabet $\mathcal{L}$ of 30 labels, and F_A assigning 5 attributes with values from an active domain of 1000 values.

GFDs *generator*. We generated sets Σ of GFDs $(Q[\bar{x}], X \rightarrow Y)$, controlled by (a) $\|\Sigma\|$, the number of GFDs, and (b) $|Q|$, the average size of graph patterns Q in Σ, with 1 or 2 connected components. For each real-life graph, (1) we first mined frequent features, including edges and paths of length up to 3. We selected top-5 most frequent features as "seeds", and combined them to form patterns Q of size $|Q|$. (2) For each Q, we constructed dependency $X \rightarrow Y$ with literals composed of the node attributes. We generated 100 GFDs on each real-life graph in this way. For synthetic graphs, we generated 50 GFDs with labels drawn from $\mathcal{L}$.

Algorithms. We implemented the following, all in Java: (1) sequential algorithm detVio (Section 5), (2) parallel algorithm repVal (Fig. 4), versus its two variants (a) rep_{ran}, which randomly assigns work units to processors, and (b) rep_{nop}, which does not support optimization strategies (multi-query processing [31] and workload reduction; see Appendix), and (3) parallel algorithm disVal (Section 6.2), versus its two variants dis_{ran} and dis_{nop} similar to their counterparts in (2).

We deployed the algorithms on Amazon EC2 c4.2xlarge instances, each is powered by an Intel Xeon processor with 2.6GHz. We used up to 20 instances. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Parallel scalability. We first evaluated parallel algorithms repVal and disVal, versus their variants. Fixing $|Q|$=5 and $\|\Sigma\|$=50, we varied the number n of processors from 4 to 20. We replicated and fragmented G for repVal and disVal, respectively. Figures 5(a), 5(b) and 5(c) report their performance on real-life *DBpedia*, *YAGO2* and *Pokec*, respectively. We find the following. (1) Both repVal and disVal substantially reduce parallel time when n increases: they are on average 3.7 and 2.4 times faster for n from 4 to 20, respectively. These validate Theorems 10 and 11. (2) Both repVal and disVal outperform their variants: repVal (resp. disVal) is on average 1.9 and 1.4 times (resp. 1.5 and 1.3 times) faster than rep_{nop} and rep_{ran} (resp. dis_{nop} and dis_{ran}), respectively. These verify the effectiveness of our optimization and load balancing techniques. (3) Algorithm repVal is

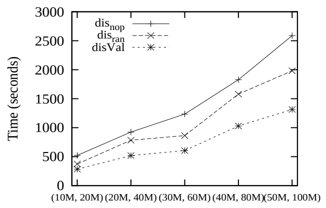


Figure 6: Scalability: Varying $|G|$ (synthetic)

faster than disVal, since it requires no data exchange by trading with replicated G. (4) Both repVal and disVal work well on large real-life graphs. For example, repVal (resp. disVal) takes 156 (resp. 326) seconds on *YAGO2* with 20 processors. In contrast, sequential algorithm detVio does not terminate on any of the three graphs within 6000 seconds. On average parallel graph replication (not shown) takes 21.3, 89 and 75 seconds for *YAGO2*, *DBpedia* and *Pokec*, respectively. The replication is performed once and is reused for all queries.

Exp-2: Workload complexity. We next evaluated the impact of the complexity of GFDs on workload estimation and partition, by varying $\|\Sigma\|$, the number of GFDs, and $|Q|$, the average pattern size. We fixed $n = 16$.

Varying $\|\Sigma\|$. Fixing $|Q| = 5$, we varied $\|\Sigma\|$ from 50 to 100. As shown in Figures 5(d), 5(f) and 5(h) on *DBpedia*, *YAGO2* and *Pokec*, respectively, (a) all the algorithms take longer time over larger Σ, as expected, and (b) repVal (resp. disVal) behaves better than rep$_{ran}$ and rep$_{nop}$ (resp. dis$_{ran}$ and dis$_{nop}$), by balancing workload and minimizing communication. However, detVio does not terminate within 120 minutes on any of the three graphs when $\|\Sigma\| \geq 80$.

Varying $|Q|$. Fixing $\|\Sigma\| = 50$, we varied $|Q|$ from 2 to 6. As shown in Figures 5(e), 5(g) and 5(i), all the algorithms take longer over larger $|Q|$, due to larger work units. However, repVal (resp. disVal) outperforms rep$_{nop}$ and rep$_{ran}$ (resp. dis$_{nop}$ and dis$_{ran}$) in all the cases, for the same reasons given above. Again, detVio does not terminate in 120 minutes when $|Q| \geq 6$ on all the three graphs.

Exp-3: Communication cost. In the same setting as Exp-1, we evaluated the total communication cost (measured as parallel data shipment time) of disVal, dis$_{ran}$ and dis$_{nop}$ over the three datasets, reported in Figures 5(j), 5(k) and 5(l), respectively. We omit repVal since it does not require data exchange. We find the following: (a) the total amount of data shipped (not shown) is far smaller than the size of the underlying graphs; this confirms our estimate of communication costs (Sections 5 and 6); (b) the communication cost takes from 12% to 24% of the overall error detection cost when n changes from 4 to 20; this is one of the reasons why adding processors does not always reduce parallel running time [19], since using more processors introduce more data exchange among different processors; and (c) although more data is shipped with larger n, the communication time is not very sensitive to n due to parallel shipment.

Exp-4: Synthetic G. We also evaluated the performance of algorithm disVal over large synthetic graphs of 50M nodes and 100M edges. We only tested the setting when G is partitioned, due to limited storage capacity for replicated G.

Fixing $n = 16$, we varied $|G|$ from (10M, 20M) to (50M, 100M). As shown in Fig. 6, (1) all the algorithms take longer time over larger $|G|$, as expected; (2) error detection is feasible in large graphs: disVal takes 21 minutes when $|G| =$ (50M, 100M); (3) disVal is on average 1.9 and 1.5 times faster than dis$_{ran}$ and dis$_{nop}$, respectively; this is consistent with the results on real-life graphs; and (4) sequential algorithm detVio does not run to completion when $|G| \geq$ (30M, 60M) within 120 minutes with one processor.

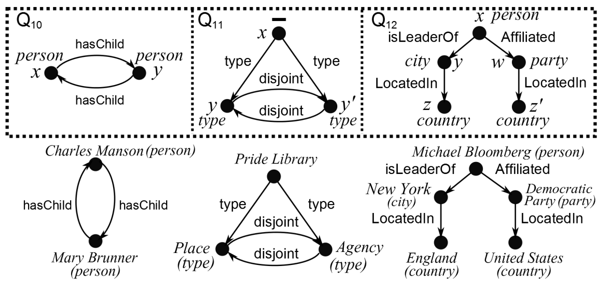


Figure 7: Real-life GFDs

Exp-5: Effectiveness. To demonstrate the effectiveness of GFDs in error detection, we show in Fig. 7 three real-life GFDs and error caught by them. Another set of experiments is reported in Appendix, comparing with other methods.

GFD *1* is $(Q_{10}[\bar{x}], \emptyset \rightarrow x.\mathsf{val} = c \ \wedge \ y.\mathsf{val} = d)$ for distinct c and d, (*i.e.,* $x.\mathsf{val} = c \ \wedge \ y.\mathsf{val} = d$ is false, stating that a person x cannot have y as both a child and a parent. It catches inconsistency in *YAGO2* shown in Fig. 7.

GFD *2* is $(Q_{11}[\bar{x}], \emptyset \rightarrow y.\mathsf{val} = y'.\mathsf{val})$, stating that an entity cannot have two disjoint types (with no common entities). It identifies an inconsistency at the "schema" level of *DBpedia* that contradicts a disjoint relationship.

GFD *3* is $(Q_{12}[\bar{x}], \emptyset \rightarrow z.\mathsf{val} = z'.\mathsf{val})$. It ensures that if a person is the mayor of a city in a country z, and is affiliated to a party of a country z', then z and z' must be the same country. It detects an error in *YAGO2* that associates different countries with New York city (NYC) and Democratic Party, witnessed by the mayor of NYC.

Summary. From the experimental results we find the following. (1) Error detection with GFDs is feasible in real-life graphs, *e.g.,* repVal (resp. disVal) takes 156 (resp. 326) seconds on *YAGO2* with 20 processors. (2) Better still, they are parallel scalable, with response time improved by 3.7 and 2.4 times, respectively, when the number of processors increase from 4 to 20. (3) Our optimization techniques improve the performance of rep$_{nop}$ and dis$_{nop}$ by 1.9 and 1.5 times, respectively; and workload balancing improves rep$_{ran}$ and dis$_{ran}$ by 1.4 and 1.3 times, respectively. (4) GFDs are capable of catching inconsistencies in real-world graphs.

8. CONCLUSION

The work is a first step towards a dependency theory for graphs. We have proposed GFDs, established complexity bounds for their classical problems, and provided parallel scalable algorithms for their application. Our experimental results have verified the effectiveness of GFD techniques.

One topic for future work is to develop effective algorithms for GFD discovery in real-life graphs. Another topic is to provide a sound and complete axiom system for GFDs, along the same lines as Armstrong's axioms for relational FDs [3]. A third topic is to re-investigate the satisfiability and implication problems for GFDs in the presence of types and other semantic constraints commonly found in knowledge bases.

Acknowledgments. Fan and Xu are supported in part by ERC 652976, 973 Program 2014CB340302 and 2012CB316200, NSFC 61133002 and 61421003, EPSRC EP/J015377/1 and EP/M025268/1, NSF III 1302212, Shenzhen Peacock Program 1105100030834361, Guangdong Innovative Research Team Program 2011D005, Shenzhen Science and Technology Fund JCYJ20150529164656096, Guangdong Applied R&D Program 2015B010131006, Beijing Advanced Innovation Centre for Big Data and Brain Computing, and a research grant from Huawei Technologies. Wu is supported by Google Faculty Research Award.

9. REFERENCES

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

[2] Pokec social network. *http://snap.stanford.edu/data/soc-pokec.html*.

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

[4] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.

[5] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, 2013.

[6] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

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

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

[9] M. Arenas and L. Libkin. A normal form for XML documents. In *PODS*, pages 85–96, 2002.

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

[11] Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *NSDI*, pages 197–210, 2012.

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

[13] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll. In *WWW*, 2004.

[14] W. Fan, Z. Fan, C. Tian, and L. X. Dong. Keys for graphs. In *PVLDB*, 2015.

[15] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.

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

[17] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *ICDE*, pages 64–75, 2010.

[18] W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. *TKDE*, 26(6), 2014.

[19] W. Fan, X. Wang, and Y. Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 2014.

[20] L. Galárraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient RDF processing. In *WWW*, 2014.

[21] 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.

[22] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 2015.

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

[24] J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional and constant constraints. *Ann. Math. Artif. Intell.*, pages 1–29, 2015.

[25] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[26] S. Huang, Q. Li, and P. Hitzler. Reasoning with inconsistencies in hybrid MKNF knowledge bases. *Logic Journal of IGPL*, 2012.

[27] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.

[28] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, 2015.

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

[30] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in MapReduce. *PVLDB*, 8(10):974–985, 2015.

[31] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012.

[32] L. Mendoza. A rule-based approach to address semantic accuracy problems on linked data. 2004.

[33] H. Mousavi and C. Zaniolo. Fast and accurate computation of equi-depth histograms over data streams. In *EDBT*, 2011.

[34] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[35] B. Parsia and E. Sirin. Pellet: An OWL DL reasoner. In *ISWC-Poster*, volume 18, 2004.

[36] T. Plantenga. Inexact subgraph isomorphism in MapReduce. *J. Parallel Distrib. Comput.*, 73(2):164–175, 2013.

[37] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI*, volume 7, pages 913–918, 2007.

[38] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in MapReduce. In *SIGMOD*, 2014.

[39] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. Pgx. iso: parallel and efficient

in-memory engine for subgraph isomorphism. In *GRADES*, pages 1–6, 2014.

[40] A. Rula and A. Zaveri. Methodology for assessment of linked data quality. In *LDQ@SEMANTiCS*, 2014.

[41] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, 2014.

[42] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[43] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1-3):461–474, 1993.

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

[45] F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: a self-organizing framework for information extraction. In *WWW*, pages 631–640, 2009.

[46] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.

[47] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[48] Y. Tao, W. Lin, and X. Xiao. Minimal MapReduce algorithms. *SIGMOD*, pages 529–540, 2013.

[49] Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *ISWC*. 2011.

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

Appendix

Optimization Strategies

We next present optimization strategies employed by our algorithms.

Skewed graphs. Real-life graphs, *e.g.,* social networks, may exhibit skewed distribution, *i.e.,* most nodes have few neighbors while a small fraction of nodes are adjacent to a large fraction of the edges. Such skewed distribution may lead to a small number of large data blocks $G_{\bar{z}}$, and hence skewed workload.

Procedures bPar and disPar can readily adapt to skewed graphs, by applying a *replicate and split* strategy for large $G_{\bar{z}}$. (a) We set a threshold θ for $|G_{\bar{z}}|$. (b) For work units w with $G_{\bar{z}}$ exceeding θ, we replicate w with the same $\bar{z}$, but split $G_{\bar{z}}$ to subgraphs with size at most θ. (c) The original w is replaced by these new units. For these units, localVio and dlovalVio detect errors in G_z by shipping partial matches rather than data blocks.

We experimentally evaluated dlovalVio over skewed graphs. We generate synthetic graphs G controlled by skew measuring the "skewness" of G, quantified as a ratio $\frac{|G_{dm}|}{|G_{dm'}|}$, where $|G_{dm}|$ (resp. $|G_{dm'}|$) is the average size of top 10% smallest (resp. largest) d-hop neighbors of the nodes in G (we set d=3). It estimates the unbalance degree of data block sizes of work units. The smaller skew, the more unbalanced.

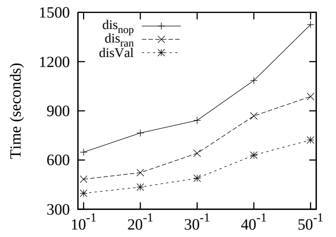


Figure 8: The impact of skewed graphs

Fixing $|G|$ = (10M, 20M) and $n = 16$, we varied skew from 0.1 to 0.02. Figure 8 shows that all the algorithms deteriorate when skew gets smaller ("more skewed"), due to unbalanced workload, as expected. However, disVal is more "robust" to skew: its running time only grows 1.7 times when G becomes 5 times more "skewed", as opposed to 2.0 times by $\mathsf{dis}_{\mathsf{ran}}$ and 2.2 by $\mathsf{dis}_{\mathsf{nop}}$. This verifies the effectiveness of our "replicate and split strategy".

Multi-query processing. There has been work on optimizing muilti-pattern matching by extracting common sub-patterns [31]. Algorithm repVal and disVal use some of the techniques, *e.g.,* pattern containment and sub-pattern scheduling, to process common sub-patterns in Σ, and to remove redundant work units (*e.g.,* Example 10).

Workload reduction. The implication analysis of GFDs can help us further reduce workload. Given a set Σ of GFDs, if $\Sigma \setminus \{\varphi\} \models \varphi$, we can safely remove φ from Σ without impacting $\mathsf{Vio}(\Sigma, G)$. As shown in Theorem 5, the implication problem is intractable. Nonetheless, it is in PTIME if the patterns in Σ are trees (Corollary 8), and moreover, there are heuristic algorithms to conduct the analysis efficiently.

Compared with Other Approaches

GFDs *vs. other models*. We evaluated the effectiveness of GFDs for error detection with *YAGO2*, by comparing with (a) the extension of CFDs to RDF [23], referred to as GCFDs, and (b) BigDansing [28]. Since the complete set of "true" errors in *YAGO2* is unknown, we sampled a set of entities. For each sampled entity x, we randomly injected noise (with probability 2%, 2690 errors in total) into *YAGO2* as suggested by [50]: (a) *attribute inconsistency*, by changing the value of an attribute $x.A$; (b) *type inconsistency*, by revising the type of x; and (c) *representational inconsistency*, by revising the value of either $x.A$ or $x'.A$ if $x.A$=$x'.A$ and x and x' are of the same type. Denote the set of entities with noise as Vio, we define the *precision* (resp. *recall*) of an error detection method $\mathcal{A}$ as $\frac{|\mathsf{Vio} \cap \mathsf{Vio}(\mathcal{A})|}{|\mathsf{Vio}(\mathcal{A})|}$ (resp. $\frac{|\mathsf{Vio} \cap \mathsf{Vio}(\mathcal{A})|}{|\mathsf{Vio}|}$, where $\mathsf{Vio}(\mathcal{A})$ denotes the inconsistent entity set detected by $\mathcal{A}$.

We constructed (1) a set Σ of 10 GFDs on *YAGO2* with frequent patterns that match a fraction of sampled entities and with constants from the original values before noise injection; and (2) a set of 7 GCFDs over sampled entities following [23], including *all* GFDs in Σ with conjunctive paths (GCFDs do not allow general graph patterns). (3) We hard-coded the GFDs as user-defined functions for each GFD in Σ, as BigDansing does not support subgraph isomorphism.

We report the running time and accuracy of these methods in Fig. 9, with $n = 16$ on *YAGO2* extended with noise. We find that (a) GFDs has higher accuracy (91%) than GCFDs,

model	recall	prec.	time
GFD	**0.91**	**1.0**	**131s**
GCFD	0.57	1.0	106s
BigDansing	0.91	1.0	609s

Figure 9: Running time and accuracy

since it catches inconsistencies with general patterns not expressible by GCFDs; (b) it takes comparable time for GFDs and GCFDs; and (c) BigDansing is 4.6 times slower, because it had to cast subgraph isomorphic testing as relational joins. It reports the same accuracy as our algorithm since it hard-coded the same set Σ of GFDs.

Real-world GFDs. Observe the following about the GFDs depicted in Fig. 7.

GFD *1* is not expressible as (a) a GCFD since Q_{10} is a cyclic pattern, or (b) a CFD or denial constraint (DC) of BigDansing, since otherwise it gets false negative if subgraph isomorphism is not enforced.

GFD *2* is not expressible as GCFD, CFD or DC for the same reason as GFD 1.

GFD *3* is is not expressible as GCFD although Q_{12} is a tree, since GCFD cannot do the test $z.\mathsf{id} = z'.\mathsf{id}$; similarly for CFD and DC of BigDansing.

Summary. From the experimental results we can see the following. (1) GFDs are more accurate than GCFDs in error detection. (2) BigDansing requires users to code GFDs and subgraph isomorphism, and is 4.6 times slower than our algorithm.