2018 IEEE 34th International Conference on Data Engineering

Parallel Reasoning of Graph Functional Dependencies

Wenfei Fan[1] Xueli Liu[2] Yingjie Cao[3]
[1]*University of Edinburgh* wenfei@inf.ed.ac.uk
[2]*Harbin Institue of Technology* xueliliu@hit.edu.cn
[3]*Beihang University* caoyj@act.buaa.edu.cn

***Abstract*—This paper develops techniques for reasoning about graph functional dependencies (GFDs). We study the satisfiability problem, to decide whether a given set of GFDs has a model, and the implication problem, to decide whether a set of GFDs entails another GFD. While these fundamental problems are important in practice, they are coNP-complete and NP-complete, respectively. We establish a small model property for satisfiability, showing that if a set Σ of GFDs is satisfiable, then it has a model of a size bounded by the size $|\Sigma|$ of Σ; similarly we prove a small model property for implication. Based on the properties, we develop algorithms for checking the satisfiability and implication of GFDs. Moreover, we provide parallel algorithms that guarantee to reduce running time when more processors are used, despite the intractability of the problems. We experimentally verify the efficiency and scalability of the algorithms.**

I. Introduction

Several classes of graph dependencies have recently been proposed to extend functional dependencies (FDs) from relations to graphs [1]–[8], referred to as *graph functional dependencies* (GFDs). The need for GFDs is evident in inconsistency detection, knowledge acquisition, knowledge base enrichment, and spam detection, among other things.

There are two fundamental problems for GFDs. One is the satisfiability problem, to decide whether a set Σ of GFDs has a model, *i.e.,* a nonempty graph that satisfies all GFDs in Σ. The other is the implication problem, to decide whether a GFD φ is entailed by a set Σ of GFDs, *i.e.,* for any graph G, if G satisfies Σ then G satisfies φ. These are classical problems associated with any dependency class, known as the static analyses.

For GFDs, these problems not only are of theoretical interest, but also find practical applications. The satisfiability analysis helps us check whether a set Σ of GFDs discovered from (possibly dirty) real-life graphs is "dirty" itself before it is used to detect errors and spam. The implication analysis eliminates redundant GFDs that are entailed by others. That is, the implication analysis provides us with an optimization strategy to speed up, *e.g.,* error detection process.

No matter how important, these problems are hard for GFDs. For relational FDs, the satisfiability problem is trivial: any set of FDs can find a nonempty relation that satisfies the FDs [9]. The implication problem is in linear time (cf. [10]). In contrast, for GFDs of [1], [2], the satisfiability and implication problems are coNP-complete and NP-complete, respectively. This is not very surprising. GFDs on graphs are more complicated than FDs on relations. A GFD is a combination of (a) a graph pattern Q, to identify entities in a graph, and (b) an "attribute dependency" $X \to Y$ that is applied to the entities identified [2]. Since graph pattern matching is NP-complete under the semantics of homomorphism (cf. [11]), the static analyses of GFDs are inevitably intractable.

This raises several questions. To check whether a set Σ of GFDs is satisfiable, what graphs G should we inspect to find a model of Σ? To decide whether Σ implies another GFD φ, do we have to examine all graphs G that satisfy Σ and check whether G satisfies φ? Is it feasible to reason about GFDs in practice? That is, does there exist effective technique for checking the satisfiability and implication of GFDs?

Contributions. This paper develops practical parallel algorithms for the satisfiability and implication analyses of GFDs. We consider the GFDs of [2] defined on generic graphs.

(1) We characterize the satisfiability of GFDs (Section IV). We show a small model property: a set Σ of GFDs is satisfiable if and only if (iff) there exists a graph G such that G satisfies Σ and the size $|G|$ of G is bounded by the size $|\Sigma|$ of Σ. This allows us to inspect graphs G of a bounded size as candidate models of Σ. Based on this, we develop a sequential (exact) algorithm SeqSat to check GFD satisfiability.

(2) We develop a parallel algorithm ParSat to check GFD satisfiability (Section V). One might think that the more processors are used, the faster a parallel algorithm would run. Unfortunately, this is not for granted. Many parallel algorithms do not warrant this. Worse yet, for some computation problems, parallel scalability is beyond reach [12], *i.e.,* no parallel algorithms would run faster given more processors.

We show that ParSat has this performance guarantee. Adopting a notion introduced [13], we show that ParSat is parallel scalable relative to SeqSat: its parallel running time is in $O(t(|\Sigma|)/p)$, where $t(|\Sigma|)$ denotes the cost of SeqSat and p is the number of processors used. As a result, it guarantees to reduce the running time when more processors are used. Hence it is feasible to scale with large Σ by increasing p, despite the intractability of GFD satisfiability.

(3) We parallelize GFD implication checking (Section VI). We show another small model property: to check whether a set Σ of GFDs implies another GFD φ, it suffices to inspect graphs of size bounded by the sizes of φ and Σ, and enforce the GFDs of Σ on the small graphs. Based on this, we develop a sequential exact algorithm SeqImp to check GFD implication. We then develop an algorithm ParImp by parallelizing SeqImp. We show that ParImp is parallel scalable relative to SeqImp, allowing us to scale with large sets Σ of GFDs.

Algorithms ParSat and ParImp explore various techniques for parallel reasoning, such as (a) a combination of data-partitioned parallelism and pipelined parallelism [14], for early termination of checking; (b) dynamic workload assignment and work unit splitting to handle stragglers; and (c) a topological order on work units based on a dependency graph.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00060

(4) Using real-life and synthetic GFDs, we empirically verify the efficiency and scalability of our algorithms (Section VII). We find the following. (a) On average SeqSat and SeqImp take 1848 and 909 seconds on up to 10000 real-life GFDs with fairly complex patterns, respectively. The performance is substantially improved by parallel ParSat and ParImp, which take 167 and 76 seconds, respectively, when $p = 20$. Hence it is feasible to reason about GFDs in practice by using the parallel algorithms. (b) ParSat and ParImp are parallel scalable: they are 3.4 and 3.6 times faster on average, respectively, when p varies from 4 to 20. (c) Our optimization strategies are effective, *e.g.,* pipelining improves the performance of parallel ParSat and ParImp by 1.5 and 1.6 times on average, and work unit splitting improves 3.8 and 4.1 times, respectively.

These algorithms yield a promising tool for reasoning about GFDs, to validate data quality rules and optimize rule-based process for cleaning graph data, among other things. To the best of our knowledge, no parallel algorithms are yet in place for the static analyses of graph dependencies.

We discuss related work in Section VIII and future work in Section IX. The proofs of the results of the paper are in [15].

II. Preliminaries

We start with basic notations. Assume two countably infinite alphabets Γ and Θ for labels and attributes, respectively.

Graphs. We consider directed graphs $G = (V, E, L, F_A)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$, in which (v, v') denotes an edge from node v to v'; (3) each node $v \in V$ is labeled $L(v) \in \Gamma$; similarly we define $L(e)$ for edge $e \in E$; 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, $A_i \in \Theta$ is an *attribute* of v, written as $v.A_i = a_i$, and $A_i \neq A_j$ if $i \neq j$; the attributes carry content as in property graphs.

A graph (V', E', L', F'_A) is a *subgraph of* (V, E, L, F_A) if $V' \subseteq V$, $E' \subseteq E$, for each node $v \in V'$, $L'(v) = L(v)$ and $F'_A(v) = F_A(v)$, and for each edge $e \in E'$, $L'(e) = L(e)$.

Graph patterns. A *graph pattern* is a graph $Q[\bar{x}] = (V_Q, E_Q, L_Q)$, where (1) V_Q (resp. E_Q) is a finite set of pattern nodes (resp. edges); (2) L_Q is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) to nodes $u \in V_Q$ (resp. edges $e \in E_Q$); and (3) $\bar{x}$ is a list of distinct variables denoting nodes in V.

Labels $L_Q(u)$ and $L_Q(e)$ are taken from Γ and moreover, we allow $L_Q(u)$ and $L_Q(e)$ to be wildcard '_'.

Pattern matching. A *match* of pattern $Q[\bar{x}]$ in a graph G is a homomorphism h from Q to G such that (a) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (b) for each edge $e = (u, u')$ in Q, $e' = (h(u), h(u'))$ is an edge in G and $L_Q(e) = L(e')$. In particular, $L_Q(u) = L(h(u))$ if $L_Q(u)$ is '_', *i.e.,* wildcard indicates generic entities and can match any label in Γ.

We also denote the match as a vector $h(\bar{x})$ if it is clear from the context, where $h(\bar{x})$ consists of $h(x)$ for each $x \in \bar{x}$. 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.

III. Graph Functional Dependencies

We next review graph functional dependencies studied in [2], referred to as GFDs, from syntax to semantics.

Syntax. A GFD φ is a pair $Q[\bar{x}](X \to Y)$ [1], 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 $\bar{x}$.

A *literal* of $\bar{x}$ is either $x.A = c$ or $x.A = y.B$, where x and y are variables in $\bar{x}$ (denoting nodes in Q), A and B are attributes in Θ (not specified in Q), and c is a constant.

Intuitively, GFD φ specifies two constraints: (a) a *topological constraint* Q, and (b) an attribute dependency $X \to Y$. Pattern Q specifies the scope of the GFD: it identifies subgraphs of G on which $X \to Y$ is enforced. As observed in [2], attribute dependencies $X \to Y$ subsume relational EGDs and CFDs, in which FDs are a special case. In particular, literals $x.A = c$ carry constant bindings along the same lines as CFDs [16]. Following [1], we refer to $Q[\bar{x}](X \to Y)$ as graph functional dependencies (GFDs).

Semantics. For a match $h(\bar{x})$ of Q in a graph G and a literal $x.A = c$ of $\bar{x}$, we say that $h(\bar{x})$ *satisfies* the literal if *there exists* attribute A at the node $v = h(x)$ and $v.A = c$; similarly for literal $x.A = y.B$. We denote by $h(\bar{x}) \models X$ if $h(\bar{x})$ satisfies *all* the literals in X; similarly for $h(\bar{x}) \models Y$.

We write $h(\bar{x}) \models X \to Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$.

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

Intuitively, $G \models \varphi$ if for each match $h(\bar{x})$ identified by Q, the attributes of the entities in $h(\bar{x})$ satisfy $X \to Y$.

Example 1: Consider GFDs defined with patterns Q_1-Q_4 shown in Fig. 1. These GFDs are able to catch semantic inconsistencies in real-life knowledge bases and social graphs.

(1) GFD $\varphi_1 = Q_1[x, y](\emptyset \to \mathsf{false})$. It states that for any place x, if x is located in another place y, then y should not be part of x. Here X is $\emptyset$, and Boolean constant false is a syntactic sugar for, *e.g.,* $x.A = c$ and $x.A = d$ with distinct constants c and d. The GFD is defined with a cyclic pattern Q_1.

In DBpedia, Bamburi_airport is located in city Bamburi, but at the same time, Bamburi is put as part of Bamburi airport. Hence DBpedia does not satisfy φ_1, and the violation is caught by match $h : x \mapsto$ Bamburi_airport and $y \mapsto$ Bamburi of Q_1. The inconsistency is detected by φ_1.

(2) GFD $\varphi_2 = Q_2[x, y, z](\emptyset \to y.\mathsf{val} = z.\mathsf{val})$, where val is an attribute of y and z. It says that the topSpeed is a functional property, *i.e.,* an object has at most one top speed. Note that x is labeled wildcard '_', and may denote, *e.g.,* car, plane.

The GFD catches the following error in DBpedia: tanks are associated with two topSpeed values, 24.076 and 33.336.

(3) GFD $\varphi_3 = Q_3[x, y, z, w](x.c = y.c \to z.\mathsf{val} = w.\mathsf{val})$, where c is an attribute of x and y indicating country, and val is an attribute of z and w indicating value. The GFD states if x and y are the president and vice president of the same country, then x and y must have the same nationality. It catches the

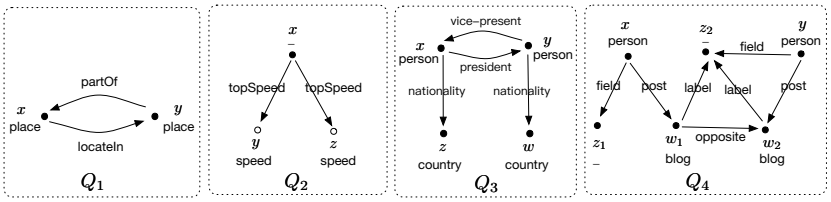


Fig. 1: Graph patterns

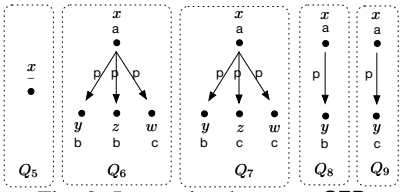


Fig. 2: Interaction between GFDs

following inconsistency in DBpedia: the president and vice-president of Botswana have nationality Botswana and Tswana, respectively, while Tswana is ethnicity, not nationality.

(4) GFD $\varphi_4 = Q_4[x, y, z_1, z_2, w_1, w_2](w_1.\text{topic} = w_1.\text{topic} \rightarrow w_2.\text{trust} = \text{"low"})$, where w_1 and w_2 carry attribute topic. It states that in a social network, if blogs w_1 and w_2 are posted by people x and y, respectively, w_1 and w_2 give inconsistent accounts of the facts on the same topic, and if x is a domain expert on the subject but y is not, then the account given by y has low credibility. For instance, if a computer scientist x and a politician y gave two accounts of facts about the future of databases, then the comment from y can be discounted. □

We consider graphs that typically do not have a schema, as found in the real world. Hence a node v may not necessarily have a particular attribute. For a literal $x.A = c$ in X, if $h(x)$ has *no* attribute A, then $h(\bar{x})$ trivially satisfies $X \rightarrow Y$ by the definition of $h(\bar{x}) \models X$. In contrast, if $x.A = c$ is in Y and $h(\bar{x}) \models Y$, then $h(x)$ must have attribute A by the definition of satisfaction; similarly for $x.A = y.B$.

In particular, if X is $\emptyset$, then $h(\bar{x}) \models X$ for any match $h(\bar{x})$ of Q in G, and Y has to be enforced on $h(\bar{x})$. In this case, if Y includes a literal $x.A = c$, then $h(x)$ must carry attribute A. If $Y = \emptyset$, then Y is true, and φ is trivially satisfied.

IV. Characterizing GFD Satisfiability

We first study the satisfiability problem for GFDs. We start with notations for formulating the problem.

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

Intuitively, if Σ has a model, then the GFDs in Σ are consistent, *i.e.,* they do not conflict with each other, since all of them can be applied to the same graph.

We say that Σ is *satisfiable* if Σ has a *model*.

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

It is known that the problem is coNP-complete [2]. However, [2] does not tell us how to develop a deterministic algorithm to check GFD satisfiability. In light of this, we establish a small model property of the problem (Section IV-B). Based on the property, we provide an exact algorithm for satisfiability checking (Section IV-C).

A. The Challenges of Satisfiability Checking

As opposed to relational FDs, a set Σ of GFDs may not be satisfiable. In fact, even if each GFD in Σ is satisfiable, Σ may not have a model, because the GFDs in Σ may interact with each other.

Example 2: Consider two GFDs defined with the same pattern Q_5 depicted in Fig. 2: $\varphi_5 = Q_5[x](\emptyset \rightarrow x.A = 0)$ and $\varphi_6 = Q_5[x](\emptyset \rightarrow x.A = 1)$, where Q_5 has a single node x labeled '_'. Then no nonempty graph G satisfies both φ_5 and φ_6. For if such G exists, φ_5 and φ_6 require $h(x).A$ to be 0 and 1, respectively, which is impossible; here $h(x)$ is a match of x.

GFDs defined with distinct patterns may also interact with each other. Consider GFDs: $\varphi_7 = Q_6[x, y, z, w](\emptyset \rightarrow x.A = 0 \wedge y.B = 1)$ and $\varphi_8 = Q_7[x, y, z, w](y.B = 1 \rightarrow x.A = 1)$, with Q_6 and Q_7 shown in Fig. 2. One can easily see that each of φ_7 and φ_8 has a model. However, there exists no model G for both φ_7 and φ_8. Indeed, if such G exists, then Q_6 has a match h in G: $h(x) \mapsto v, h(y) \mapsto v_b, h(z) \mapsto v_z, h(w) \mapsto v_c$. Hence φ_7 applies to the match and enforces $v_b.B = 1$. A match h' of Q_7 in G can be given as $h'(x) \mapsto v, h'(y) \mapsto v_b, h'(z) \mapsto v_c, h'(w) \mapsto v_c$, and φ_8 applies to the match since $h'(x, y, z, w) \models h'(y).B = 1$. As a result, φ_7 and φ_8 require node $v.A$ to be 1 and 0, respectively. □

As shown by Example 2, while Q_7 is not homomorphic to Q_6 and vice versa, φ_7 and φ_8 can be enforced on the same node. Thus GFD satisfiability is nontrivial. It is shown coNP-hard by reduction from the complement of 3-colorability [2].

B. A Small Model Property

To find a model of a set Σ of GFDs, we cannot afford to enumerate all (infinitely many) finite graphs G and check whether $G \models \Sigma$. This motivates us to establish a small model property for the problem, to reduce the search space.

Canonical graphs. We borrow a notation from [2]. The *canonical graph* G_Σ of Σ is defined to be $(V_\Sigma, E_\Sigma, L_\Sigma, F_A^\Sigma)$, where (a) V_Σ is the union of V_i's, (b) E_Σ is the union of E_i's, and (c) L_Σ is the union of L_i's; but (d) F_A^Σ is empty. We assume *w.l.o.g.* that patterns in Σ are pairwise disjoint, *i.e.,* their nodes are denoted by distinct variables by renaming.

Intuitively, G_Σ is the union of all graph patterns in Σ, in which patterns from different GFDs are disjoint. We keep wildcard _ of Q in G_Q and treat it as a "normal" label such that only _ in a pattern can match _ in G_Σ.

Example 3: Consider a set Σ consisting of φ_7 and φ_8 of Example 2. Its canonical graph G_Σ is the graph by putting together Q_6 and Q_7 of Fig. 2, except that variables x, y, z, w in Q_7 are renamed as, *e.g.,* x', y', z', w', respectively. □

A *population* of G_Σ is a graph $G = (V_\Sigma, E_\Sigma, L_\Sigma, F_A)$, where F_A is a function that for each node $v \in V_\Sigma$, assigns $F_A(v) = (A_1 = a_1, \ldots, A_m = a_m)$, a (finite) tuple of attributes from Θ and their corresponding constant values.

Population G of G_Σ is said to be Σ-*bounded* if all attribute values in F_A have total size bounded by $O(|\Sigma|)$, *i.e.,* the values of all attributes in G are determined by Σ alone.

Intuitively, G and G_Σ have the same topological structure and labels, and G extends G_Σ with attributes and values. It is Σ-bounded if its size $|G|$ is in $O(|\Sigma|)$, including nodes, edges, attributes and all constant values in G.

Small model property. We next show that to check the satisfiability of Σ, it suffices to inspect Σ-bounded populations of the canonical graph G_Σ of Σ. We will develop a satisfiability checking algorithm based on this small model property.

Theorem 1: *A set Σ of* GFDs *is satisfiable iff there exists a model G of Σ that is an Σ-bounded population of G_Σ.* □

Proof: If there exists an Σ-bounded population of G_Σ that is a model of Σ, then obviously Σ is satisfiable. Conversely, if Σ has a model G, then there exists a homomorphism h from G_Σ to G. Employing h, we construct an Σ-bounded population G' of G_Σ. We populate attributes of G' by taking only relevant attributes from G, and by normalizing these attributes to make them Σ-bounded. The population preserves the constant values that appear in Σ and the equality on the attributes. We show that $G' \models \Sigma$ by contradiction (see [15] for details). □

As an immediate corollary, we give an alternative proof for the upper bound of the satisfiability problem for GFDs, instead of revising and using the chase as in [2].

Corollary 2: *The* GFDs *satisfiability problem is in* coNP. □

Proof: We give an NP algorithm to check whether a set Σ of GFDs is *not* satisfiable, as follows: (a) guess an Σ-bounded attribute population G of G_Σ, and a match h_i for each pattern Q_i of Σ in G; (b) check whether each h_i makes a match; if so, (c) check whether the matches violate any GFD in Σ in G. The correctness follows from Theorem 1. The algorithm is in NP since steps (b) and (c) are in PTIME (polynomial time). Thus the satisfiability problem is in coNP. Note that we cannot guess G as above and check whether $G \models \Sigma$, since checking $G \models \Sigma$ is already coNP-complete itself [2]. □

C. A Sequential Algorithm for Satisfiability

Based on the small model property, we develop an exact algorithm, referred to as SeqSat, that takes as input a set Σ of GFDs, and returns true if and only if Σ is satisfiable.

Algorithm. Algorithm SeqSat first builds the canonical graph $G_\Sigma = (V_\Sigma, E_\Sigma, L_\Sigma, F_A^\Sigma)$ of Σ. It then processes each GFD $\varphi = Q[\bar{x}](X \to Y)$ in Σ and populates F_A^Σ by invoking procedure Expand. Expand (a) finds matches $h(\bar{x})$ of Q in G_Σ; and (b) checks whether $h(\bar{x}) \models X$ and if so, it adds attributes $x.A$ to F_Σ and/or instantiates attributes $x.A$ with constants for each literal $x.A = c$ or $x.A = y.B$ in Y, *i.e.,* it "enforces" φ on the match $h(\bar{x})$. If a *conflict* emerges, *i.e.,* if there exists $x.A$ such that $x.A$ is assigned two distinct constants, SeqSat terminates with false. The process iterates until all GFDs in Σ are processed. If no conflict occurs, SeqSat returns true.

Algorithm SeqSat supports *early termination*. It terminates with false as soon as a conflict is detected. Moreover, when a match $h(\bar{x})$ is found, it *expands F_A^Σ by enforcing φ at match* $h(\bar{x})$, instead of waiting until all matches of Q are in place.

The correctness of SeqSat is assured by the following: (a) it suffices to inspect populations of G_Σ by Theorem 1, and (b) attributes are populated by enforcing GFD φ on each match $h(\bar{x})$ of Q, which is necessary for any population of G_Σ to satisfy Σ, by the semantics of GFD satisfaction.

We next provide more details about algorithm SeqSat.

Equivalence class. To speedup checking, we represent F_Σ as an equivalence relation Eq. For each node $x \in V_\Sigma$ and each attribute A of x, its equivalence class, denoted by $[x.A]_{\mathsf{Eq}}$, is a set of attributes $y.B$ and constants c, such that $x.A = y.B$ and $x.A = c$ are enforced by GFDs in Σ (see below). One can easily verify that Eq is reflexive, symmetric and transitive.

Given a GFD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, Expand generates matches $h(\bar{x})$ of Q in G_Σ along the same lines as VF2 [17] for subgraph isomorphism, except enforcing homomorphism rather than isomorphism. Then for each match $h(\bar{x})$ found, Expand checks whether $h(\bar{x}) \models X$. If so, it *expands* Eq *by enforcing φ at $h(\bar{x})$*, with the following rules.

(Rule 1) If l is $x.A = c$, it checks whether $[x.A]_{\mathsf{Eq}}$ does not yet exist in Eq. If so, it adds $[x.A]_{\mathsf{Eq}}$ to Eq and c to $[x.A]_{\mathsf{Eq}}$. If $[x.A]_{\mathsf{Eq}}$ has a constant $d \neq c$, it stops the process and SeqSat terminates with false immediately.

(Rule 2) If l is $x.A = y.B$, it checks whether $[x.A]_{\mathsf{Eq}}$ and $[y.B]_{\mathsf{Eq}}$ are in Eq. If not, it adds the missing ones to Eq, and merges $[x.A]_{\mathsf{Eq}}$ and $[y.B]_{\mathsf{Eq}}$ into one. If the merged class includes distinct constants, SeqSat terminates with false.

That is, Expand generates new attributes, instantiates and equalizes attributes as required the satisfiability of GFDs.

There is a complication when checking $h(\bar{x}) \models X$. For a literal $x.A = c$ in X, $x.A$ may not yet exist in F_A^Σ or is not instantiated (*i.e.,* $[x.A]_{\mathsf{Eq}}$ does not include any constant). To cope with this, we do the following.

(a) Algorithm SeqSat processes GFDs of the form $Q[\bar{x}](\emptyset \to Y)$ first, if any in Σ. These add an initial batch of attributes.

(b) Expand maintains a list of matches $h(\bar{x})$ and an *inverted index* with attribute $h(x).A$ that appears in X, but either $[h(x).A]_{\mathsf{Eq}}$ does not exist or is not instantiated. When $h(x).A$ is instantiated in a later stage, $h(\bar{x})$ is efficiently retrieved by the inverted index using $h(x).A$, and is checked again.

(c) At the end of the process of SeqSat, some attribute $x.A$ in Eq may still not be instantiated. The missing values do not affect the decision of SeqSat on the satisfiability of Σ, since we can always complete F_A^Σ by assigning a distinct constant to each of such $[x.A]_{\mathsf{Eq}}$, without inflicting conflicts.

Example 4: Consider $\Sigma = \{\varphi_7, \varphi_9, \varphi_{10}\}$, where φ_7 is given in Example 2, $\varphi_9 = Q_6[\bar{x}](y.B = 1 \to w.C = 1)$ and $\varphi_{10} = Q_7[\bar{x}](w.C = 1 \to x.A = 1)$, with Q_6 and Q_7 of Fig. 2. Its canonical graph G_Σ is similar to the one given in Example 3,

with Q_7 and two distinct copies of Q_6 (from φ_7 and φ_9). Assume that SeqSat checks φ_7, φ_{10}, φ_9 in this order.

(1) For φ_7, Expand finds a match of Q_6 in G_Σ: $h(x) \mapsto x, h(y) \mapsto y, h(z) \mapsto z, h(w) \mapsto w$. Since $h(\bar{x}) \models \emptyset$, it adds $[x.A]_{\mathsf{Eq}}$ ($[y.B]_{\mathsf{Eq}}$) to Eq and "0" ("1") to $[x.A]_{\mathsf{Eq}}$ ($[y.B]_{\mathsf{Eq}}$).

(2) When processing φ_{10}, Expand finds a match of Q_7 in G_Σ: $h'(x) \mapsto x, h'(y) \mapsto y, h'(z) \mapsto w, h'(w) \mapsto w$. As $[w.C]_{\mathsf{Eq}}$ is not in Eq, it adds $(h'(\bar{x}), \varphi_{10})$ to an inverted index with $w.C$.

(3) When processing φ_9, Expand finds a match of Q_6: $h_1(x) \mapsto x, h_1(y) \mapsto y, h_1(z) \mapsto y, h(w) \mapsto w$. It adds $w.C$ to $[y.B]_{\mathsf{Eq}}$. This triggers re-checking of $(h'(\bar{x}), \varphi_{10})$ with the inverted index. Now $h'(\bar{x}) \models w.C = 1$. Expand adds "1" to $[x.A]_{\mathsf{Eq}}$, to enforce φ_{10}. However, "0" is already in $[x.A]_{\mathsf{Eq}}$, a conflict. Hence SeqSat stops and returns false. □

Analysis. Algorithm SeqSat enforces GFDs of Σ by the semantics of GFDs. By Theorem 1, it returns true iff Σ has a model. One can verify that SeqSat guarantees to converge at the same result no matter in what order the GFDs of Σ are applied, *i.e.,* Church-Rosser, along the same lines as the characterization of GFD satisfiability in [2]. We will see how to order GFDs applied in Section V. SeqSat terminates early as soon as a conflict is spotted, and does not enumerate all matches by pruning to eliminate irrelevant matches early.

V. Checking Satisfiability in Parallel

Algorithm SeqSat is an exact algorithm. When Σ is large, it is costly due to the intractable nature of the satisfiability problem. This motivates us to parallelize SeqSat.

Below we first review a characterization of parallel algorithms (Section V-A). We then develop a parallel algorithm ParSat with performance guarantees (Section V-B).

A. Parallel Scalability

As remarked in Section I, a parallel algorithm for a problem may not necessarily reduce its sequential running time. This suggests that we characterize the effectiveness of parallel algorithms. To this end, we revise a notion of parallel scalability introduced by [13] and widely used in practice.

We say that an algorithm $\mathcal{A}_p$ for GFD satisfiability checking is *parallel scalable relative to sequential algorithm* SeqSat if its running time can be expressed as:

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

where $t(|\Sigma|)$ denotes the cost of SeqSat, and p is the number of processors employed by $\mathcal{A}_p$ for parallel computation.

Intuitively, a parallel scalable $\mathcal{A}_p$ linearly reduces the sequential cost of SeqSat when p increases. The main conclusion we can draw from the parallel scalability is that by taking SeqSat as a yardstick, $\mathcal{A}_p$ guarantees to run faster when adding more processors, and hence scale with large Σ.

B. A Parallel Scalable Algorithm for Satisfiability

We develop an algorithm for checking the satisfiability of a set Σ of GFDs, denoted as ParSat, by parallelizing SeqSat. We show that ParSat is parallel scalable relative to SeqSat.

The novelty of algorithm ParSat includes the following. (1) It makes use of both *data partitioned parallelism* and *pipelined parallelism* to speed up the process and facilitate interactions between GFDs. (2) It proposes *dynamic workload balancing* and *work unit splitting*, to handle stragglers, *i.e.,* work units that take substantially longer than the others (see below). (3) It deduces a topological order on work units to reduce the impact of their interaction, based on a *dependency graph*. (4) It retains the *early termination* property of SeqSat.

Below we present the details of algorithm ParSat.

Setting. ParSat works with a coordinator S_c and p workers $(P_1, \cdots, P_p)$. Following [1], [18], we replicate canonical graph G_Σ at each worker, to reduce graph partition complication and communication costs. This is feasible since G_Σ is much smaller than real-life data graphs such as social networks, which have billions of nodes and trillions of edges [19].

We adopt the notion of work units of [1]. Consider a GFD $\varphi = Q[\bar{x}](X \to Y)$. To simplify the discussion, assume *w.l.o.g.* that Q is connected. A node $x \in \bar{x}$ is designated as a *pivot* of Q. A *work unit* w *of* Σ is a pair $(Q[z], \varphi)$, where z is a node in G_Σ that matches the label of x. Intuitively, w indicates a set of candidate matches of Q to be checked.

Intuitively, we use pivot x to explore the data locality of graph homomorphism: for any v in G_Σ, if there exists a match h of Q in G_Σ such that $h(x) = v$, then $h(\bar{x})$ consists of only nodes in the d_Q-neighbor of v. Here d_Q is the *radius* of Q at v, *i.e.,* the longest shortest path from v to any node in Q. The d_Q neighbor of v includes all nodes and edges within d_Q hops of v. Thus each candidate match v of x determines a work unit, namely, the d_Q-neighbor of v, and we can check these work units in parallel. Ideally, we pick a pivot x that is selective, *i.e.,* it carries a label that does not occur often in G_Σ; nonetheless, any node x in $\bar{x}$ can serve as a pivot.

When Q is disconnected, a work unit is $(Q[\bar{z}], \varphi)$, where $\bar{z}$ includes a pivot for each connected component of Q [1].

Algorithm. As shown in Fig. 3, ParSat works as follows.

(1) Coordinator. Given Σ, coordinator S_c first (a) builds its canonical graph G_Σ and replicates G_Σ at each worker (line 1), and (b) constructs a priority queue W of all work units of Σ (line 2), following a topological order based on a dependency graph of Σ (see details below). It then activates each worker P_i with one work unit w from the front of W (line 3). In fact, work units can be assigned to worker in a small batch rather than a single w, to reduce the communication cost.

The coordinator then interacts with workers and dynamically assigns workload, starting from the units of W with the highest priority (line 4-10). A worker P_i may send two flags to S_c: (a) f_i^c if P_i detects a conflict when expanding equivalence relation Eq (Section IV-C); and (b) f_i^d if P_i is done with its work unit. If S_c receives f_i^c for any $i \in [1, p]$, ParSat *terminates immediately with* false (lines 5-6). If S_c receives f_i^d, it assigns the next unit w' in W to P_i and removes w' from W (lines 7-8). The process iterates until either a conflict is detected, or W becomes empty, *i.e.,* all work units

have been processed. At this point algorithm ParSat concludes that Σ is satisfiable and returns true (line 11).

As will be seen shortly, a worker may split its unit w into a list L_i of sub-units if w is a straggler. Upon receiving L_i, S_c adds L_i to the front of the priority queue W (lines 9-10).

Putting these together, ParSat implements data partitioned parallelism (by distributing work units of W), dynamic workload assignment and early termination.

(2) Workers. Each worker P_i maintains the following: (a) local canonical graph G_Σ in which an equivalence relation Eq_i represents its local F_A^Σ, and (b) a buffer $\Delta\mathsf{Eq}^i$ that receives and stores updates to Eq_i from other workers. It processes its work unit w locally, and interacts with coordinator S_c and other workers asynchronously as follows.

(a) Local checking. Upon receiving a work unit $(Q[z], \varphi)$, P_i conducts local checking in the d_Q-neighbor of z. This suffices to find matches $h(\bar{x})$ of Q in G_Σ when the pivot x of Q is mapped to z, by the data locality of graph homomorphism.

Algorithm ParSat implements Expand (Section IV-C) with two procedures: (i) HomMatch finds matches $h(\bar{x})$ of Q in G_Σ pivoted at z, and (ii) CheckAttr expands Eq_i by enforcing φ at match $h(\bar{x})$ based on the two expansion rules of Section IV-C. It differs from Expand in the following.

- The two procedures work *in pipeline*: as soon as a match $h(\bar{x})$ is generated by HomMatch, CheckAttr is triggered to check $h(\bar{x})$ in a different thread, instead of waiting for all matches of Q to be found (lines 2-3 of HomMatch).
- When enforcing φ at $h(\bar{x})$, CheckAttr computes $\mathsf{Eq}_i^{(1)} = \mathsf{Eq}_i \cup \Delta\mathsf{Eq}^i$, incorporating changes $\Delta\mathsf{Eq}^i$ from other workers, and CheckAttr expands $\mathsf{Eq}_i^{(1)}$ to $\mathsf{Eq}_i^{(2)}$ with φ.

If a conflict emerges, *i.e.,* if some class $[y.B]_{\mathsf{Eq}_i^{(2)}}$ includes distinct constants, worker P_i sends flag f_i^c to coordinator S_c and terminates the process (line 4). After all matches of Q pivoted at z are processed, P_i sends flag f_i^d to S_c (line 9).

Like procedure Expand, CheckAttr also maintains an inverted index on matches $h(\bar{x})$ that need to be re-checked upon the availability of (instantiated) attributes needed.

(b) Interaction. Worker P_i broadcasts its local changes $\Delta\mathsf{Eq}_i$ to other workers (line 5). It keeps receiving changes $\Delta\mathsf{Eq}^i$ from other processors, and updates its local Eq_i by CheckAttr. The communication is *asynchronous*, *i.e.,* there is no need to coordinate the exchange through S_c. This does not affect the correctness of ParSat since equivalence relation Eq is *monotonically expanding*, and a conflict f_i^c terminates the process no matter at which worker f_i^c emerges.

Example 5: Recall $\Sigma = \{\varphi_7, \varphi_9, \varphi_{10}\}$ from Example 4. Its canonical graph G_Σ includes two copies of $Q_6[x, y, z, w]$ and a copy of $Q_7[x, y, z, w]$, in which variable x (designated as a pivot) in the three patterns is renamed as x_1, x_2, x_3, respectively; similarly for variables y, z, w.

We show how ParSat works with coordinator S_c and two workers P_1 and P_2. It first creates a priority queue W, where W has 9 work units $w_i = (Q_6[x_i], \varphi_7)$, $w_{3+i} = (Q_6[x_i], \varphi_9)$

Algorithm ParSat
Input: A set Σ of GFDs.
Output: true if Σ is satisfiable, and false otherwise.

/* executed at coordinator S_c */
1. construct canonical graph G_Σ and replicate it at each worker;
2. create priority queue W for work units, via dependency graph;
3. invoke HomMatch with initial work unit w at each worker P_i in parallel; remove w from W;
4. **repeat until** $W = \emptyset$;
5. **if** Sc receives a flag f_i^c **then**
6. terminate with false immediately;
7. **if** Sc receives a flag f_i^d **then**
8. send the first unit of W to P_i; remove the unit from W;
9. **if** Sc receives a list L_i of work units from a worker **then**
10. add L_i to the front of W;
11. **return** true;

/* executed at each worker P_i in parallel*/
Procedure HomMatch
Input: A work unit $w = (Q[z], \varphi)$; maintains G_Σ, Eq_i and $\Delta\mathsf{Eq}_i$.
Output: $\Delta\mathsf{Eq}_i$, and Boolean flags.
1. **repeat until** all matches of Q pivoted at z are processed
2. finds the next match $h(\bar{x})$ of Q at z;
3. $(\Delta\mathsf{Eq}_i, f_i^c)$:= CheckAttr$(h(\bar{x}), \varphi, \Delta\mathsf{Eq}^i)$;
4. **if** f_i^c **then** send f_i^c to coordinator Sc and terminate;
5. broadcast $\Delta\mathsf{Eq}_i$ to all workers;
6. **if** time spent exceeds TTL **then**
7. construct a list L_i of work units and send L_i to S_c;
8. complete the remaining work of w excluding those in L_i;
9. send f_i^d to S_c and terminate;

Procedure CheckAttr
Input: A match $h(\bar{x})$, GFD φ, buffer $\Delta\mathsf{Eq}^i$.
Output: Changes $\Delta\mathsf{Eq}_i$, and Boolean flag f_i^c.
1. $\mathsf{Eq}_i^{(1)} := \mathsf{Eq}_i \cup \Delta\mathsf{Eq}^i$; /* expand Eq_i with $\Delta\mathsf{Eq}^i$ */
2. expand $\mathsf{Eq}_i^{(1)}$ to get $\mathsf{Eq}_i^{(2)}$ by enforcing φ at φ;
3. compute $\Delta\mathsf{Eq}_i$; /* the difference between $\mathsf{Eq}_i^{(2)}$ and Eq */;
4. $\mathsf{Eq}_i := \mathsf{Eq}_i^{(2)}$;
5. **if** there exists a conflict in Eq_i **then** return $(\emptyset, f_o^c = \mathsf{true})$;
6. **else** return $(\Delta\mathsf{Eq}_i, f_o^c = \mathsf{false})$;

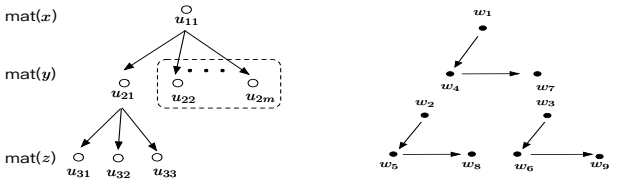
Fig. 3: **Algorithm** ParSat

and $w_{6+i} = (Q_7[x_i], \varphi_{10})$ in this order, for $i \in [1, 3]$ (see Example 7). Then S_c sends w_1 to P_1 and w_2 to P_2. It then dynamically assigns the remaining units to P_1 and P_2 one by one following the order of W, upon receiving f_j^d for $j \in [1, 2]$.

Suppose that at a stage, worker P_1 has processed w_1, w_3 and w_5, and P_2 has handled w_2, w_4 and w_6. After P_1 is done with w_5, it sends f_1^d to S_c and gets a new work unit w_7. At this point, Eq_1 at P_1 includes $[x_1.A]_{\mathsf{Eq}_1} = \{0, x_2.A, x_3.A\}$ and $[y_1.B]_{\mathsf{Eq}_1} = \{1, y_2.B, y_3.B, z_1.C, z_3.C, w_1.C, w_2.C, w_3.C\}$, after incorporating changes made at P_2. Now Expand finds a match $h(\bar{x})$: $h(x) \mapsto x_1$, $h(y) \mapsto y_2$, $h(z) \mapsto w_1$ and $h(w) \mapsto w_1$ at P_1. Assume *w.l.o.g.* that $\Delta\mathsf{Eq}^1 = \emptyset$, *i.e.,* no new changes are passed from P_2. By enforcing φ_{10} at match $h(\bar{x})$, CheckAttr obtains $\Delta\mathsf{Eq}_1$, which requires to add 1 to $[x.A]_{\mathsf{Eq}_1}$. At this point, it finds a conflict: both 0 and 1 are in $[x.A]_{\mathsf{Eq}_1}$. Hence CheckAttr sends f_1^c to S_c. When S_c receives f_1^c, ParSat returns false and terminates. □

Optimization. We speed up parallel checking as follows.

Unit splitting. A work unit $w = (Q[z], \varphi)$ assigned to a worker



(a) Work unit split (b) Dependency graph

Fig. 4: Work unit split and dependency graph

P_i may become a straggler and is "skewed". ParSat handles stragglers as follows. Recall that matching dominates the cost of w, and HomMatch computes matches via backtracking like in VF2 (Section IV-C). When it is triggered, it starts keeping track of the time τ spent on w. If τ exceeds a threshold TTL, it picks a set L_i of partial match $h(\bar{y})$ of Q, *i.e.,* $\bar{y}$ is a proper sub-list $\bar{x}$. It treats $(Q[\bar{y}], \varphi)$ as a work unit, and sends L_i to S_c (lines 6-7). Worker P_i resets $\tau = 0$ and continues with the remaining work of w excluding those in L_i (line 8).

Coordinator S_c adds L_i to the front of W, and distributes the units to workers as usual. Upon receiving such $(Q[\bar{w}], \varphi)$, worker P_j resumes the checking from $Q[\bar{w}]$.

Example 6: Consider a work unit $w = (Q[u_{11}], \varphi)$, where $\varphi = Q[x, y, z](X \to Y)$, in which x is pivoted at u_{11}. Suppose that at one point, HomMatch finds a match $h(x, y, z)$: $x \mapsto u_{11}$, $y \mapsto u_{21}$ and $z \mapsto u_{32}$ as shown in Fig. 4(a), and the time spent on w has exceeded TTL. Hence it decides to split w. It creates a list L_i consisting of $w_j = (Q[u_{11}, u_{2j}], \varphi)$ for $j \in [2, m]$, where w_j corresponds to a partial match $h_i[x, y]$: $x \mapsto u_{11}$ and $y \mapsto u_{2i}$, and it does not include match for z; here partial matches h_j are found by backtracking one step. HomMatch sends L_i to S_c, restarts counter τ and continues to complete the processing of the current match $h(x, y, z)$, to process, *e.g.,* matches in which z ranges over u_{32} and u_{33}.

Upon receiving L_i, coordinator S_c adds its units to the front of the priority queue, and assigns them to available workers as usual. When, *e.g.,* w_j is sent to a worker P_k, P_k resumes the processing of w_j starting from partial match $h_i[x, y]$. □

Dependency graph. We now show how to build priority queue W of work units (line 2 of ParSat). We construct a *dependency graph* $G_d = (V, E)$, where V is the set of work units, and (w_1, w_2) is a directed edge if (a) there exists an attribute $x.A$ that appears in both Y_1 and X_2, where $w_1 = (Q_1[z_1], \phi_1)$, $w_2 = (Q_2[z_2], \phi_2)$, $\phi_1 = Q_1[\bar{x}_1](X_1 \to Y_1)$, $\phi_2 = Q_2[\bar{x}_2](X_2 \to Y_2)$, *i.e.,* the antecedent X_2 of ϕ_2 may depend on the consequence Y_1 of ϕ_1; and (b) z_2 is within d_{Q_1} hops of z_1, *i.e.,* the two pivots are close enough to interact.

ParSat deduces a topological order from G_d and sorts W accordingly. Note that work units for GFDs $Q[\bar{x}](\emptyset \to Y)$ are at the front of queue W, with the highest priority.

Example 7: For the 9 work units of Example 5, the dependency graph G_d is depicted in Fig. 4. There exists an edge (w_1, w_4) since w_1 and w_4 are both pivoted at x_1, w_1 carries $\varphi_7 = Q_6[x, y, z, w](\emptyset \to x.A = 0 \wedge y.B = 1)$, w_4 carries $\varphi_9 = Q_6[\bar{x}](y.B = 1 \to w.C = 1)$, and $y.B$ is in both the consequence of φ_7 and the antecedent of φ_9; similarly for other edges in Fig. 4. In contrast, there is no edge between w_2 and w_4 since their pivots are not close, although they also carry φ_7 and φ_9, respectively. From G_d a topological order is deduced, to sort the work units of Example 5. □

As another optimization strategy, ParSat also extracts common sub-patterns that appear in multiple GFDs of Σ, finds matches of the sub-patterns at common pivots early, and reuses the matches when processing relevant GFDs. This is a common practice of multi-query optimization (*e.g.,* [20]). To avoid the complexity of finding common sub-patterns, following [21], we use graph simulation [22] to check whether a pattern Q_1 is homomorphic to a sub-pattern Q_2' of Q_2. In a nutshell, if Q_1 does not match Q_2' by simulation, then Q_1 is not homomorphic to Q_2'. Since graph simulation is in $O(|Q_1| \cdot |Q_2'|)$ time, this method reduces the (possibly exponential) cost of checking homomorphism.

Analysis. The correctness of ParSat is warranted by Theorem 1 and the fact that equivalence relation Eq is monotonically increasing, similar to the inflational semantics of fixpoint computation (see, *e.g.,* [10]). ParSat parallelizes SeqSat, and is parallel scalable relative to SeqSat by dynamic work unit assignment to balance workload, and work unit splitting to handle stragglers. One can verify by induction on the number of work units that the parallel runtime of ParSat is in $O(\frac{t(|\Sigma|)}{p})$, where $t(|\Sigma|)$ denotes the cost of SeqSat.

VI. Parallel Implication Checking

A set Σ of GFDs *implies* another GFD φ, denoted by $\Sigma \models \varphi$, if for all graphs G, if $G \models \Sigma$ then $G \models \varphi$.

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

We first prove a small model property of the implication problem (Section VI-A). Capitalizing on the property, we develop a sequential exact algorithm SeqImp for implication checking (Section VI-B). We then parallelize SeqImp and develop a parallel scalable algorithm ParImp (Section VI-C).

A. A Small Model Property of GFD Implication

Recall that for traditional FDs over relations, the implication analysis is simple and takes linear time (cf. [10]). When it comes to GFDs, however, the story is more complicated.

Example 8: Consider a set $\Sigma = \{\varphi_{11}, \varphi_{12}\}$ of GFDs, where $\varphi_{11} = Q_8[\bar{x}](\emptyset \to x.A = 1)$, $\varphi_{12} = Q_9[\bar{x}](x.A = 1 \wedge y.B = 2 \to y.C = 2)$, and pattern Q_8 and Q_9 are shown in Fig. 2. Consider $\varphi_{13} = Q_7[\bar{x}](z.B = 2 \to z.C = 2)$, with Q_7 in Fig. 2. Then $\Sigma \models \varphi_{13}$. Indeed, for any graph G such that $G \models \Sigma$, and for any match $h(\bar{x})$ of Q_7 in G, $h(\bar{x})$ can be written as $h(x) \mapsto u_1$, $h(y) \mapsto u_2$, $h(z) \mapsto u_3$, $h(w) \mapsto u_4$, where u_i is in G for $1 \le i \le 4$. If $h(\bar{x}) \models z.B = 2$, then by enforcing φ_{11} and φ_{12} on $h(\bar{x})$, we have that $u_1.A = 1$, $u_3.B = 2$ and $u_3.C = 2$. Then $h(\bar{x}) \models z.C = 2$. Hence $G \models \varphi_{13}$. Note that φ_{13} is not implied by each of φ_{11} and φ_{12} alone. However, when φ_{11} and φ_{12} are put together, they can deduce the consequence $z.C = 2$ of φ_{13}.

Now consider $\varphi_{14} = Q_7[\bar{x}](x.A = 0 \rightarrow z.C = 2)$. Again one can verify that $\Sigma \models \varphi_{14}$. This is because for any graph G and any match $h(\bar{x})$ of Q_7 in G, if $G \models \Sigma$ then $h(\bar{x}) \not\models h(x).A = 0$, since φ_{11} enforces $h(x).A = 0$. That is, Q_7, $x.A = 0$ and Σ are "inconsistent" when put together. □

We tackle the implication problem also by proving a small model property. This is more involved than its counterpart for satisfiability. We first review a few notations of [2].

Canonical graphs. Consider $\varphi = Q[\bar{x}](X \rightarrow Y)$, where $Q = (V_Q, E_Q, L_Q)$. The *canonical graph* of φ is $G_Q^X = (V_Q, E_Q, L_Q, F_A^X)$, where F_A^X is defined as follows. For each node $x \in V_Q$ (*i.e.,* each $x \in \bar{x}$), (a) if $x.A = c$ is in X, then $F_A^X(x)$ has attribute A with $x.A = c$; (b) if $x.A = y.B$ is in X, then $F_A^X(x)$ has attributes A and B such that $x.A = y.B$; and moreover, (c) F_A^X is closed under the transitivity of equality, *i.e.,* if $x.A = y.B$ and $y.B = z.C$, then $x.A = z.C$; similarly if $x.A = c$ and $z.C = c$, then $x.A = z.C$.

We keep wildcard _ of Q in G_Q just like in G_Σ.

Along the same lines as Section IV-B, we define a (Σ, φ)-bounded population of G_Q^X for canonical graph G_Q^X of φ as a population of G_Q^X such that its size is in $O(|\Sigma| + |\varphi|)$.

A small model property. We show that to check whether $\Sigma \models \varphi$, it suffices to populate the canonical graph G_Q^X.

Theorem 3: *For any set* Σ *of* GFDs *and* GFD $\varphi = Q[\bar{x}](X \rightarrow Y)$, $\Sigma \models \varphi$ *iff for all* (Σ, φ)*-bounded populations* G *of* G_Q^X, *either (a)* $G \not\models \Sigma$, *or (b)* $G \models \Sigma$ *and* $G \models \varphi$. □

Proof: If $\Sigma \models \varphi$, then for all graphs G, if $G \models \Sigma$ then $G \models \varphi$. These graphs include (Σ, φ)-bounded populations of G_Q^X. From this it follows that conditions (a) and (b) hold.

Conversely, assume that $\Sigma \not\models \varphi$, *i.e.,* there exists a graph G such that $G \models \Sigma$ but $G \not\models \varphi$. We construct a (Σ, φ)-bounded population G' of G_Q^X such that $G' \models \Sigma$ but $G' \not\models \varphi$, violating conditions (a) and (b). The construction makes use of the "witness" of $G \not\models \varphi$ (the match of the pattern of φ in G that violates φ), and requires attribute value normalization as in the proof of Theorem 1, such that the total size of attributes in G are in $O(|\Sigma| + |\varphi|)$ (see [15] for details). □

Checking implication. To check whether $\Sigma \models \varphi$, Theorem 3 allows us to inspect (Σ, φ)-bounded populations G of G_Q^X only. However, it requires us to check all such small graphs, exponentially many in total. To further reduce the search space, we next prove a corollary of Theorem 3.

We first present some notations. Recall equivalence class Eq representing F_A^Σ. Given a GFD $\phi = Q'[\bar{x}'](X' \rightarrow Y')$ in Σ and a match h' of Q' in G_Q^X, Eq can be expanded by enforcing ϕ at h' with the two rules given in Section IV-C.

We refer to a list H of such pairs (h', ϕ) as a *partial enforcement of* Σ *on* G_Q^X. We use Eq_H to denote *the expansion of* Eq *by* H, by enforcing ϕ at h' one by one.

We say that Eq_H is *conflicting* if there exists $[x.A]_{\mathsf{Eq}_H}$ that includes distinct constants c and d. Intuitively, this means that the GFDs in H and Q, X are inconsistent.

Recall that $\varphi = Q[\bar{x}](X \rightarrow Y)$. We write $Y \subseteq \mathsf{Eq}_H$ if for any literal $u = v$, $v \in [u]_{\mathsf{Eq}_H}$, where $u = v$ is either $x.A = c$ or $x.A = y.B$. That is, the literal can be deduced from the equivalence relation Eq_H via the transitivity of equality.

Corollary 4: *For any set* Σ *of* GFDs *and* $\varphi = Q[\bar{x}](X \rightarrow Y)$, $\Sigma \models \varphi$ *iff there exists a partial enforcement* H *of* Σ *on* G_Q^X *such that either* Eq_H *is conflicting, or* $Y \subseteq \mathsf{Eq}_H$. □

The two cases of Corollary 4 are illustrated in Example 8.

Proof: If $\Sigma \models \varphi$, then such an H exists by Theorem 3, since each Eq_H is a (Σ, φ)-bounded population of G_Q^X.

Conversely, we show the following by induction on the length of H. (1) If Eq_H is conflicting, then for all (Σ, φ)-bounded populations G of G_Q^X, $G \not\models \Sigma$. (2) If $Y \subseteq \mathsf{Eq}_H$, then for all (Σ, φ)-bounded populations G of G_Q^X, if $G \models \Sigma$, then $G \models \varphi$. From this and Theorem 3 it follows that $\Sigma \models \varphi$. □

Corollary 4 allows us to check $\Sigma \models \varphi$ by selectively inspecting H, instead of enumerating all (Σ, φ)-bounded populations. Leveraging Corollary 4, we verify the following along the same lines as the proof of Corollary 2.

Corollary 5: *The* GFD *implication problem is in* NP. □

B. A Sequential Algorithm for Implication

Capitalizing on Corollary 4, we develop an exact sequential algorithm for checking GFD implication.

Algorithm. The algorithm, denoted by SeqImp, takes as input a set Σ of GFDs and another GFD φ. It returns true if $\Sigma \models \varphi$, and false otherwise. Let $\varphi = Q[\bar{x}](X \rightarrow Y)$. Similar to SeqSat for satisfiability checking (Section IV-C), algorithm SeqImp enforces GFDs of Σ on matches of Q in the canonical graph G_Q^X one by one, and terminates as soon as $\Sigma \models \varphi$ can be decided. It has the following subtle differences from SeqImp.

(a) In contrast to SeqImp that starts with Eq initially empty, SeqImp uses Eq_H to represent partial enforcement, initialized as Eq_X, the (nonempty) equivalence relation encoding F_A^X.

(b) SeqImp terminates with true when either (i) Eq_H is conflicting, or (ii) $Y \subseteq \mathsf{Eq}_H$. It terminates with false when all GFDs are processed, if neither conflict is detected nor $Y \not\subseteq \mathsf{Eq}_H$ in the entire process, concluding that $\Sigma \not\models \varphi$.

Example 9: Recall Σ and φ_{13} from Example 8. The canonical graph of φ_{13} is Q_7 of Fig. 2 with $F_A^X = \{z.B = 2\}$. Given these, SeqImp initializes Eq_H with $[z.B]_{\mathsf{Eq}} = \{2\}$. It expands Eq_H at (a) a match $h(\bar{x})$ of Q_8: $x_8 \mapsto x$ and $y_8 \mapsto y$, and (b) a match $h'(\bar{x})$ of Q_9: $x_9 \mapsto x$, $y_9 \mapsto z$, where x_i and y_i denote variables x and y, respectively, in Q_i for $i \in [8, 9]$. After enforcing φ_{11} and φ_{12} at $h(\bar{x})$ and $h'(\bar{x})$, respectively, it finds that $(z.C = 2) \subseteq \mathsf{Eq}_H$, and terminates with true.

Now for φ_{14}. SeqImp starts with $F_A^{X'} = \{x.A = 0\}$. After enforcing φ_{11} at $h(\bar{x})$, it adds "1" to $[x.A]_{\mathsf{Eq}}$, a conflict with $[x.A]_{\mathsf{Eq}} = \{0\}$. Hence SeqImp returns true and terminates. □

Analysis. The correctness of SeqImp follows from Corollary 4. Its complexity is dominated by generating matches of graph patterns in Σ, while $Y \subseteq \mathsf{Eq}_H$ and conflicts in Eq_H can be checked efficiently. In particular, the equivalence relation Eq_H

can be computed in linear time with index. Moreover, one can verify that the length of Eq_H is bounded by $|Q|\cdot|\Sigma|$ (see [15]).

C. Checking Implication in Parallel

We next develop algorithm ParImp that is parallel scalable relative to SeqImp. Hence ParImp is capable of dealing with large Σ by adding processors as needed.

Algorithm. ParImp works with a coordinator S_c and p workers $(P_1, \ldots, P_p)$, like ParSat. It first constructs the canonical graph G_Q^X of φ, initializes Eq_H as Eq_X (Section VI-B), and replicates G_Q^X and Eq_H at each worker.

The idea is to expand Eq_H in parallel, by distributing work units across p workers. A work unit $(Q_\phi[z], \phi)$ is defined in the same way as in Section V, for GFDs $\phi = Q_\phi[\bar{x}_\phi](X_\phi \to Y_\phi)$ in Σ at pivots z in G_Q^X. The work units are organized in a priority queue W as before, based on a revised notion of dependency graph (see below). Algorithm ParImp dynamically assigns work units of W to workers, starting with the ones with the highest priority, in small batches. Workers process their assigned work units in parallel, broadcast their local Eq_H expansions to other workers, and send flags to S_c. The process proceeds until (a) either at a partial enforcement H of G at some worker, Eq_H has conflict or $Y \subseteq \mathsf{Eq}_H$, or (b) all work units in W have been examined. It returns true in case (a), and false in case (b), by Corollary 4.

ParImp employs the same dynamic workload assignment and unit splitting strategies of ParSat to handle stragglers. It also supports a combination of data partitioned parallelism and pipelined parallelism. It differs from ParSat in the following.

(a) Dependency graph. ParImp deduces a topological order on W also based on the dependency graph of work units. The only difference is that a unit $(Q_\phi[z], \phi)$ is associated with the highest priority if $\phi = Q_\phi[\bar{x}_\phi](X_\phi \to Y_\phi)$ and X subsumes X_ϕ, *i.e.,* each literal in X_ϕ can be deduced from Eq_X.

(b) Early termination. Each worker P_i sends flag f_i^c to coordinator if either (i) a conflict is detected in its local copy of Eq_H, or (ii) $Y \subseteq \mathsf{Eq}_H$. Upon receiving f_i^c, algorithm ParImp terminates immediately with true, regardless of what P_i is.

Example 10: Assume a coordinator S_c and two workers P_1 and P_2. Given Σ and φ_{13} of Example 9, ParImp creates the canonical graph of φ_{13} and replicates it at P_1 and P_2, where $\mathsf{Eq}_{(H,i)}$ at P_i is initialized as Eq_H for $i \in [1, 2]$. It creates a priority queue $W = [w_1 = (Q_8[x], \varphi_{11}), w_2 = (Q_9[x], \varphi_{12})]$. Then S_c sends w_1 to P_1 and w_2 to P_2. After P_1 enforces φ_{11} on match $h(\bar{x})$ given in Example 9, it sends changes $\Delta\mathsf{Eq}_{(H,1)} = \{[x.A]_{\mathsf{Eq}_{(H,1)}} = \{1\}, [z.B]_{\mathsf{Eq}_{(H,1)}} = \{2\}\}$ to P_2. Worker P_2 enforces φ_{12} on match $h'(\bar{x})$ (Example 9). By incorporating changes from P_1, P_2 adds $z.C$ to $[z.B]_{\mathsf{Eq}_{(H,2)}}$, which contains value 2. As a result, $(z.C = 2) \subseteq \mathsf{Eq}_{(H,2)}$. Hence it sends f_2^c to S_c and ParImp terminates with true.

Now consider φ_{14} instead of φ_{13}. ParImp creates priority queue $W = [w_1 = (Q_9[x], \varphi_{12}), w_2 = (Q_8[x], \varphi_{11})]$. Note that W is different from the queue for φ_{13}, since the initial Eq_H includes $[x.A]_{\mathsf{Eq}_H} = \{0\}$. Coordinator S_c sends w_1 to P_1 and w_2 to P_2. When P_2 enforces φ_{11} on match $h'(\bar{x})$, it adds "1" to $[x.A]_{\mathsf{Eq}_{(H,2)}}$, but "0" is already in $[x.A]_{\mathsf{Eq}_{(H,2)}}$. Thus P_2 sends f_2^c to S_c and ParImp stops with true. □

algorithms	*DBpedia*	*YAGO2*	*Pokec*
SeqSat	1728	1341	2475
SeqImp	728	644	1355
ChaseImp$_{\mathsf{RDF}}$	1026	987	1907

Fig. 5: Sequential running time on real-life GFDs

Analysis. The correctness of ParImp is assured by Corollary 4 and monotonic expansion of Eq_H. ParImp is parallel scalable relative to SeqImp by dynamic workload balancing and unit splitting. Formally, one can show that ParSat takes $O(\frac{t(|\Sigma|,|\varphi|)}{p})$ time with p workers, where $t(|\Sigma|, |\varphi|)$ is the cost of SeqImp, by induction on the number of work units.

VII. Experimental Study

Using GFDs on real-life and synthetic graphs, we conducted four sets of experiments to evaluate the efficiency and scalability of our algorithms. We evaluated the impact of (1) the number of processors used in the parallel algorithms, (2) the number of GFDs, (3) the complexity (patterns and literals) of GFDs, and (4) TTL for work unit splitting (see [15] for more).

Experimental setting. We used three sets of GFDs discovered by the algorithm of [23] from real-life graphs. (a) *DBpedia*, a knowledge graph [24] with 1.72 million entities of 200 types, and 31 million links of 160 types; (b) *YAGO2*, an extension of knowledge base YAGO [25] with 1.99 million nodes of 13 types, and 5.65 million links of 36 types; and (c) *Pokec* [26], a social graph with 1.63 million nodes of 269 types and 30.6 million edges of 11 types. We mined more than 8000, 6000 and 10000 frequent GFDs from *DBpedia*, *YAGO2* and *Pokec*, respectively, *e.g.,* φ_1–φ_3 of Example 1 from *DBpedia*.

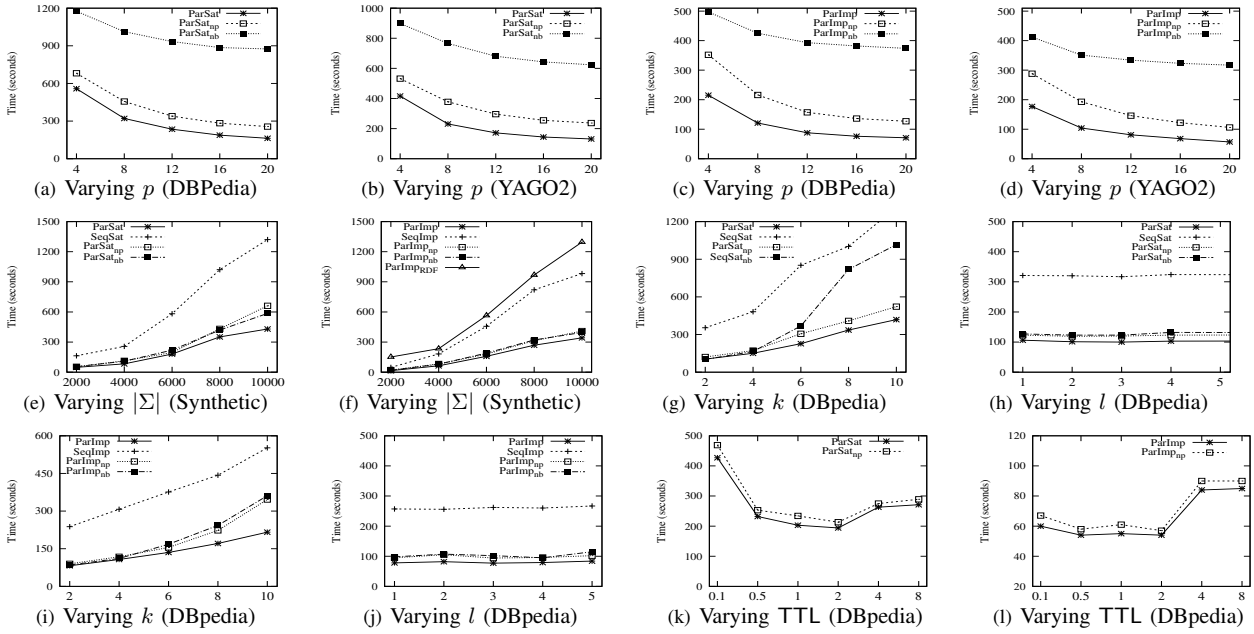
Each set Σ of GFDs discovered from graph G has a model, *i.e.,* G itself. Hence to test satisfiability, we expanded Σ by adding up to 10 GFDs randomly generated using attributes and edges from G (see GFD generator below), also denoted as Σ.

GFD generator. As no existing benchmarks are able to generate GFDs (see Section VIII), we also developed a generator to produce sets Σ of GFDs $Q[\bar{x}](X \to Y)$, controlled by (a) $|\Sigma|$ (up to 10000); (b) the maximum number k of nodes in pattern Q, up to 6; and (c) the maximum number l of literals in X and Y, up to 5. We controlled k and l to evaluate the impact of the complexity of GFDs (see Exp-3).

Algorithms. We implemented the following, all in Java.

(1) Satisfiability: (a) sequential SeqSat (Section IV-C), (b) parallel ParSat (Section V). To test the effectiveness of the optimization strategies, we also implemented (c) ParSat$_{np}$, a variant of ParSat without pipelining, *i.e.,* for each work unit $w = (Q[\bar{z}], \varphi)$, it first enumerates the matches of $Q[\bar{z}]$, and then for each match $h(\bar{x})$, it enforces φ at $h(\bar{x})$; and (d) ParSat$_{nb}$, a variant of ParSat without work unit splitting.

(2) Implication: (a) sequential SeqImp (Section VI-B), (b) parallel ParImp (Section VI-C), (c) a chase-based sequential algorithm ChaseImp$_{\mathsf{RDF}}$ for FD implication following [5], by



(a) Varying p (DBPedia) (b) Varying p (YAGO2) (c) Varying p (DBPedia) (d) Varying p (YAGO2)

(e) Varying $|\Sigma|$ (Synthetic) (f) Varying $|\Sigma|$ (Synthetic) (g) Varying k (DBpedia) (h) Varying l (DBpedia)

(i) Varying k (DBpedia) (j) Varying l (DBpedia) (k) Varying TTL (DBpedia) (l) Varying TTL (DBpedia)

Fig. 6: Performance evaluation of GFD reasoning

representing the triple patterns in the FDs of [5] as graphs, (d) ParImp$_{np}$, a variant of ParImp without pipelining, and (e) ParImp$_{nb}$, a variant of ParImp without work unit splitting.

All the algorithms sort GFDs with dependency graphs, including sequential SeqSat and SeqImp, except ChaseImp$_{\mathsf{RDF}}$.

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

Experimental results. We next report our findings. We first evaluated sequential algorithms SeqSat, SeqImp and ChaseImp$_{\mathsf{RDF}}$ using real-life GFDs. As shown in Fig. 5, (a) SeqSat and SeqImp perform reasonably well, *e.g.,* they take 1728 and 728 seconds on GFDs from *DBpedia*, respectively; and (b) SeqImp outperforms ChaseImp$_{\mathsf{RDF}}$ by 1.4, 1.5 and 1.4 times on GFDs from *DBpedia*, *YAGO2* and *Pokec*, respectively. We find that implementations of the chase [2] are much slower than SeqSat and SeqImp (hence not shown). These justify the effectiveness of our algorithms.

Exp-1: Parallel scalability. We then evaluated the parallel scalability of algorithms ParSat, ParImp and their corresponding variants. Fixing TTL $= 2$ seconds and varying the number p of processors from 4 to 20, we report the results on real-life GFDs on *DBpedia* and *YAGO2* in Figures 6(a)–6(d), respectively. The results on *Pokec* are consistent (not shown).

(1) Satisfiability. As shown in Figures 6(a) and 6(b), (a) ParSat is 3.7 times and 3.2 faster on average when p increases from 4 to 20 on *DBpedia* and *YAGO2*, respectively. (b) Parallelization substantially speeds up satisfiability checking: ParSat is 15 and 10.2 times faster than SeqSat, respectively, when $p = 20$. (c) ParSat is feasible in practice. It takes 163 and 131 seconds, respectively, when p = 20. (d) ParSat outperforms ParSat$_{nb}$ (resp. ParSat$_{np}$) by 3.8 and 3.7 (resp. 1.4 and 1.6) times on average, and in particular, 5.3 and 4.8 (resp. 1.5 and 1.6) times when p = 20, on *DBpedia* and *YAGO2*, respectively. These verify the effectiveness of our optimization strategies.

(2) Implication. As shown in Figures 6(c) and 6(d) using GFDs from *DBpedia* and *YAGO2*, respectively, ParImp is (a) 3 and 3.1 times faster when p varies from 4 to 20. (b) It does substantially better than SeqImp, by 10.3 and 11.3 times when $p = 20$, and (c) it outperforms ParImp$_{nb}$ (resp. ParImp$_{np}$) by 4.1 and 4.1 (resp. 1.7 and 1.8) times on average, respectively.

Note that SeqImp and ParImp are faster than SeqSat and ParSat, respectively, as the canonical graph G_Q^X for implication is smaller than G_Σ for satisfiability (Sections IV and VI).

Exp-2: Scalability with $|\Sigma|$. Fixing $k = 6$ and $l = 5$, we evaluated the scalability of the algorithms by varying the number $|\Sigma|$ of synthetic GFDs in Σ from 2000 to 10000. For parallel algorithms, we used $p = 4$ processors.

(1) Satisfiability. As shown in Fig. 6(e), (a) the larger $|\Sigma|$ is, the longer all algorithms take, as expected; (b) nonetheless, ParSat outperforms SeqSat by 3.14 times on average; (c) ParSat is on average 1.24 and 1.26 times faster than ParSat$_{nb}$ and ParSat$_{np}$, respectively; the improvement over ParSat$_{nb}$ is not as significant as in Exp-1 since k is fixed to be 6, and work unit splitting is more effective on GFDs with larger k; (d) SeqSat and ParSat are insensitive to the growth of $|\Sigma|$ when Σ is not satisfiable (not shown), justifying the effectiveness of our early termination strategy; and (d) SeqSat and ParSat take 1321 and 430 seconds when $|\Sigma|$ = 10000, respectively; *i.e.,* the parallel cost is reasonable when $p = 4$.

(2) Implication. As shown in Fig. 6(f), the implication algorithms behave consistently with their satisfiability counter-

parts. All algorithms take longer on larger Σ, while SeqImp and ParImp are less sensitive to $|\Sigma|$ when $\Sigma \models \varphi$, due to early termination. Moreover, (a) ParImp is 3.1 and 4.8 times faster than SeqImp and ChaseImp$_{\mathsf{RDF}}$ on average, respectively, (b) ParImp outperforms ParImp$_{nb}$ and ParImp$_{np}$ by 1.3 and 1.2 times on average, respectively, and (c) SeqImp and ParImp take 982 and 342 seconds when $|\Sigma|$ = 10000.

Exp-3: Impact of complexity of GFDs. We next evaluated the impact of k and l on reasoning about GFDs. We used synthetic GFDs generated with seed patterns, frequent edges and active attributes from *DBpedia* (the results on *YAGO2* and *Pokec* are consistent and are not shown). We fixed $|\Sigma| = 5000$ and $p = 4$ when testing the parallel algorithms.

(1) Varying k. Fixing $l = 3$, we varied k from 4 to 10. The results are reported in Fig. 6(g) for satisfiability, and Fig. 6(i) for implication. We can see the following. (a) The larger k is, the longer all algorithms take, as expected. (b) The larger k is, the more effective our optimization strategies are. (c) When $k = 10$, on average ParSat and ParImp take 398 and 201 seconds, and SeqSat and SeqImp take 1253 and 538 seconds, respectively. Thus the algorithms are able to deal with GFDs with fairly large patterns, especially the parallel algorithms.

(2) Varying l. Fixing $k = 5$, we varied l from 1 to 5. As shown in Figures 6(h) and 6(j) for satisfiability and implication, respectively, (a) all algorithms are not very sensitive to l. While more literals take longer to process, they may also make the process terminate earlier. (b) ParSat and ParImp perform the best in all cases. (c) Our algorithms work well: when $l = 5$, ParSat and ParImp take 108 and 77 seconds on average, and SeqSat and SeqImp take 351 and 262 seconds, respectively.

Exp-4: Impact of straggler parameter TTL. Fixing $p = 4$, we evaluated the impact of work unit splitting by varying TTL from 0.1s to 8s. The results are reported in Fig. 6(k) for satisfiability and Fig. 6(l) for implication, using GFDs from *DBpedia* (the results on *YAGO2* and *Pokec* are consistent; hence not shown). Observe the following. (a) When TTL gets larger, on one hand, the workload is less balanced due to a higher bar on stragglers; on the other hand, less communication cost is incurred since less work units are split. (b) The cost is no longer reduced when TTL reaches a point. The optimal value of TTL is 2 in both Figures 6(k) and and 6(l).

Summary. From the experiments we find the following. (a) Our sequential algorithms work reasonably well on real-life GFDs: SeqSat and SeqImp take 1848 and 909 seconds on average, respectively. (b) Parallelization substantially improves the performance. When $p = 20$, ParSat and ParImp take 167 and 76 seconds on average on real-life GFDs, respectively, and are hence feasible in practice. (c) Better still, ParSat and ParImp are parallel scalable. On real-life GFDs, they are 3.4 and 3.6 times faster on average, respectively, when p varies from 4 to 20. (d) Our optimization strategies are effective. The combination of data-partitioned and pipelined parallelism improves the performance of ParSat and ParImp by 1.5 and 1.6 times, and work unit splitting speeds up 3.8 and 4.1 times, respectively. (e) Our algorithms are able to reason about large sets Σ of GFDs with complex patterns and literals. For instance, when $|\Sigma| = 10000$, $k = 6$ and $l = 5$, ParSat and ParImp take 430 and 342 seconds, respectively, when $p = 4$.

VIII. Related Work

Graph functional dependencies. While a "standard form" of FDs for graphs is not yet in place, several proposals have been published for RDF. Based on triple patterns and homomorphism, a class of FDs was proposed in [3], [4]. By composing properties in RDF, [6] defines FDs with path patterns, which were extended in [8] to support constants. The FDs of [7] support tree patterns. A form of keys was defined in [27]. These proposals are for RDF, and do not support either cyclic graph patterns [6]–[8] or constants [3], [4].

We adopt GFDs of [2] since the GFDs are defined on general graphs beyond RDF, and support (possibly cyclic) graph patterns and constant literals as in CFDs [16]. This form of GFDs was introduced in [1], via subgraph isomorphism for pattern matching. As noted in [2], it is more natural to interpret GFDs and keys under the homomorphism semantics.

Static analyses. Over relations, the satisfiability and implication problems are in $O(1)$ and linear time for FDs (cf. [10]), and are NP-complete and coNP-complete for CFDs [16], respectively. Over RDF, chase-based implication algorithms are provided for the FDs of [4], [5]. However, the exact complexity of the satisfiability and implication problems for the dependencies of [3]–[8] remains open.

Over graphs, the problems are shown coNP-complete and NP-complete for GFDs of [2], respectively. The upper bounds are verified by characterizing GFD satisfiability and implication in terms of an extension of the chase [28] to graphs [2]. A sound and complete axiom system was given in [2] for finite implication of GEDs, an extension of GFDs.

This work extends [2] from the practical side. (a) We establish small model properties for GFD satisfiability and implication. We show that the static analyses can be conducted by inspecting only graphs with size and structure determined by the patterns in the given GFDs. These yield upper bound proofs different from [2], without using the chase. (b) We develop (parallel) algorithms to check GFD satisfiability and implication. We propose a combination of pipelined parallelism and data partitioned parallelism to speed up the process, and a combination of dynamic work assignment and unit splitting to reduce stragglers. These were not studied in [1], [2].

The chase. Related also the chase [28], a classical tool in the relational dependency theory. The chase has been employed in data exchange [29], [30], data repairing [31] and query rewriting [32], with relational tuple-generating dependencies (TGDs) and equality generating dependencies (EGD; see [33] for a survey). As remarked earlier, the chase has also been studied for FDs on RDF [4], [5] and for GEDs [2].

We have only empirically compared with the chase-based method for FD implication on RDF [5] (Section VII), for the

following reasons. (a) No chase algorithms are in place for FD satisfiability checking on graphs, except a theoretical method of [2]. The method of [2] is not very practical since it nondeterministically applies GEDs and requires graph coercion. (b) GFDs are not expressible as relational EGDs because GFDs support wildcard '_' and "generate new attributes". To see this, consider $\varphi = Q[x](\emptyset \rightarrow x.A = x.A)$, where pattern Q consists of a single node x labeled '_'. To enforce φ on a graph G, each node must have attribute A, which is not warranted since unlike relations, G may not have a schema. As another evidence, the satisfiability problem is coNP-complete for GFDs, but is NP-complete for EGDs [34]. (c) The chase with TGDs is generally undecidable [32]. While some special cases have been studied, *e.g.,* oblivious terminating TGDs and EGDs [35], their satisfiability problem is open. It is not clear whether GFDs can be expressed in the special forms, and even so, what results GFDs can inherit from them. (d) As observed in [36], native graph techniques perform "significantly better than relational databases" on graphs. Indeed, we make use of the data locality of graph homomorphism to check GFDs (see Section V), which is not offered by the relational chase. (e) We develop parallel techniques to, *e.g.,* reduce stragglers, which were not studied by the prior work on the chase. These said, the algorithms of this paper could be regarded as a parallel implementation of the theoretical chase method of [2].

Parallel reasoning. We are not aware of any prior parallel algorithms for reasoning about graph dependencies, not to mention algorithms with parallel scalability. There are, however, several methods to deal with stragglers. Speculative execution [37] prioritizes slowest tasks. Work stealing [38] and shedding [39] adaptively re-balance work queues among workers. Fine grained partition strategy [40] reduces performance variation by restricting the interdependence among workers. In contrast, we explore a new method, by dynamic straggler (work unit) splitting and dynamic work unit assignment.

IX. Conclusion

We have shown the small model properties of the satisfiability and implication problems for GFDs. We have developed sequential and parallel algorithms for reasoning about GFDs, and a set of new parallel reasoning techniques. Our experimental study has verified the scalability and efficiency of the algorithms. The work is among the first effort to reason about dependencies in parallel, with parallel scalability.

We are currently extending the algorithms to reason about GEDs [2] with recursively-defined keys, and their extensions with built-in predicates ($\leq, <, \geq, >, \neq$) and disjunction.

References

[1] W. Fan, Y. Wu, and J. Xu, "Functional dependencies for graphs," in *SIGMOD*, 2016.

[2] W. Fan and P. Lu, "Dependencies for graphs," in *PODS*, 2017.

[3] A. Cortés-Calabuig and J. Paredaens, "Semantics of constraints in RDFS," in *AMW*, 2012, pp. 75–90.

[4] W. Akhtar, A. Cortés-Calabuig, and J. Paredaens, "Constraints in RDF," in *SDKB*, 2010, pp. 23–39.

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

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

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

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

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

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

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

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

[13] C. P. Kruskal, L. Rudolph, and M. Snir, "A complexity theory of efficient parallel algorithms," *TCS*, vol. 71, no. 1, pp. 95–132, 1990.

[14] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[15] Full version, *http://homepages.inf.ed.ac.uk/wenfei/RGFD.pdf*.

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

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *TPAMI*, vol. 26, no. 10, pp. 1367–1372, 2004.

[18] 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.

[19] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov, "Collecting and analyzing data from e-government Facebook pages," in *ICT Innovations*, 2014.

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

[21] W. Fan, X. Wang, Y. Wu, and J. Xu, "Association rules with graph patterns," *PVLDB*, vol. 8, no. 12, pp. 1502–1513, 2015.

[22] M. R. Henzinger, T. Henzinger, and P. Kopke, "Computing simulations on finite and infinite graphs," in *FOCS*, 1995.

[23] W. Fan, X. Liu, P. Lu, Y. Wu, and J. Xu, "Discovering graph functional dependencies," Submitted for publication, 2017.

[24] DBpedia, *http://wiki.dbpedia.org/Datasets*.

[25] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW*, 2007.

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

[27] W. Fan, Z. Fan, C. Tian, and X. L. Dong, "Keys for graphs," *PVLDB*, vol. 8, no. 12, pp. 1590–1601, 2015.

[28] F. Sadri and J. D. Ullman, "The interaction between functional dependencies and template dependencies," in *SIGMOD*, 1980, pp. 45–51.

[29] B. Marnette, G. Mecca, and P. Papotti, "Scalable data exchange with functional dependencies," *PVLDB*, vol. 3, no. 1, pp. 105–116, 2010.

[30] A. Bonifati, I. Ileana, and M. Linardi, "Functional dependencies unleashed for scalable data exchange," in *PSSDBM*, 2016, pp. 2:1–2:12.

[31] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, "The LLUNATIC data-cleaning framework," *PVLDB*, vol. 6, no. 9, pp. 625–636, 2013.

[32] A. Deutsch, A. Nash, and J. B. Remmel, "The chase revisited," in *PODS*, 2008, pp. 149–158.

[33] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura, "Benchmarking the chase," in *PODS*, 2017.

[34] M. H. Graham, A. O. Mendelzon, and M. Y. Vardi, "Notions of dependency satisfaction," *JACM*, vol. 33, no. 1, pp. 105–129, 1986.

[35] B. Marnette, "Generalized schema-mappings: From termination to tractability," in *PODS*, 2009, pp. 13–22.

[36] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *ACM SE*, 2010, p. 42.

[37] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.

[38] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *PPoPP*, 2013.

[39] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *IPDPS*, 2007.

[40] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri, "Ontological pathfinding," in *SIGMOD*, 2016.