Deducing Certain Fixes to Graphs

Wenfei Fan[1,2,4] Ping Lu[2] Chao Tian[3] Jingren Zhou[3]
[1]University of Edinburgh [2]Beihang University [3]Alibaba Group [4]SICS, Shenzhen University
wenfei@inf.ed.ac.uk, luping@buaa.edu.cn, {tianchao.tc, jingren.zhou}@alibaba-inc.com

ABSTRACT

This paper proposes to deduce *certain fixes* to graphs G based on data quality rules Σ and ground truth Γ (*i.e.,* validated attribute values and entity matches). We fix errors detected by Σ in G such that the fixes are assured correct as long as Σ and Γ are correct. We deduce certain fixes in two paradigms. (a) We interact with users and "incrementally" fix errors online. Whenever users pick a small set V_0 of nodes in G, we fix all errors pertaining to V_0 and accumulate ground truth in the process. (b) Based on accumulated Γ, we repair the entire graph G offline; while this may not correct all errors in G, all fixes are guaranteed certain.

We develop techniques for deducing certain fixes. (1) We define data quality rules to support conditional functional dependencies, recursively defined keys and negative rules on graphs, such that we can deduce fixes by combining data repairing and object identification. (2) We show that deducing certain fixes is Church-Rosser, *i.e.,* the deduction converges at the same fixes regardless of the order of rules applied. (3) We establish the complexity of three fundamental problems associated with certain fixes. (4) We provide (parallel) algorithms for deducing certain fixes online and offline, and guarantee to reduce running time when given more processors. Using real-life and synthetic data, we experimentally verify the effectiveness and scalability of our methods.

PVLDB Reference Format:
Wenfei Fan, Ping Lu, Chao Tian, Jingren Zhou. Deducing Certain Fixes to Graphs. *PVLDB*, 12(7): 752-765, 2019.
DOI: https://doi.org/10.14778/3317315.3317318

1. INTRODUCTION

It is common to find semantic inconsistencies in real-life graphs such as knowledge bases and e-commerce networks. A host of graph dependencies have been proposed to catch the inconsistencies as violations of the dependencies [14, 3, 24, 51, 11, 30, 26]. These dependencies can detect common errors. However, they do not tell us how to fix the errors.

Proceedings of the VLDB Endowment, Vol. 12, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3317315.3317318

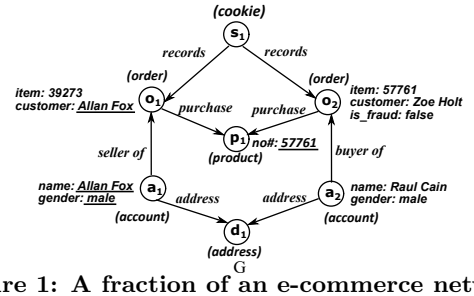


Figure 1: A fraction of an e-commerce network

Example 1: A fraction of an e-commerce network is depicted as graph G in Fig. 1, where each node denotes an entity carrying a tuple of attributes. It shows that the product p_1 is purchased in orders o_1 and o_2 with seller account a_1 and buyer account a_2, respectively. The network often contains errors and has information missing. A dependency φ_1 is: if a product is purchased in an order, then its attribute no# must be the same as the attribute item of the order.

Dependency φ_1 detects an *inconsistency pertaining to* product p_1, between its no# (57761) and the attribute item (39273) of order o_1. However, it does not tell us which attribute is wrong and to what value it should be updated. Worse still, incorrect fixes may even introduce new errors, *e.g.,* changing p_1.no# to 39273 by taking the value of o_1.item adds a new violation of φ_1 by product p_1 and order o_2. □

Is there a systematic method to fix errors in graphs detected by dependencies such that the fixes are justified? To the best of our knowledge, no prior work has studied how to clean graphs with correctness justification.

To answer the question, we model certain fixes relative to data quality rules Σ and ground truth Γ. Here Γ consists of entity matches and attribute values that have been validated by users, domain experts or crowd-sourcing, and are accumulated over time. A fix is *certain* if it is a logical consequence of Σ and Γ, *i.e.,* it is assured correct if Σ and Γ are validated. Moreover, the deduction of certain fixes should be *Church-Rosser*, *i.e.,* the process converges at the same certain fixes regardless of the order of rules applied.

We compute certain fixes to a graph G in two settings.

Online mode. Users iteratively pick a small set V_0 of vertices in G, *i.e.,* entities of their interest, *e.g.,* items to search or terms updated. We correct *all errors* pertaining to V_0 and detected by Σ, by deducing certain fixes from Γ and Σ. The process may ask users to check ground truth if necessary; it expands Γ with newly validated ground truth and fixes.

Offline mode. Employing accumulated ground truth Γ, it fixes errors in the entire G without user interaction. It *may not* fix all errors in G when Γ is not inclusive, but it guarantees that *each and every fix deduced is certain.*

Both modes are practical paradigms for repairing graphs [52, 12]. The online mode interacts with users and "incrementally" repairs graphs; it accumulates ground truth continuously. We repair the entire graph offline periodically, once Γ accumulates sufficient new ground truth.

Example 2: A network analyst suspects that the orders of products are "fraudulently" created, *e.g.,* the sellers of p_1 purchase it themselves to increase merchandise popularity, and requests to check $V_0 = \{p_1\}$. To verify this, the inconsistency in Example 1 has to be *correctly* fixed in the first place. The analyst may inspect and confirm that p_1.no# is accurate as ground truth. Thus both orders o_1 and o_2 must carry an attribute item of 57761 by rule φ_1, which are indeed *certain* fixes and expand ground truth. As will be seen later, these further help identify objects and repair other attributes, *e.g.,* o_1 and o_2 should refer to the same order since they share the same attribute item and are recorded in the same system cookie. It proceeds until all the errors related to p_1 are fixed and all the fraudulent orders of p_1 are caught, by combining data repairing and object identification.

Analysts repeatedly inspect entities and get "local" fixes in the online mode. Employing ground truth Γ accumulated in this way, we can improve the quality of the entire G periodically offline, by deducing certain fixes to all errors that are "covered" by Γ, without user involvement. □

Contributions & organization. This paper is a step towards effective methods to deduce certain fixes to graphs.

(1) Rules (Section 2). We introduce a class of graph quality rules for graphs, referred to as GQRs, such that we can simultaneously (a) repair data, *i.e.,* fix attribute values, by using graph functional dependencies (GFDs [26]) along the same lines as conditional functional dependencies (CFDs [19]) for cleaning relations, (b) identify objects, *i.e.,* determine whether two vertices in a graph refer to the same entity, by using recursively defined keys [16], and moreover, (c) deduce entities that do not match, to reduce false positives, with a form of forbidding constraints (FCs).

(2) Certain fixes (Section 3). At the conceptual level, we model deducing certain fixes as chasing graph G with a set Σ of GQRs and a block Γ of ground truth, by extending the chase on relations (cf. [2]). It integrates data repairing and object identification in a single process. We show that the chase is Church-Rosser, *i.e.,* it guarantees to converge at the same certain fixes no matter how the GQRs are applied.

(3) Fundamental problems (Section 4). We settle three problems for deciding whether Σ and Γ are consistent, whether a fix can be deduced from Σ and Γ, and whether all errors pertaining to a given set V_0 of nodes can be fixed by Σ and Γ. We establish matching bounds for their combined and data complexity, from PTIME (polynomial time), coNP-complete and NP-complete to $\mathsf{P}_{||}^{\mathsf{NP}}$-complete.

(4) Practical algorithms (Sections 5-6). We develop algorithms for deducing certain fixes to graphs in the two modes.

(a) Online mode (Section 5). We give a sequential (single-machine) algorithm to fix *all* the errors pertaining to a *small* set V_0 of vertices of users' interest. While our chase can deduce certain fixes in theory, it is nondeterministic and typically incurs excessive and expensive graph homomorphism checking. We make it practical by proposing deterministic selection of GQRs to be enforced, and incremental expansion of certain fixes. These substantially reduce the cost.

(b) Offline mode (Section 6). We develop a parallel algorithm to clean the *entire graph* G, by using the ground truth Γ accumulated in the online mode or acquired from other high-quality sources, without user involvement. We show that the algorithm is *parallel scalable* [35] *relative to* the sequential algorithm of (a), *i.e.,* it takes $O(t(|G|, |\Sigma|, |\Gamma|)/p)$ time, where $t(|G|, |\Sigma|, |\Gamma|)$ is the cost of the algorithm of the online mode pertaining to all vertices in G, and p is the number of processors used. It aims to scale with real-life graphs G by adding more processors when G grows big.

(6) Experimental study (Section 7). Using real-life and synthetic graphs, we empirically verify the effectiveness and scalability of our methods. We find the following. (a) Deducing certain fixes online is effective: all errors pertaining to V_0 ($|V_0| \leq 12$) can be corrected in 3 user interaction rounds using a small number of rules and ground truth ($|\Sigma| \leq 200$ and $|\Gamma| \leq 10$); and each round needs 6s on average. (b) By interleaving data repairing and object identification, our methods outperform attribute repairing and entity resolution taken separately, by 37.6% and 44.7%, respectively. (c) Offline repairing with certainty is feasible in practice. It takes 877s to deduce certain fixes to entire graphs with up to 120 million nodes and edges using 16 processors, as opposed 8000s by other methods, and achieves F-measure of 0.924 on real-life graphs. (d) Our parallel algorithm is scalable: it is 3.9 times faster when 20 processors are used instead of 4.

To the best of our knowledge, this work is the first effort to deduce certain fixes to graph-structured data, from foundation to (parallel) algorithms. As demonstrated in [43] and confirmed by our industry collaborators, the quality rules play a vital role in industrial data cleaning tools. Our empirical study suggests that the approaches are promising for cleaning large-scale graphs in different paradigms.

Related work. We categorize related work as follows.

Graph dependencies. Various graph dependencies have been studied [11, 36, 2, 14, 3, 6, 51, 30, 26, 16, 44, 27]. We define GQRs by extending graph entity dependencies (GEDs) of [24] since GEDs can express GFDs [26] and keys [16], to support both data repairing and object identification; moreover, GEDs are defined for general graphs, not limited to RDF.

This work differs from [24] as follows. (1) We extend GEDs with negative rules (Section 2), which reduce false positives in object identification as observed in [22, 5]. (2) We settle fundamental problems for deducing certain fixes to graphs (Section 4), which are not studied in [24]. (3) We develop chase to find certain fixes, with syntax and semantics quite different from the chase of [24] for static analyses (see Section 3). We also parallelize the chase to make it scale with large graphs. (4) We provide practical techniques to clean graphs online and offline, a topic not studied in [24].

Data cleaning. Prior work has mostly focused on cleaning relations (see [18] for a survey). Heuristic methods are typically used to fix violations of relational dependencies [9, 19] with minimum cost; user feedback is adopted in GDR [50], FALCON [31] and DANCE [7]. Rule-based approaches and machine learning are combined in [40] to minimize changes and statistical distortion. Similarly, HoloClean [42] unifies qualitative and quantitative techniques to repair relations. There has also been a host of work on entity resolution [39, 47, 17, 49, 41, 5], possibly with user interaction [32]. Closer to this work are [23, 22], where [23] repairs relations with

certain fixes based on master data and CFDs, and [22] unifies entity resolution and data repairing to clean relations by using CFDs [19] and matching dependencies [17].

The work differs from [23, 22] in the following. (1) We use graph quality rules that cannot be expressed as CFDs [23] or matching dependencies [22]. (2) We compute certain fixes based on the chase with ground truth, while [23, 22] do not use chase, and assume the availability of master data. (3) We find certain fixes by means of both object identification and repairing, where [23] focuses on repairing only. (4) Some fundamental problems for deducing certain fixes to graphs have different complexity from repairing relations with CFDs (Section 4). (5) Our algorithm leverages the locality of graph homomorphism (Section 2), a property that is not explored by relational methods. (6) We develop parallel scalable algorithms, which are not studied in [23, 22].

On graphs, [44] enforces neighborhood constraints to restrict labels on adjacent nodes. Unsupervised clustering is studied for entity resolution in bibliographic datasets [8]. Based on a form of tuple generating dependencies (TGDs) and a class of contradiction detecting dependencies, [6] proposes a user-guided approach to resolving conflicts in knowledge bases. Closer to this work are [16, 12], where [16] uses keys for object identification, and [12] cleans graphs by applying a class of graph repairing rules (GRRs) recursively.

We make the first effort to fix graphs with certainty. As opposed to [6, 16, 8], we support both data repairing and object identification. We fix inconsistencies among attributes of different entities, beyond node labeling [44]. In contrast to [12], (1) we clean graphs online and offline with ground truth. (2) [12] offers no correctness guarantees and may even produce different results depending on the orders of GRRs applied. In contrast, our methods deduce certain fixes and are Church-Rosser. (3) Since GRRs of [12] subsume TGDs, their implication problem is undecidable, and chasing with GRRs may not terminate. We use GQRs to strike a balance between the types of errors fixed and the complexity.

Parallel data cleaning. Parallel algorithms have been studied for entity resolution [34, 4, 16, 13]. BigDansing [33] repairs relations on top of MapReduce. There has also been work on error detection in distributed relations [20] with CFDs, and in partitioned graphs [26] using GFDs.

None of these targets fixing inconsistencies in graphs. We present a simple workload partition strategy for deducing certain fixes to graphs, and guarantee parallel scalability that has not been accomplished by previous methods.

2. DATA QUALITY RULES

We now introduce GQRs as data quality rules for graphs.

2.1 Preliminaries

Assume three countably infinite sets Θ, Υ and U, denoting labels, attributes and constant values, respectively.

Graphs. We consider directed graphs $G = (V, E, L, F_A)$ with labeled nodes and edges, where (a) V is a finite set of nodes; each node $v \in V$ carries a label $L(v) \in \Theta$; (b) $E \subseteq V \times \Theta \times V$ is a finite set of edges, in which each $e = (v, \iota, v')$ denotes an edge from node v to v' labeled with ι; we write e as (v, v') and $L(e) = \iota$ if it is clear in the context; (c) each node $v \in V$ carries a tuple $F_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$ of *attributes* (properties), where $A_i \in \Upsilon$ and $a_i \in U$, written as $v.A_i = a_i$, and $A_i \neq A_j$ if $i \neq j$. In particular, each node v has a special attribute id denoting its *node identity*.

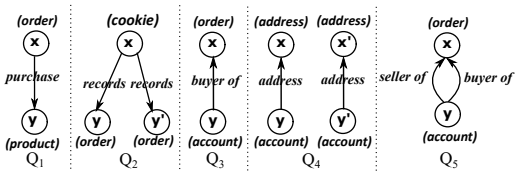


Figure 2: Graph patterns of GQRs

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 each node $u \in V_Q$ (resp. edge $e \in E_Q$); and (3) $\bar{x}$ denotes the nodes in V_Q as a list of distinct variables. Labels $L_Q(u)$ and $L_Q(e)$ are drawn from the alphabet Θ. Moreover, we allow wildcard '_' as a special label in Q.

Pattern matching. We say that a label ι *matches* ι', denoted by $\iota \asymp \iota'$, if either (a) both ι and ι' are in Θ and $\iota = \iota'$, or (b) $\iota' \in \Theta$ and ι is '_', *i.e.,* wildcard matches any label.

A *match* of pattern $Q[\bar{x}]$ in graph G is a homomorphism h from Q to G such that for each node $u \in V_Q$, $L_Q(u) \asymp L(h(u))$; and for each edge $e = (u, \iota, u')$ in Q, there exists an edge $e' = (h(u), \iota', h(u'))$ in G such that $\iota \asymp \iota'$.

We 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 all variables $x \in \bar{x}$. Intuitively, $h(\bar{x})$ is a list of entities identified by pattern Q.

2.2 Graph Quality Rules

A *graph quality rule* φ (GQR) is defined as $Q[\bar{x}](X \to Y)$, where $Q[\bar{x}]$ is a graph pattern, and X and Y are (possibly empty) sets of literals of $\bar{x}$. For $x, y \in \bar{x}$, a *literal* of $\bar{x}$ is

(a) $x.A = c$ or $x.A \neq c$, where c is a constant, and A is an attribute in Υ that is not id;
(b) $x.A = y.B$ or $x.A \neq y.B$, where A and B are attributes in Υ that are not id; or
(c) $x.\mathsf{id} = y.\mathsf{id}$ or $x.\mathsf{id} \neq y.\mathsf{id}$.

We refer to $x.A = c$, $x.A = y.B$ and $x.\mathsf{id} = y.\mathsf{id}$ as *equality literals*, and $x.A \neq c$, $x.A \neq y.B$ and $x.\mathsf{id} \neq y.\mathsf{id}$ as *inequality literals*. We refer to $Q[\bar{x}]$ and $X \to Y$ as the *pattern* and *attribute constraint* of GQR φ, respectively.

Intuitively, φ combines pattern Q to identify entities in a graph, and attribute constraint $X \to Y$ to be applied to the entities identified by Q. Literals like $x.A = c$ enforce constant bindings like in CFDs [19]. A literal of the form $x.\mathsf{id} = y.\mathsf{id}$ states that x and y denote the same node (entity).

GQRs extend the graph entity dependencies (GEDs of [24]) by supporting inequality literals, to reduce false positives in object identification, *i.e.,* entities that do not match.

Example 3: Consider the following simple GQRs defined in terms of graph patterns shown in Fig. 2.

(1) $\varphi_1 = Q_1[x, y](\emptyset \to x.\mathsf{item} = y.\mathsf{no}\#)$. It states that if an order x is to purchase product y, then the value of attribute x.item coincides with the value of attribute y.no#.

(2) $\varphi_2 = Q_2[x, y, y'](y.\mathsf{item} = y'.\mathsf{item} \to y.\mathsf{id} = y'.\mathsf{id})$. It says that the orders y and y' should denote the same node, *i.e.,* be identified, if they have the same attribute item and are recorded in the same system cookie (entity).

(3) $\varphi_3 = Q_3[x, y](\emptyset \to x.\mathsf{customer} = y.\mathsf{name})$. This GQR states that the value of attribute customer of order x must be the same as the value of attribute name of its buyer account y.

(4) $\varphi_4 = Q_4[x, x', y, y'](x.\mathsf{id} = x'.\mathsf{id}, y.\mathsf{name} = y'.\mathsf{name} \rightarrow y.\mathsf{id} = y'.\mathsf{id})$. It says that an account can be uniquely identified by its attribute name and the id of its address (entity).

(5) $\varphi_5 = Q_5[x, y](\emptyset \rightarrow x.\mathsf{is\_fraud} = \mathit{true})$. It says that order x is a fraud if it is bought and sold by the same account.

(6) $\varphi_6 = Q_6[x, y](x.\mathsf{gender} = \text{“male”}, y.\mathsf{gender} = \text{“female”} \rightarrow x.\mathsf{id} \neq y.\mathsf{id})$, where Q_6 is a pattern of two isolated account nodes x and y (not shown). It states that x and y cannot be the same account if they differ in the attribute gender. □

Semantics. Consider a GQR $Q[\bar{x}](X \rightarrow Y)$, a match $h(\bar{x})$ of Q in graph G, and a literal l of $\bar{x}$. We say that $h(\bar{x})$ *satisfies* l, denoted by $h(\bar{x}) \models l$, if (a) when l is $x.A = c$, *there exists* an attribute A at node $v = h(x)$, and $v.A = c$; (b) when l is $x.A = y.B$, nodes $v = h(x)$ and $v' = h(y)$ carry attributes A and B, respectively, and $v.A = v'.B$; and (c) when l is $x.\mathsf{id} = y.\mathsf{id}$, $h(x)$ and $h(y)$ denote the same node; thus *they have the same set of attributes*. Similarly we define $h(\bar{x}) \models l$ for an inequality literal l, *e.g.,* when l is $x.A \neq c$, $h(\bar{x}) \models l$ if $v = h(x)$ has an attribute A and $v.A \neq c$.

We write $h(\bar{x}) \models X$ if $h(\bar{x})$ satisfies *all* literals in X. In particular, if X is $\emptyset$, $h(\bar{x}) \models X$ for any match $h(\bar{x})$ of Q in G. We write $h(\bar{x}) \models X \rightarrow Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$.

Given a GQR $\varphi = Q[\bar{x}](X \rightarrow Y)$, a graph G and a node v in G, we say that a match $h(\bar{x})$ of Q in G is a *violation* of φ *pertaining to* node v if $h(\bar{x}) \not\models X \rightarrow Y$ and $v = h(x)$ for some $x \in \bar{x}$, *i.e., v* is involved in the violation (match).

A graph G *satisfies* GQR φ *w.r.t.* a set V_0 of nodes, denoted by $G \models_{V_0} \varphi$, if *for all* matches $h(\bar{x})$ of Q in G such that $h(x) \in V_0$ for some pattern node $x \in \bar{x}$, $h(\bar{x}) \models X \rightarrow Y$.

A graph G *satisfies* a set Σ of GQRs *w.r.t.* V_0, denoted by $G \models_{V_0} \Sigma$, if for all $\varphi \in \Sigma$, $G \models_{V_0} \varphi$.

We write $G \models \varphi$ (resp. $G \models \Sigma$) if $G \models_V \varphi$ (resp. $G \models_V \Sigma$), where V is the set of all nodes in G.

Example 4: Recall graph G and $V_0 = \{p_1\}$ of Example 1 and the GQRs of Example 3. Then $G \not\models_{V_0} \varphi_1$ as $h(x).\mathsf{item} \neq h(y).\mathsf{no\#}$ at $h : x \mapsto p_1$ and $y \mapsto o_1$. In contrast, $G \models_{V_0} \varphi_2$ since there is no match of pattern Q_2 pertaining to V_0. □

Remark. It is more challenging to clean graphs with GQRs than to clean relations with rules used in practice, *e.g.,*CFDs.

(1) As opposed to relations, real-life graphs typically do not have a schema. A schemaless graph does not specify what attributes exist. To cope with this, GQRs enforce the existence of certain attributes, and can fix limited edge errors (since edges can be coded as attributes like in RDF). In light of the existential semantics, GQRs cannot be expressed as CFDs or even as equality generating dependencies (EGDs [2]). In contrast, GQRs can express all EGDs and CFDs if we represent relation tuples as nodes in a graph [26].

(2) To clean a graph, we have to inspect not only its attribute values but also its topological structures. In particular, when two nodes are identified by literals like $x.\mathsf{id}$=$y.\mathsf{id}$, their attributes have to be pairwise equal. Worse yet, patterns in GQRs can be of unbounded size to specify correlated or co-occurred entities, making the graph cleaning problem harder due to the intractability of graph homomorphism.

(3) In addition, it demands methods different from repairing relations. For instance, to check whether a (connected) Q has a match $h(\bar{x})$ pertaining to a node v, *i.e.,* $h(x)$=v for $x \in \bar{x}$, we can inspect nodes within $|Q|$ hops of v. In contrast, relation repairing does not use this *locality* of graph homomorphism, but uses (expensive) joins to correlate entities.

Table 1: Notations

symbols	notations
G, $Q[\bar{x}]$	graph and graph pattern, respectively
φ, Σ	a GQR $\varphi = Q[\bar{x}](X \rightarrow Y)$, Σ is a set of GQRs
$h(\bar{x}) \models X \rightarrow Y$	a match $h(\bar{x})$ of $Q[\bar{x}]$ satisfies $X \rightarrow Y$
Γ	a block of ground truth
V_0	a set of nodes of users' interest
Eq, NEq	the relations of certain fixes
$[x]_{\mathsf{Eq}}$, $[x.A]_{\mathsf{Eq}}$	the certain fixes for node x and attribute $x.A$
G_{Eq}	the repair of G by Eq
$h(\bar{x}_s)$	a partial match of $Q[\bar{x}]$ with $\bar{x}_s \subseteq \bar{x}$

Special cases. We highlight some special cases of GQRs.

GFDs. Following [26], we refer to GQRs without id literals as GFDs, *i.e.,* neither X nor Y contains $x.\mathsf{id} = y.\mathsf{id}$ or $x.\mathsf{id} \neq y.\mathsf{id}$. In Example 3, φ_1, φ_3 and φ_5 are GFDs.

As shown in [19], GFDs extend CFDs [19] and can hence repair data by correcting attribute values along the same lines as CFDs. They are able to catch fraud as well, *e.g.,* φ_5 can decide whether e-commerce orders are fraudulent.

Keys. A *key* is a GQR $Q[\bar{z}](X \rightarrow x_0.\mathsf{id} = y_0.\mathsf{id})$, where $x_0, y_0 \in \bar{x}$ are two nodes in Q; *e.g.,* φ_2 and φ_4 are keys.

Keys are used to identify vertices that refer to the same entity, *i.e., object identification* [16]. As shown in Example 3, φ_4 identifies accounts using node id's of addresses. Keys may be recursively defined, *e.g.,* there is another key to identify address entities using the node id's of accounts (not shown).

*Forbidding constraints (*FCs*)*. An FC is a GQR $Q[\bar{x}](X \rightarrow x.\mathsf{id} \neq y.\mathsf{id})$, to deduce node pairs that should not be identified, *e.g.,* φ_6 tells us when two accounts do not match.

Summing up, GFDs help us repair inconsistent attribute values. Keys perform object identification. Negative rules FCs reduce false positives in object identification. The need for all these constraints has been verified by the experience of relational data cleaning (see, *e.g.,* [18] for a survey).

The notations of this paper are summarized in Table 1.

3. CERTAIN FIXES WITH THE CHASE

We define fixes (Section 3.1) and propose a model for deducing certain fixes by using GQRs (Section 3.2).

3.1 Fixes to Graphs

We start with a representation of fixes. Consider a graph $G = (V, E, L, F_A)$. We use x, y to range over nodes in G.

Fixes. We represent fixes as an equivalence relation, denoted by Eq. It includes equivalence classes $[x]_{\mathsf{Eq}}$ for nodes x in V, and $[x.A]_{\mathsf{Eq}}$ for attributes $x.A$ in $F_A(x)$. More specifically, $[x]_{\mathsf{Eq}}$ is a set of nodes $y \in V$, including x for all $x \in V$ in particular; and $[x.A]_{\mathsf{Eq}}$ is a set of attributes $y.B$ and constants c, including $x.A$ for all $x.A$ in $F_A(x)$.

Intuitively, for each $y \in [x]_{\mathsf{Eq}}$, the pair (x, y) is a match for object identification. For each $y.B \in [x.A]_{\mathsf{Eq}}$, $x.A = y.B$; and if $c \in [x.A]_{\mathsf{Eq}}$, then $x.A$ has value c, for data repairing. The relation Eq is reflexive, symmetric and transitive.

We also use relation NEq to keep track of entities that do not match. Here $[x]_{\mathsf{NEq}}$ includes nodes y such that $x.\mathsf{id} \neq y.\mathsf{id}$, and $[x.A]_{\mathsf{NEq}}$ includes c (resp. $y.B$) such that $x.A \neq c$ (resp. $x.A \neq y.B$). The relation is symmetric and "transitive" via Eq, *e.g.,* for $z \in [x]_{\mathsf{Eq}}$ and $w \in [x]_{\mathsf{NEq}}$, $z.\mathsf{id} \neq w.\mathsf{id}$.

For $x \in [y]_{\mathsf{Eq}}$ or $x \in [y]_{\mathsf{NEq}}$, we refer to (x, y) as a *fix*.

Cleaning. We clean G by applying the fixes of Eq as follows.

(1) For each $[x]_{\mathsf{Eq}}$ in Eq and $y \in [x]_{\mathsf{Eq}}$, *i.e.,* if x matches y, we merge x and y into a single node x_{Eq}, which retains the attributes and adjacent edges of x and y.

(2) For each $[x.A]_{\mathsf{Eq}}$, if constant $c \in [x.A]_{\mathsf{Eq}}$, we *generate new attribute* $x.A$ if $x.A$ does not yet exist, and repair $x.A$ with correct value c by letting $x_{\mathsf{Eq}}.A = c$, no matter whether $x.A$ has a value or not. Here x_{Eq} is identical to node x if it has not been joined with others as in (1).

(3) For each $[x.A]_{\mathsf{Eq}}$ and $y.B \in [x.A]_{\mathsf{Eq}}$, we *generate new attributes* when necessary. Moreover, we equalize $x.A$ and $y.B$ by letting $x_{\mathsf{Eq}}.A = y_{\mathsf{Eq}}.B = c$ if there exists $c \in [x.A]_{\mathsf{Eq}}$; otherwise we let $x_{\mathsf{Eq}}.A = y_{\mathsf{Eq}}.B = \#$, denoting value to be assigned to x_{Eq} and y_{Eq} like labeled nulls.

The process proceeds until all equivalence classes $[x]_{\mathsf{Eq}}$ and $[x.A]_{\mathsf{Eq}}$ of Eq are enforced on G. It yields a graph, referred to as *the repair of G by* Eq, denoted by G_{Eq}. The process supports object identification (when $y \in [x]_{\mathsf{Eq}}$) and data repairing (when $c \in [x.A]_{\mathsf{Eq}}$ or $y.B \in [x.A]_{\mathsf{Eq}}$) at the same time, and may *generate and instantiate new attributes.*

3.2 Deducing Certain Fixes

We next revise the chase to deduce (certain) fixes.

The chase. We compute fixes by chasing graph G with a set Σ of GQRs, starting from a designated set V_0 of nodes and a block Γ of ground truth represented as an equivalence relation Eq as in Section 3.1. For simplicity, we assume an initial $\mathsf{NEq} = \emptyset$. The chase is a classical tool in the relational database theory [2]. Below we extend it and make it a tool for deducing (certain) fixes to graphs. Intuitively, starting from nodes of V_0 in G and ground truth in Γ, the chase steps inductively deduce fixes by enforcing GQRs in Σ on G.

Formally, *a chase step of G by Σ at* $(\mathsf{Eq}, \mathsf{NEq}, V_{\mathsf{Eq}})$ is

$$(\mathsf{Eq}, \mathsf{NEq}, V_{\mathsf{Eq}}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}', \mathsf{NEq}', V_{\mathsf{Eq}'}).$$

Here $\varphi = Q[\bar{x}](X \to Y)$ is a GQR in Σ, V_{Eq} (resp. $V_{\mathsf{Eq}'}$) is a set of nodes in the repair G_{Eq} (resp. $G_{\mathsf{Eq}'}$) of G by Eq (resp. Eq'), and $h(\bar{x})$ is a match of Q in G_{Eq} with $h(x) \in V_{\mathsf{Eq}}$ for a node $x \in \bar{x}$, satisfying the conditions below:

(1) X is *entailed by* $(\mathsf{Eq}, \mathsf{NEq})$ at $h(\bar{x})$, *i.e.,* for each literal $l \in X$, if l is $x.A = c$, then $c \in [h(x).A]_{\mathsf{Eq}}$; similarly for other literals with Eq and inequality literals with NEq. Abusing the notations, here $h(x)$ and $h(y)$ range over all nodes of $[h(x)]_{\mathsf{Eq}}$ and $[h(y)]_{\mathsf{Eq}}$ in the original graph G, respectively.

(2) Either Eq' extends Eq or NEq' extends NEq with fixes.

(2.1) Eq' extends Eq by instantiating one equality literal $l \in Y$; *e.g.,* if l is $x.\mathsf{id} = y.\mathsf{id}$ and $h(y) \notin [h(x)]_{\mathsf{Eq}}$, then add $h(y)$ to $[h(x)]_{\mathsf{Eq}'}$ and $h(y).A$ to $[h(x).A]_{\mathsf{Eq}'}$ for each attribute A of $h(y)$ in G_{Eq}; similarly for other literals. We make Eq' an equivalence relation (reflexive, symmetric and transitive).

(2.2) NEq' extends NEq by instantiating an inequality literal $l \in Y$ as in (2.1); *e.g.,* if l is $x.\mathsf{id} \neq y.\mathsf{id}$ and $h(y) \notin [h(x)]_{\mathsf{NEq}}$, then NEq' extends NEq by adding $h(y)$ to $[h(x)]_{\mathsf{NEq}'}$.

(3) The set $V_{\mathsf{Eq}'}$ extends V_{Eq} with v_p if it is not there, where v_p is the node *updated* in the chase step, *i.e.,* it has an attribute or id different from its counterpart in G_{Eq}. It also removes those nodes that no longer exist in $G_{\mathsf{Eq}'}$.

Validity. We say that the chase step $(\mathsf{Eq}, \mathsf{NEq}, V_{\mathsf{Eq}}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}', \mathsf{NEq}', V_{\mathsf{Eq}'})$ is *valid* if none of conflicts below occurs:

- there exists $y \in [x]_{\mathsf{Eq}'}$ such that $L(x) \neq L(y)$, *i.e.,* Eq' merges nodes with distinct labels;
- there exist $y.B \in [x.A]_{\mathsf{Eq}'}$, $c \in [x.A]_{\mathsf{Eq}'}$ and $d \in [y.B]_{\mathsf{Eq}'}$ such that $c \neq d$, *i.e.,* Eq' assigns distinct values to the same attribute; or
- either $[x]_{\mathsf{Eq}'} \cap [x]_{\mathsf{NEq}'} \neq \emptyset$ or $[x.A]_{\mathsf{Eq}'} \cap [x.A]_{\mathsf{NEq}'} \neq \emptyset$, *i.e.,* Eq' and NEq' must be disjoint for all x and $x.A$.

Otherwise we say that Eq' is *inconsistent.*

One can verify that when Eq' is consistent, the repair $G_{\mathsf{Eq}'}$ of G by fixes Eq' is well defined. In particular, if A is an attribute of both x and y, and if $y \in [x]_{\mathsf{Eq}'}$, then $x.A = y.A$.

Chasing. A *chasing sequence* ρ of G by (Σ, Γ) from V_0 is

$$(\mathsf{Eq}_0, \mathsf{NEq}_0, V_{\mathsf{Eq}_0}), \ \ldots, \ (\mathsf{Eq}_k, \mathsf{NEq}_k, V_{\mathsf{Eq}_k}).$$

Here $\mathsf{Eq}_0 = \Gamma$, $\mathsf{NEq}_0 = \emptyset$, $V_{\mathsf{Eq}_0} = V_0$, and moreover, for all $i \in [1, k]$, there exist a GQR $\varphi = Q[\bar{x}](X \to Y)$ in Σ and a match h of Q in the repair $G_{\mathsf{Eq}_{i-1}}$ such that $(\mathsf{Eq}_{i-1}, \mathsf{NEq}_{i-1}, V_{\mathsf{Eq}_{i-1}}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}_i, \mathsf{NEq}_i, V_{\mathsf{Eq}_i})$ is a valid chase step.

This is used to deduce fixes to errors pertaining to V_0 and *recursively* resolves violations related to the nodes updated in the chase steps. That is, we focus on *local* fixing of the errors related to a set V_0 of vertices picked by users.

The sequence is *terminal* if there exist no GQR φ in Σ, match h of Q in G_{Eq_k} and $(\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1}, V_{\mathsf{Eq}_{k+1}})$ such that $(\mathsf{Eq}_k, \mathsf{NEq}_k, V_{\mathsf{Eq}_k}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1}, V_{\mathsf{Eq}_{k+1}})$ is valid.

Example 5: Consider the GQRs of Example 3 and graph G of Fig. 1 with a block Γ of ground truth including those attributes underlined. Then $[x]_{\mathsf{Eq}_0} = \{x\}$ and $[x.A]_{\mathsf{Eq}_0} = \{x.A, \Gamma(x.A)\} \cup \{y.B \mid \Gamma(x.A) = \Gamma(y.B)\}$ for each node x and attribute $x.A$ in G, where $\Gamma(x.A)$ (resp. $\Gamma(y.B)$) is the confirmed value of $x.A$ (resp. $y.B$) in Γ if it exists.

From $V_0 = \{p_1\}$, a chasing sequence ρ starts with step (1): $(\mathsf{Eq}_0, \mathsf{NEq}, \{p_1\}) \Rightarrow_{(\varphi_1,h_1)} (\mathsf{Eq}_1, \mathsf{NEq}, \{p_1, o_1\})$, where $\mathsf{NEq} = \emptyset$; h_1: $x \mapsto o_1$, $y \mapsto p_1$; and Eq_1 extends Eq_0 by letting $[o_1.\mathsf{item}]_{\mathsf{Eq}_1} = [p_1.\mathsf{no}\#]_{\mathsf{Eq}_1} = \{o_1.\mathsf{item}, p_1.\mathsf{no}\#, 57761\}$. This first chase step repairs the value of $o_1.\mathsf{item}$ to 57761, *i.e.,* $o_1.\mathsf{item}$ is updated and hence included in V_{Eq_1}.

The sequence ρ can further (2) repair (*i.e.,* confirm) the value of $o_2.\mathsf{item}$ in Eq_2; (3) identify entities o_1 and o_2 in Eq_3; (4) include "Allan Fox" in $[a_2.\mathsf{name}]_{\mathsf{Eq}_4}$; (5) identify a_1 and a_2 by combining equivalence classes of a_1 and a_2 along with their attributes in Eq_5; and finally (6) add *true* to both $[o_1.\mathsf{is\_fake}]_{\mathsf{Eq}_6}$ and $[o_2.\mathsf{is\_fake}]_{\mathsf{Eq}_6}$.

The chase *interleaves* data repairing and object identification: correcting $a_2.\mathsf{name}$ in step (4) helps identify accounts in step (5), which in turn helps deduce certain fix to attribute is_fake to catch fraudulent orders in step (6). □

Chasing sequence ρ *terminates* in one of the cases below.

(a) No GQRs in Σ can be further applied. If so, we say that ρ is *valid*, and refer to $(\mathsf{Eq}_k, \mathsf{NEq}_k, G_{\mathsf{Eq}_k}, V_{\mathsf{Eq}_k})$ as *its result.* One can verify that G_{Eq_k} is well-defined.

(b) Either Eq_0 is inconsistent or there exist φ, h, Eq_{k+1}, NEq_{k+1} and $V_{\mathsf{Eq}_{k+1}}$ such that $(\mathsf{Eq}_k, \mathsf{NEq}_k, V_{\mathsf{Eq}_k}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1}, V_{\mathsf{Eq}_{k+1}})$ but Eq_{k+1} is inconsistent. Such ρ is *invalid*, with result $\perp$ (undefined).

Church-Rosser property. It is natural to ask whether the chase always terminates with the same fixes. Following [2], we say that chasing with GQRs has the *Church-Rosser property* if for all Σ, Γ, G and V_0 in G, all terminal chasing sequences of graph G by (Σ, Γ) from V_0 converge at the same result, regardless of in what order the GQRs are applied.

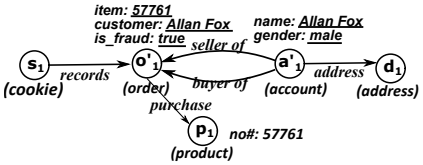


Figure 3: The repair of graph G

Theorem 1: *For any graph G, any set Σ of* GQRs*, any block Γ of ground truth, and any set V_0 of nodes within G, all chasing sequences of G by (Σ, Γ) from V_0 are terminal, and moreover, converge at the same result.* □

Thus we can define *the result of chasing G by (Σ, Γ) from V_0* as the result of any terminal chasing sequence of G by (Σ, Γ) from V_0, denoted by $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$.

For instance, Figure 3 depicts the repair G_{Eq_6} of graph G of Fig. 1 from V_0=$\{p_1\}$, by the GQRs of Example 3 and the ground truth of Example 5. The result of chasing is $(\mathsf{Eq}_6, \mathsf{NEq}, G_{\mathsf{Eq}_6}, \{p_1, o'_1, a'_1\})$; it remains the same no matter what rules of Σ are used and how they are applied. Now all attributes in G_{Eq_6} have correct values (underlined).

Proof: We show that in any chasing sequence ρ of G by (Σ, Γ) from V_0, $|\mathsf{Eq}_i| \leq 4|G||\Sigma|+2|\Gamma|$ and $|\mathsf{NEq}_i| \leq (4|G||\Sigma|+2|\Gamma|)^2$ at any chase step. Based on the size bounds, we show that the length of ρ is at most $8(2|G||\Sigma| + |\Gamma|)^3$.

We prove the Church-Rosser property by contradiction. Suppose that there exist two chasing sequences that end up with different results. Then one of them is not terminal. □

Remark. As opposed to the chase of [24] for the satisfiability and implication analyses of GEDs, (a) a chase step by GQR $Q[\bar{x}](X \to Y)$ is applied *only if* X has been validated (condition (1) of the chase step), *i.e., Y* is enforced only after all literals of X are deduced from *ground truth* Γ; (b) we revise the chase to identify objects and repair attributes for errors arisen by a particular set of nodes (entities), rather than for the static analyses of dependencies; and (c) Theorem 1 extends the result of [24] to GQRs with inequality literals.

Certain fixes. When $\mathsf{chase}(G, \Sigma, \Gamma, V_0) \neq \bot$ for *any* set V_0 of nodes in G, we say that (Σ, Γ) is *consistent for G*. Given consistent (Σ, Γ), all terminal chasing sequences of G by (Σ, Γ) from V_0 are valid, and end up with the same result $\mathsf{chase}(G, \Sigma, \Gamma, V_0) = (\mathsf{Eq}, \mathsf{NEq}, G_{\mathsf{Eq}}, V_{\mathsf{Eq}})$ by Theorem 1. In this case, we refer to Eq and G_{Eq} as *the certain fixes* and *the repair* of graph G by (Σ, Γ) *w.r.t.* V_0, respectively.

Intuitively, a chasing sequence ρ deduces fixes to errors pertaining to V_0 from confirmed matches and attributes by iteratively applying GQRs, using ground truth Γ. The fixes in Eq are *certain* since they are logical consequences of (Σ, Γ), *i.e.,* as long as Σ and Γ are correct, so are the fixes.

4. FUNDAMENTAL PROBLEMS

We now outline a framework to clean graphs with certainty. We also identify three fundamental problems underlying certain fix deduction, and settle their complexity.

Framework. We propose a model for cleaning graphs, referred to as GFix and shown in Fig. 4. Given graphs G, it starts with offline preprocessing. (a) It discovers a set Σ of GQRs by using discovery algorithms (*e.g.,* [21]), and solicits initial ground truth Γ by consulting experts, crowd-sourcing, and referencing high-quality knowledge bases [29]. (b) It validates the rules and ground truth by a consistency analysis, to ensure that Σ and Γ have no conflicts (see below).

Online. When Σ and initial Γ are in place, GFix interacts with users in the online mode. Users iteratively pick a small

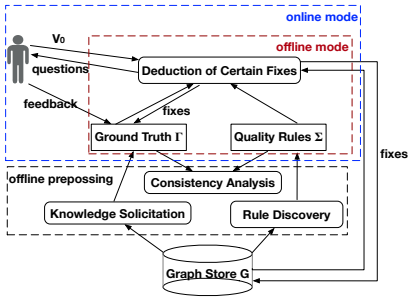


Figure 4: GFix: A model for cleaning graphs

set V_0 of nodes in G, which represent entities of their interest or entities newly updated. GFix deduces certain fixes to all errors pertaining to V_0. The repairing problem is as follows.

- *Input*: A graph G, a small set V_0 of nodes in G of users' interest, a set Σ of GQRs, and a block Γ of ground truth such that (Σ, Γ) is consistent for G.
- *Output*: Certain fixes and repair of G by (Σ, Γ) *w.r.t.* V_0.

GFix may not be able to fix all errors pertaining to V_0 if Γ does not have adequate information to cover V_0. If this happens, GFix asks users to inspect a small number of attributes or entities. It expands Γ with the validated fields, incrementally checks the consistency of Σ and Γ, and continues to fix remaining errors. The interaction iterates in rounds until all violations of Σ pertaining to V_0 are fixed.

Offline. Periodically GFix repairs the entire graph G in the backend, after Γ accumulates sufficient ground truth, or if graph G has been substantially changed over time. This is the repairing problem above when V_0 is the entire set of nodes in G. When user interaction is turned off and if Γ is not informative enough, GFix may not be able to fix all violations of Σ in G. However, GFix generates only certain fixes and hence guarantees to improve the quality of G.

We remark the following. (1) In practice, the number of GQRs in Σ is small, and Σ is validated by domain experts before it is used. (2) Consistency checking is invoked automatically to check whether (Σ, Γ) is consistent for G before each run online or offline. (2) GFix also periodically consults domain experts, conducts crowd-sourcing and references high-quality knowledge bases to both validate the changes that have been made to G, and expand ground truth Γ.

Fundamental problems. We will provide practical algorithms for the online and offline mode of GFix in Sections 5 and 6, respectively. Below we first settle fundamental problems underlying GFix for deducing certain fixes.

Consistency. We start with *the consistency problem*.

- *Input*: A set Σ of GQRs and a block Γ of ground truth.
- *Question*: Is (Σ, Γ) consistent for G?

It is to decide whether the GQRs discovered and the ground truth accumulated have no conflicts.

The problem has the same complexity as its relational counterpart using extended CFDs [23]. That is, our choice of data quality rules on graphs does not make our lives harder.

Theorem 2: *The consistency problem is* coNP*-complete.* □

Proof: We give an NP algorithm to check whether (Σ, Γ) is *inconsistent* for G, based on the Church-Rosser property and the upper bound on the length of chasing sequences given in the proof of Theorem 1. The problem is shown coNP-hard by reduction from the satisfiability problem for GEDs, which is coNP-complete [24]. It is to decide, given a set Σ_1 of GEDs, whether there exists a graph G such that $G \models \Sigma_1$ and for each $Q[\bar{x}](X \to Y)$ in Σ_1, Q has a match in G. □

Cleaning. The *certain fix problem* is stated as follows.

- *Input*: A graph G, a set V_0 of nodes in G, a fix l, GQRs Σ and block Γ such that (Σ, Γ) is consistent for G.
- *Question*: Does $v \in [u]_{\mathsf{Eq}}$ (resp. $v \in [u]_{\mathsf{NEq}}$) when l is an equality fix $u = v$ (resp. an inequality fix $u \neq v$)?

Here Eq and NEq are certain fixes in $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$.

This is to settle the complexity of deducing certain fixes. We study the combined complexity, when graph G, nodes V_0, GQRs Σ and ground truth Γ may all vary; and the data complexity, when the Σ is fixed, but G, V_0 and Γ may vary. The problem was not studied for relations [23].

Theorem 3: *The certain fix problem is* NP*-complete, and its data complexity is in* PTIME. □

Proof: We give an NP algorithm that guesses a chasing sequence of bounded length and checks whether $v \in [u]_{\mathsf{Eq}}$ or $v \in [u]_{\mathsf{NEq}}$, by Theorem 1. When Σ is fixed, we can compute $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$ in polynomial time in $|G|$, $|V_0|$ and $|\Gamma|$, and hence the data complexity is in PTIME. We prove the lower bound by reduction from the 3-colorability problem, which is NP-complete [28]. The latter problem is to decide, given an undirected graph G_1, whether there exists a 3-coloring ν of G_1 such that for each edge (u, v) in G_1, $\nu(u) \neq \nu(v)$. □

Coverage. Does Γ have enough information to fix *all violations* of Σ pertaining to V_0? In particular, can GFix correct all errors in G without user interaction when V_0 is the set of nodes in G? This gives rise to *the coverage problem.*

- *Input*: G, V_0, Σ and Γ as in the certain fix problem.
- *Question*: Does $G_{\mathsf{Eq}} \models_{V_{\mathsf{Eq}}} \Sigma$ for the repair G_{Eq} of graph G by (Σ, Γ) *w.r.t.* V_0?

Here V_{Eq} is the set of nodes in the result of $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$.

For relations, the data complexity was not studied, and the combined complexity of a stronger problem is coNP-complete [23], to decide whether all instances of a relation schema can be fixed with certainty. We show that unless P = NP, the coverage problem is harder for graphs, in $\mathsf{P}_{||}^{\mathsf{NP}}$.

The complexity class $\mathsf{P}_{||}^{\mathsf{NP}}$ consists of decision problems that can be solved by a PTIME Turing machine making polynomially many queries to an NP oracle in parallel, *i.e.,* all queries are formed before knowing the results of the others [48]. The class $\mathsf{P}_{||}^{\mathsf{NP}}$ is a subclass of $\Delta_2^p = \mathsf{P}^{\mathsf{NP}}$ [38].

Theorem 4: *The coverage problem is* $\mathsf{P}_{||}^{\mathsf{NP}}$*-complete; and its data complexity is in* PTIME. □

Proof: This proof is quite involved.

Upper bound. It suffices to compute $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$, and check whether $G_{\mathsf{Eq}} \models_{V_{\mathsf{Eq}}} \Sigma$. There are polynomially many fixes in $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$ (see proof of Theorem 1). Thus we can check fixes in $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$ in parallel using an NP oracle by Theorem 3. We use another query to check whether $G_{\mathsf{Eq}} \models_{V_{\mathsf{Eq}}} \Sigma$ in coNP [24]. These yield an algorithm using two rounds of parallel queries. As using constant rounds of parallel queries is equivalent to using one round of parallel queries [10], the algorithm is in $\mathsf{P}_{||}^{\mathsf{NP}}$. The data complexity follows from that computing $\mathsf{chase}(G, \Sigma, \Gamma, V_0)$ and checking $G_{\mathsf{Eq}} \models_{V_{\mathsf{Eq}}} \Sigma$ are both in PTIME when Σ is fixed.

Lower bound. We show the lower bound by reduction from the odd max true 3SAT (OMT3) problem, which is $\mathsf{P}_{||}^{\mathsf{NP}}$-complete [45]. OMT3 is to decide, given a 3-CNF formula ϕ with variables $\bar{x}$, whether there is a truth assignment μ of $\bar{x}$ such that μ satisfies ϕ with maximum true among all assignments, and it sets an odd number of variables true. □

5. DEDUCTION ALGORITHM

We now provide a practical single-machine algorithm for the online mode of GFix, referred to as GMend, to deduce certain fixes pertaining to a small set V_0 of nodes.

Overview. The chase of Section 3 provides a conceptual-level method. However, it is too costly to be practical. Each chase step starts from scratch: it (a) nondeterministically picks a GQR $\varphi = Q(\bar{x})(X \to Y)$ from Σ, (b) finds a mapping h from $Q[\bar{x}]$ to G pertaining to some nodes in V_0, and (c) checks whether X is entailed by (Eq, NEq) at $h(\bar{x})$. Among these, step (b) is costly given the intractability of graph homomorphism (cf. [28]). After computing h, the chase may find that X is not entailed by (Eq, NEq) and φ cannot be enforced; then it has to start steps (b) and (c) again.

To make the chase practical, algorithm GMend adopts two strategies. (a) It *deterministically* picks GQRs from Σ to expand fixes in the current chase step by capturing the impacts of GQRs enforced in prior steps. (b) It *incrementally* expands Eq and NEq, *i.e.,* a chase step is taken *only if* it is "triggered" by newly generated fixes. These reduce redundant computation of costly graph homomorphism.

(a) Selection of GQRs. To carry out deterministic selection, we use a notion of templates. For an equality literal l in GQR $Q[\bar{x}](X \to Y)$, we define its *template* $t_l(l)$ by substituting label $L_Q(x)$ for each node x in l, *e.g.,* $L_Q(x).A{=}c$ for $x.A{=}c$ and $L_Q(x).A = L_Q(y).B$ for $x.A = y.B$. We also extend t_l to fixes (u, v) in Eq and NEq along the same lines. The templates are used to catch precedence on the constraints of GQRs, by "generalizing" nodes to their labels.

Consider a pair of GQRs $\varphi_i = Q_i[\bar{x}](X_i \to Y_i)(i \in [1, 2])$. Then firing φ_1 can cause the firing of φ_2 at a later chase step only if Y_1 and X_2 use the same attributes after the "generalization". More specifically, φ_2 is a *successor* of φ_1 in the precedence relationship *only if* (i) there exist $y.A$ in Y_1 and $x.A$ in X_2, such that $L_{Q_2}(x) \asymp L_{Q_1}(y)$; or (ii) there exist literal $y.\mathsf{id} = y'.\mathsf{id}$ in Y_1 and pattern node x in Q_2 such that $L_{Q_2}(x) \asymp L_{Q_1}(y) \asymp L_{Q_1}(y')$; or (iii) there exists a constant c that appears in both Y_1 and X_2.

Example 6: Consider GQRs $\varphi_1'{=}Q[\bar{x}](x.A{=}y.B \to x.C{=}x.D)$ and $\varphi_2'{=}Q[\bar{x}](x.C{=}y.D \to x.A{=}y.A)$, where Q consists of two nodes labeled 'a'. Templates of literals in φ_1' and φ_2' are 'a'.A='a'.B, 'a'.C='a'.D, 'a'.C='a'.D, and 'a'.A='a'.A. We can verify that φ_2' is a successor of φ_1', since they have the same attribute 'a'.C satisfying condition (i). □

Based on the precedence relation, GMend applies a GQR φ at one step only when φ is a successor for some GQR that has been already enforced during the last step.

(b) Incremental expansion. To achieve this, GMend only finds mappings h that involve nodes updated in the last step. For instance, after φ_1' of Example 6 is applied, it enforces φ_2' to the area surrounding the nodes involved in newly generated fixes (see procedures filterPAns and Match below).

Algorithm. As shown in Fig. 5, GMend first initializes fixes Eq, repair G_{Eq}, a set V_d of "designated" nodes by taking ground truth Γ, graph G and nodes V_0, respectively (line 1). It then iteratively selects active GQRs from Σ based on precedence and applies the GQRs to deduce fixes of graph G *w.r.t.* V_0 (lines 3-11). It invites users to inspect a set of fields for expanding Γ when errors persist (lines 13-14), and continues to compute fixes with Γ revised by users (line 15).

Input: Graph G, set V_0 of nodes in G, GQRs Σ, ground truth Γ.
Output: The repair G_{Eq} of G *w.r.t.* V_0.

1. $\mathsf{Eq} := \Gamma$; $\mathsf{NEq} := \emptyset$; $G_{\mathsf{Eq}} := G$; $V_d := V_0$;
2. $\mathcal{A} := \Sigma[t_l(\Gamma)]$; $\delta_{\mathsf{Eq}} := \Gamma$;
3. **repeat**
4. **while** δ_{Eq} is not empty **do**
5. $\mathcal{A}_{\mathsf{next}} := \emptyset$; $\delta_{\mathsf{cur}} := \emptyset$;
6. **for each** GQR $\varphi = Q[\bar{x}](X \to Y)$ in $\mathcal{A}$ **do**
7. $S_\varphi := \mathsf{filterPAns}(X, \mathsf{Eq}, \mathsf{NEq}, \delta_{\mathsf{Eq}})$;
8. $F := \mathsf{Match}(Q[\bar{x}], G_{\mathsf{Eq}}, S_\varphi, V_d)$;
9. $\delta_{\mathsf{cur}} := \delta_{\mathsf{cur}} \cup F$; $\mathcal{A}_{\mathsf{next}} := \mathcal{A}_{\mathsf{next}} \cup \varphi.\mathsf{succ}[t_l(F)]$;
10. $(\mathsf{Eq}, \mathsf{NEq}, \delta_{\mathsf{Eq}}) := \mathsf{cEqv}(\delta_{\mathsf{cur}}, \mathsf{Eq}, \mathsf{NEq})$;
11. mutate G_{Eq} by δ_{Eq}; expand V_d with δ_{Eq}; $\mathcal{A} := \mathcal{A}_{\mathsf{next}}$;
12. $\mathcal{E} := \mathsf{Vio}(\Sigma, G_{\mathsf{Eq}}, V_d)$; /* *user interaction* */
13. **if** $\mathcal{E} \neq \emptyset$ **then** invite users to inspect fields relevant to $\mathcal{E}$;
14. expand Γ with validated fields;
15. update Eq, NEq, δ_{Eq}, V_d, G_{Eq}; update $\mathcal{A}$ with δ_{Eq};
16. **until** $\mathcal{E} = \emptyset$;
17. **return** G_{Eq};

Figure 5: Algorithm GMend

The process proceeds until all the violations pertaining to V_0 and nodes updated in the process are fixed (line 16). Then GMend returns the repair G_{Eq} *w.r.t.* V_0 (line 17).

We next present its procedures filterPAns and Match, and the details of user interaction in GMend (lines 12-14).

Procedure filterPAns. The procedure finds partial matches that involve nodes in δ_{Eq}, which were updated in the *last* iteration. More specifically, it takes the following as input: X from GQR $Q[\bar{x}](X \to Y)$, Eq, NEq, and fixes δ_{Eq} newly added. It finds a set S_φ of homomorphic mappings $h(\bar{x}_s)$ from variables that appear in X to nodes in Eq and NEq, refereed to as *partial matches*. It ensures that each $h(\bar{x}_s)$ in S_φ satisfies X and contains at least one node from δ_{Eq}.

To speed up this process, filterPAns dynamically maintains an inverted index from templates t to fixes with the same t. It combines newly enforced fixes with previous ones to *incrementally* find partial matches where X is validated.

Procedure Match. It completes partial matches S_φ of Q that were computed by filterPAns. Following generic subgraph matching [37], it verifies graph homomorphism by checking edge connections. It also deduces new fixes to be included in Eq and NEq. The procedure only *incrementally enumerates* matches by accessing nodes within $|Q|$ hops of the designated nodes V_d by the locality (Section 2). If $S_\varphi = \emptyset$, it ensures that the matches involve lately merged nodes.

User interaction. When no more fixes can be found, GMend checks whether all violations of Σ pertaining to V_d are fixed by calling procedure Vio (line 12). Similar to Match, Vio adopts the locality to enumerate matches *violating* the attribute constraints $X \to Y$. For each such violation h, users are invited to inspect a *field relevant to* h, which includes a subset of attributes and entities in h instantiating the literals of X and Y only (line 13). The small confirmed field helps us fix violation h and expand ground truth Γ (line 14). Then these newly validated fixes are added to δ_{Eq} directly, and fixes Eq, NEq, designated nodes V_d and repair G_{Eq} are updated accordingly (line 15). At the end of GMend, Γ is also expanded with deduced certain fixes.

Example 7: Given graph G of Fig. 1, GQRs Σ of Example 3, ground truth Γ of Example 5 (excluding a_1.name), and product p_1 picked by user, GMend computes $\mathsf{chase}(G, \Sigma, \Gamma, \{p_1\})$ iteratively, guided by the precedence of GQRs.

Consider the processing of φ_2. After o_1.item is updated by φ_1 (see Example 5), procedure filterPAns first selects a fix o_1.item=o_2.item from Eq that has the same templates as for y.item=y'.item of φ_2. It finds a partial match $h(\bar{x}_s)$ in which y (resp. y') is mapped to o_1 (resp. o_2). It then completes this partial match via procedure Match, which maps x to s_1 and returns a fix o_1.id=o_2.id. Thus GMend updates G by merging o_1 and o_2, and GQR φ_3 is reserved for the next iteration due to the inclusion of an order node in Q_3.

Since there still is a violation of φ_4 between the node id's of accounts a_1 and a_2 after all fixes are deduced, GMend asks the user to check their name values to expand Γ. It proceeds to find fixes once a_1.name is validated and added to Γ. □

Analyses. Algorithm GMend checks all the applicable chase steps that can extend Eq or NEq at some point, guided by the precedence of Σ. Its correctness follows from Proposition 5, which is verified by induction on the number of chase steps.

Proposition 5: *All the fixes are included in* Eq *and* NEq *following the shortest sequences by algorithm* GMend. □

Algorithm GMend is in $O(|\Sigma||G|^{|\Sigma|})$ time (excluding user interaction time), since both deducing fixes and checking violations take $O(|\Sigma||G|^{|\Sigma|})$ time, dominating the cost.

6. PARALLEL SCALABLE ALGORITHM

Theorem 3 tells us that deducing certain fixes to graphs is intractable. Algorithm GMend works in the online mode when the set V_0 of entities of users' interest is small. However, when V_0 is large or when we want to repair entire graphs in the offline mode, it is often necessary to use parallel algorithms. Below we first review a characterization of parallel algorithms (Section 6.1). We then parallelize algorithm GMend, with a performance guarantee (Section 6.2).

6.1 Parallel Scalability

One might be tempted to think that a parallel algorithm would run faster given more resources. However, few algorithms in the literature guarantee this. Worse still, for some computation problems, no parallel algorithms run much faster no matter how many processors are added [25]. This suggests that we characterize the effectiveness of parallel algorithms. To this end we revise a notion of parallel scalability introduced by [35] and widely used in practice.

A parallel algorithm $\mathcal{A}_p$ for deducing certain fixes is said to be *parallel scalable relative to* the sequential algorithm GMend if its running time can be expressed as:

$$T(|G|, |\Sigma|, |\Gamma|, |V_0|, p) = O(\frac{t(|G|, |\Sigma|, |\Gamma|, |V_0|)}{p}),$$

where $t(|G|, |\Sigma|, |\Gamma|, |V_0|)$ denotes the cost of GMend, and p is the number of processors used for parallel computation.

Intuitively, a parallel scalable $\mathcal{A}_p$ reduces the cost when p increases, by taking sequential GMend as a yardstick. Hence $\mathcal{A}_p$ is able to run faster when adding more processors, and thus makes it feasible to scale with large graphs G.

6.2 Parallelizing Algorithm GMend

We provide an algorithm, denoted as PMend, by parallelizing algorithm GMend. It fixes graphs that are fragmented and distributed across workers (processors) $W_0, \ldots, W_{p-1}$. We show that PMend is parallel scalable relative to GMend.

Algorithm PMend runs in supersteps, in which GQRs that are inspected in the same iteration of GMend are processed *in parallel in the same superstep*. Similar to algorithm GMend, PMend expands Eq and NEq *incrementally*, guided

by the precedence relationship of GQRs, while the procedures of partial match identification and completion are *parallelized* by using a workload partition strategy. Relations Eq and NEq are also distributed across p workers, such that each worker W_i maintains $\mathsf{Eq}^{(i)}$ and $\mathsf{NEq}^{(i)}$, and each fix in the local copies contains at least one node stored at W_i.

We next show how to parallelize filterPAns for partial match identification and Match for partial match completion, to evenly partition their computations across p workers, which dominate the cost of algorithm GMend.

Parallel partial match. This is to identify partial matches that satisfy the entailment condition of the chase. It is performed in parallel by procedure PFilter. The challenges here are that a partial match may be deduced from fixes that are filtered and maintained at different workers,

Procedure PFilter takes a set $\mathcal{A}$ of active GQRs as part of its input. Consider GQR $Q[\bar{x}](X \to Y)$ in $\mathcal{A}$, and a literal l in X. At each worker W_i, PFilter inspects the fixes in the local $\mathsf{Eq}^{(i)}$ and $\mathsf{NEq}^{(i)}$ that share the same template $t(l)$ of l, referred to as *the fixes by* $t(l)$. Since fixes are scattered all over the p workers, in contrast to sequential filterPAns that combines them directly to construct partial matches, W_i broadcasts its local fixes filtered to other workers. Upon receiving the fixes, all workers sort the set of fixes by $t(l)$, denoted by $\mathsf{Eq}[t_l]$, based on a predefined order.

It then assembles these fixes to construct partial matches *in parallel*. Let $t_1, \ldots, t_m$ be the templates of literals in X. Since the cardinality $\|\mathsf{Eq}[t_i]\|$ of each $\mathsf{Eq}[t_i]$ is known, PFilter evenly partitions the assembling work, in which each worker W_j is assigned an index IDX_i (range) for sorted $\mathsf{Eq}[t_i]$, such that the indices $\{\mathsf{IDX}_i \mid i \in [1, m]\}$ of its groups satisfy

$$\Sigma_{i=1}^{m-1}(\mathsf{IDX}_i \cdot \Pi_{s=i+1}^{m}\|\mathsf{Eq}[t_s]\|) + \mathsf{IDX}_m \bmod p = j.$$

That is, the combinations of fixes are evenly partitioned across p workers in a "round-robin" manner. The partial matches with nodes from newly deduced fixes are returned.

Example 8: Consider GQR φ_6 of Example 3, a graph G' with accounts m_0–m_9, where m_0–m_4 (resp. m_5–m_9) have value "male" (resp. "female") for gender as ground truth.

Given these, PFilter first picks from ground truth $\mathsf{Eq}[t_1] = \{m_i.\mathsf{gender}=\text{"male"} \mid i \in [0, 4]\}$ and $\mathsf{Eq}[t_2]=\{m_j.\mathsf{gender} = \text{"female"} \mid j \in [5, 9]\}$, where t_1 (resp. t_2) is a template of account.gender="male" (resp. "female"). Then it evenly distributes the work of combining fixes in $\mathsf{Eq}[t_1]$ and $\mathsf{Eq}[t_2]$, 25 in total, to derive partial matches. Assume that there are 5 workers, and $\mathsf{Eq}[t_1]$ (resp. $\mathsf{Eq}[t_2]$) is such sorted that $m_i.\mathsf{gender}$="male" (resp. $m_j.\mathsf{gender}$="female") is assigned index i (resp. $j-5$). By the equation above, each worker W_k validates 5 partial matches consisting of m_i and m_j with $5i + j \mod 5 = k$, for $k \in [0, 4]$, *i.e.,* x (resp. y) in Q_6 is mapped to account m_i (resp. m_j). □

Parallel match completion. PMend invokes procedure PMatch to complete the partial matches and to find fixes to be added, *in parallel*. PMatch works as follows.

(1) For each partial match $h(\bar{x}_s)$ of $Q[\bar{x}]$ assembled at worker W_i, PMatch identifies the *candidates* $\mathcal{C}_h(u)$ for pattern nodes u that remain to be matched in G_{Eq}, by breadth first search from the designated nodes V_d, *i.e.,* local search. Here $\mathcal{C}_h(u)$ consists of nodes v such that $L(v)$ matches $L_Q(u)$; and there exists $v' \in \mathcal{C}_h(u')$ or v' matches u' in $h(\bar{x}_s)$ such that edge (v', ι, v) (resp. (v, ι, v')) is in G_{Eq} and ι matches label ι' of (u', ι', u) (resp. (u, ι', u')) in $Q[\bar{x}]$. When reaching "borders"

Input: A fragmented graph G, and V_0, Σ and Γ as in GMend.
Output: The repair G_{Eq} of G *w.r.t.* V_0.

1. initialize Eq, NEq, G_{Eq}, V_d; $\Delta_{\mathsf{Eq}} := 0$; $\mathcal{A} := \Sigma$;
2. **while** there exist violations of Σ pertaining to V_d **do**
3. **repeat** /*one superstep*/
4. $S_{\mathcal{A}}$:=PFilter($\mathcal{A}$, Eq, NEq, Δ_{Eq}); /*parallel identification*/
5. Δ_{Eq}:=PMatch($\mathcal{A}, G_{\mathsf{Eq}}, S_{\mathcal{A}}, V_d$); /*parallel completion*/
6. expand Eq and NEq with Δ_{Eq};
7. update G_{Eq}, V_d by Δ_{Eq}; update $\mathcal{A}$ with Δ_{Eq};
8. **until** Δ_{Eq} is empty;
9. expand Γ via user interaction (if in the online mode);
10. **return** G_{Eq};

Figure 6: Algorithm PMend

of the partitioned graph, *e.g.,* nodes with crossing edges to other fragments, it notifies other workers the next pattern node to match, and the traversal continues there.

Then $h(\bar{x}_s)$ is broadcast along with the candidates and connecting edges. It does not broadcast those complete matches, which will be validated locally. Each worker sorts the candidates like how PFilter treats $\mathsf{Eq}[t_l]$.

(2) Procedure PMatch then completes the partial matches by combining and verifying candidates for pattern nodes that remain to be matched, *in parallel*. For each partial match $h(\bar{x}_s)$ of $Q[\bar{x}]$, worker W_j groups candidates from $\mathcal{C}_h(u)$ for each $u \in \bar{x} \setminus \bar{x}_s$, along the same lines as combining fixes in PFilter. It adopts the same workload partition strategy as PFilter. That is, the indices of each group of candidates processed by W_j satisfy the same equation presented there by treating $\mathcal{C}_h(u)$ as $\mathsf{Eq}[t_i]$. Like Match, it *incrementally* expands matches with newly added fixes. It then checks edge connections for homomorphism, and returns fixes from the qualified matches. For those complete matches $h(\bar{x}_s)$, only homomorphic checking is performed locally.

Algorithm. The main driver of PMend is shown in Fig. 6. After initializing distributed fixes and repair (line 1), it deduces fixes in supersteps following the precedence of Σ (lines 3-8). In each superstep, PFilter identifies partial matches (line 4) and PMatch completes those matches pertaining to designated V_d (line 5), *in parallel* at all workers. The new fixes Δ_{Eq} returned are used to extend Eq and NEq (line 6), and each worker mutates its fragment of G_{Eq} by identifying objects and repairing attributes accordingly (line 7). The set $\mathcal{A}$ of active GQRs is adjusted based on the new fixes as in GMend (line 7). If not all errors pertaining to V_d are fixed, it asks users to expand ground truth in the online mode, along the same lines as GMend (line 9); in the offline mode, user interaction is turned off. Finally, each worker returns its local copy of repair G_{Eq} (line 10).

Parallel scalability. To see that PMend is parallel scalable relative to GMend, we can show that PFilter and PMatch take $O(|\varphi||G|^{|\varphi|}/p)$ time for each GQR $\varphi \in \Sigma$. If it holds, by $p \ll |G|$, PMend is in $O(|\Sigma||G|^{|\Sigma|}/p) = O(t(|G|, |\Sigma|, |\Gamma|, |V_0|)/p)$ time; note that costs of updating Eq, NEq and G_{Eq} are much less in each superstep.

7. EXPERIMENTAL EVALUATION

Using real-life and synthetic graphs, we evaluated the accuracy, efficiency and (parallel) scalability of our methods for deducing certain fixes online (Section 7.1) and offline (Section 7.2). We also conducted a case study from industry to show the effectiveness of our proposed approaches (Section 7.3). We focus on fixing erroneous attributes and identifying duplicates, a type of common errors in graphs.

7.1 Online Mode

We start with the setting and results for the online mode.

Experimental setting. We used the following datasets.

Real-life graphs. We used two real-life graphs: (a) DBpedia [1], a knowledge graph with 28 million entities of 200 types and 33.4 million edges of 160 types; and (b) Yago2, an extended knowledge base of YAGO [46] with 2 million entities of 13 types and 5.7 million edges of 36 types.

GQRs. We discovered a set Σ of GQRs for each graph G by extending algorithms of [21]. Starting from frequent single-node GQRs, we find GQRs by interleaving vertical spawning to extend graph patterns Q, and horizontal spawning to generate attribute constraints $X \rightarrow Y$ with a generation tree. We selected GQRs of high support, *i.e.,* GQRs that can be frequently applied to G and capture the regularity. We mined 200 and 150 GQRs from DBpedia and Yago2, respectively, in which the graph patterns consists of at most 4 nodes and 8 edges, and the number of literals is at most 7. All GQRs in Σ were examined manually to guarantee correctness.

These GQRs detected 324 and 366 errors in DBpedia and Yago2, respectively, *e.g.,* a soccer player has multiple career records although the records associate with the same team.

We chose V_0 from vertices involved in errors detected by Σ. We resolved true attribute values and id's of vertices that violate Σ directly or indirectly. These validated attributes and entities were used as "gold standard" in our experiments.

Measurements. The accuracy of the algorithms is evaluated by precision, recall, and F-measure defined as $2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$. Here precision is the ratio of *true* fixes to all fixes derived; and recall is the ratio of errors correctly fixed to all the errors pertaining to V_0. Note that precision *is always* 100% for our methods by the nature of certain fixes, given that Σ and Γ are validated; but recall depends on how informative GQRs and ground truth are.

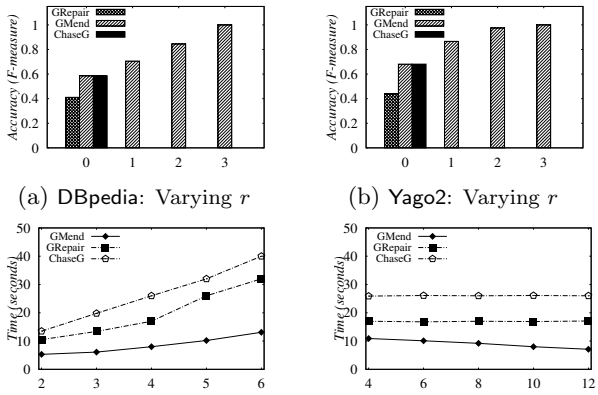
Algorithms. We implemented three sequential algorithms in Java: (1) GMend (Section 5); (2) GRepair of [12], in which we transformed each GQR into a GRR [12] retaining the semantics; and (2) ChaseG that implements the chase (Section 3) directly. The latter two do not allow user interaction.

We conducted the experiments on a single processor powered by Intel Xeon E5-2670v2 with 61 GB memory and 122 GB SSD storage. All the experiments (Sections 7.1 and 7.2) were run 3 times. The average is reported here.

Experimental results. We evaluated the accuracy and efficiency of the three algorithms, and the number r of user interaction rounds needed by algorithm GMend.

Accuracy. Fixing $|V_0| = 4$ and initial $|\Gamma| = 10$, the F-measure on DBpedia and Yago2 is reported in Figures 7(a) and 7(b), respectively. We can see that a large percentage of the errors can be correctly fixed by GMend and ChaseG without user involvement, *e.g.,* in round 0, the F-measure is 0.586 for DBpedia and 0.681 for Yago2, as opposed to 0.411 and 0.439 by GRepair, respectively. This is because GRepair applies GRRs indistinguishably, *i.e.,* it treats all the data surrounding V_0 as ground truth. As remarked earlier, a fix generated in this way may not be correct and worse yet, may introduce new inconsistencies. In contrast, both GMend and ChaseG compute certain fixes from validated ground truth.

Efficiency. Fixing $|\Gamma|$=10 (resp. $|V_0|$=4), we varied $|V_0|$ (resp. $|\Gamma|$) from 2 to 6 (resp. 4 to 12), to evaluate the impact



(a) DBpedia: Varying r (b) Yago2: Varying r

(c) DBpedia: Varying $|V_0|$ (d) DBpedia: Varying $|\Gamma|$

Figure 7: Performance of the online mode

of $|V_0|$ (resp. $|\Gamma|$) on algorithms over DBpedia. As shown in Figures 7(c) and 7(d), (1) GMend takes longer with larger $|V_0|$ or smaller $|\Gamma|$, due to more user interactions. (2) GMend needs 8s when $|V_0|$=4 and $|\Gamma|$=10 despite using a single machine. In contrast, GRepair and ChaseG take 17s and 26s, respectively. The results on Yago2 are consistent (not shown).

User intervention. At most three rounds of user interactions suffice to fix all errors related to V_0 when $|V_0| \leq 12$ (not shown). In each round, at most 8 attributes or matches require user inspection (see Section 5), and less is needed in later rounds. On average, Γ is expanded with 30 pieces of ground truth after fixing errors pertaining to each V_0, including fields validated by users and certain fixes generated.

7.2 Offline Mode

We next evaluated our method in the offline mode, including the accuracy, the effectiveness of combining data repairing and object identification in deducing certain fixes, and the efficiency and scalability of parallel algorithm PMend.

Experimental setting. In addition to two real-life graphs DBpedia and Yago2, we used larger synthetic graphs.

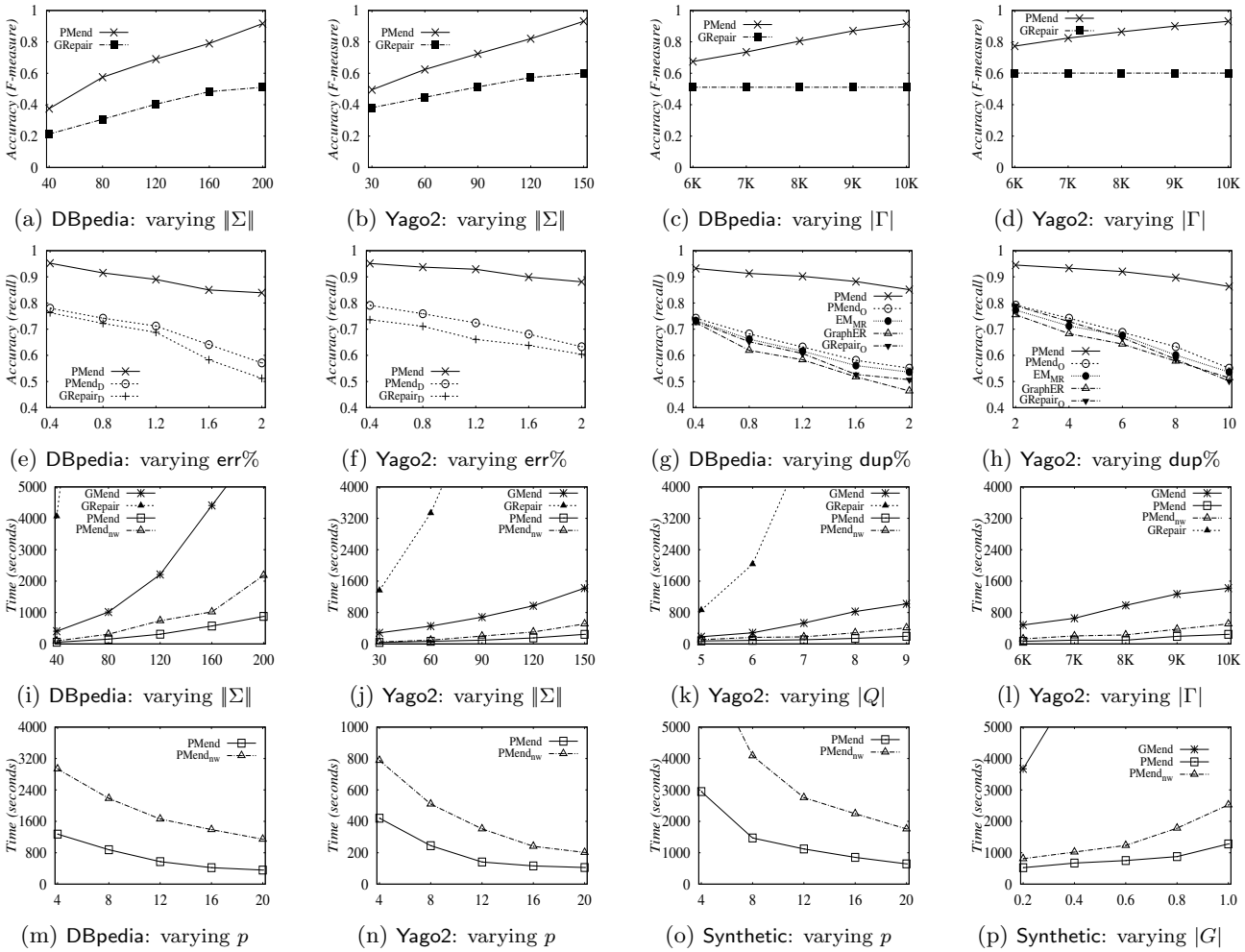
Synthetic graph. We designed a generator to produce synthetic graphs G, controlled by the number of nodes $|V|$ (up to 50 million) and the number of edges $|E|$ (up to 100 million), with labels L and attributes F_A drawn from an alphabet of 100 symbols. Each node in G is assigned 5 attributes with values drawn from a domain of 200 distinct constants.

We took G with 40 million nodes and 80 million edges as default. We also discovered a set Σ of 100 GQRs from synthetic graphs along the same lines as in Section 7.1.

Noises. To evaluate the accuracy and efficiency of PMend for fixing a large number of errors, apart from the errors caught in the real-life graphs (Section 7.1), we introduced noises to the graphs. More specifically, we randomly (a) updated values of the attributes that appear in GQRs Σ, controlled by *inconsistency rate* err%, the ratio of the number of updated attributes to the total number of such attributes; and (b) added duplicate entities, controlled by *duplicate rate* dup%, the percentage of duplicate entities in the entire graph.

Ground truth. We sampled a block Γ from the following. (1) Validated attributes and entities that are accumulated in the online mode when fixing violations of GQRs in the original graphs. (2) Attributes and matches that are unchanged by injected errors, without the need of manual checking. In both cases, the GQRs Σ and ground truth Γ are consistent.

Algorithms. In addition to the algorithms in Section 7.1, we implemented the following. (1) PMend (Section 6) and



(a) DBpedia: varying $\|\Sigma\|$ (b) Yago2: varying $\|\Sigma\|$ (c) DBpedia: varying $|\Gamma|$ (d) Yago2: varying $|\Gamma|$

(e) DBpedia: varying err% (f) Yago2: varying err% (g) DBpedia: varying dup% (h) Yago2: varying dup%

(i) DBpedia: varying $\|\Sigma\|$ (j) Yago2: varying $\|\Sigma\|$ (k) Yago2: varying $|Q|$ (l) Yago2: varying $|\Gamma|$

(m) DBpedia: varying p (n) Yago2: varying p (o) Synthetic: varying p (p) Synthetic: varying $|G|$

Figure 8: Performance of the offline mode

three variants: (a) $\mathsf{PMend_D}$, which applies GFDs to correct attribute values only; (b) $\mathsf{PMend_O}$, which only identifies entities by using keys; and (c) $\mathsf{PMend_{nw}}$ that uses random workload partition. (2) The entity matching algorithm $\mathsf{EM_{MR}}$ of [16] and an entity resolution algorithm GraphER by graph clustering [8]. (3) Two variants $\mathsf{GRepair_D}$ and $\mathsf{GRepair_O}$ of sequential GRepair [12] for data repairing and object identification, respectively, adopting the same strategy as in (1).

The algorithms were deployed on a cluster of up to 20 processors; each had the same setting as that in Section 7.1.

Experimental results. We next report our findings.

Exp-1: Effectiveness. We first evaluated the quality of the fixes derived by PMend versus the repairing method GRepair. We conducted the experiments on real-life graphs and fixed err% = 2%, dup% = 2%, $|\Gamma| = 10K$ and used the entire set Σ of GQRs unless stated otherwise. We found $16.8K$ and $9.5K$ errors in total in the two graphs using Σ. Among these, $7.1K$ and $3K$ involved duplicates, and $9.7K$ and $6.5K$ involved inconsistent values, respectively.

Varying $\|\Sigma\|$. We varied $\|\Sigma\|$, the number of GQRs, from 40 to 200 over DBpedia and from 30 to 150 over Yago2. As shown in Figures 8(a)-8(b), (1) the more GQRs are available, the higher F-measure gets by both methods, as expected; and (2) PMend consistently outperforms GRepair in quality by 58.3% on average, and the improvement becomes more substantial with the increase of $\|\Sigma\|$ for the same reason as given in Section 7.1, *i.e.,* GRepair outputs more erroneous fixes when more GQRs (GRRs) are available for its indistinguishable enforcement of GRRs to unvalidated part of the graphs.

Varying $|\Gamma|$. Varying $|\Gamma|$ from $6K$ to $10K$, we report results on DBpedia and Yago2 in Figures 8(c) and 8(d), respectively. As shown there, PMend does better when given a larger block of ground truth, as expected. In contrast, GRepair is indifferent to $|\Gamma|$ for the same reason given above.

Exp-2: Interleaving attribute repairing and object identification. We next tested the accuracy (recall) of PMend against different methods for (1) graph repairing only and (2) object identification (duplicate detection) only. For repairing attributes only, recall is the ratio of attributes correctly fixed to all the erroneous attributes in the graph, without considering duplicates; similarly we specialize recall for object identification only. We used the same real-life graphs and ground truth Γ as in Exp-1, and all the GQRs discovered.

Deduplication helps repairing. Fixing the duplicate rate dup% = 2%, we varied the inconsistency rate err% from 0.4% to 2%. The results on DBpedia and Yago2 are reported in Figures 8(e) and 8(f), respectively. We find the following. (a) PMend consistently outperforms $\mathsf{PMend_D}$ and $\mathsf{GRepair_D}$; it is up to 63.7% more accurate in correcting attributes. Thus object identification indeed helps attribute repairing. (b) The accuracy (recall) of the three algorithms decreases when err% grows, as expected. However, PMend is

less sensitive to the increase of err%. This also benefits from the combination of repairing and deduplication since true attribute values can be taken from the matching entities.

Repairing helps deduplication. Fixing err% = 2%, we varied dup% from 0.4% to 2%. As shown in Figures 8(g) and 8(h) on DBpedia and Yago2, respectively, (a) algorithm PMend outperforms PMend$_O$, EM$_{MR}$, GraphER, and GRepair$_O$ by 38.6%, 42.7%, 51.2% and 46.2% on average in matching entities, respectively. In addition, EM$_{MR}$, GraphER and GRepair$_O$ identify a number of false positives based on incorrect attribute values (not shown), although EM$_{MR}$ and GraphER are designed for entity matching. (b) The larger dup% is, the less accurate is for each algorithm. However, the accuracy gaps between PMend and others get larger with the increase of dup%, *i.e.,* more entities are matched using the repaired attribute values. These verify that data repairing helps object identification in real-life graphs.

Exp-3: Efficiency and scalability. Finally, we evaluated the efficiency and (parallel) scalability of the algorithms in cleaning entire graphs. We used the same default settings as in Exp-1 and $p = 8$ processors unless stated otherwise.

We do not report ChaseG as it is sequential and far slower than the others for its redundant enforcement of GQRs. It could *not* run to completion within 2.5 hours in most cases.

Impact of $\|\Sigma\|$. Varying $\|\Sigma\|$ in the same way as in Exp-1, we evaluated the impact of GQRs on the efficiency. We find the following from Figs. 8(i) and 8(j) on DBpedia and Yago2, respectively. (1) All algorithms take longer to process more GQRs, as expected. (2) PMend does the best in all cases, and is on average 6.9 times faster than the sequential GMend (without user interaction). In contrast, GRepair does not terminate in 100 minutes when applying 90 GQRs on Yago2 since GRepair does not support parallel processing.

Impact of $|Q|$. Fixing $\|\Sigma\| = 80$, we varied the average size $|Q|$ of graph patterns in Σ from 5 to 9 over Yago2. Results in Fig. 8(k) show that all algorithms take longer to process GQRs with more pattern nodes and edges. We also find that they are less sensitive to the number of literals (not shown).

Impact of $|\Gamma|$. We varied $|\Gamma|$ from $6K$ to $10K$ on Yago2. The result in Fig. 8(l) shows that (1) the larger $|\Gamma|$ is, the longer time is taken by algorithms GMend, PMend and PMend$_{nw}$. (2) Again, PMend outperforms the others, while GRepair does not terminate within 100 minutes in all the cases.

Parallel scalability. We evaluated the parallel scalability of the algorithms by varying the number p of processors from 4 to 20. As shown in Fig. 8(m) over DBpedia (resp. 8(n), 8(o) over Yago2, Synthetic), (1) PMend scales well with p: the improvement is 3.6 (resp. 4.0 and 4.2) times when p increases from 4 to 20; this validates the parallel scalability of algorithm PMend. (2) PMend outperforms PMend$_{nw}$ by 3.02 times on average, verifying the effectiveness of our workload partition strategy. (3) Algorithm PMend works well on real-life graphs. For example, it needs less than 6 minutes to process DBpedia over 20 processors, while none of GMend, GRepair and ChaseG completes within 2 hours.

Impact of $|G|$. Using synthetic graphs G, we varied $|G|$ with scale factors from 0.2 to 1 and used 16 processors. Figure 8(p) tells us the following. (1) Parallel algorithm PMend scales well with $|G|$ and is feasible on large graphs despite the exponential theoretical cost. It takes 877 seconds when G has 40 million nodes and 80 million edges, as opposed to more than 8000 seconds taken by GMend and GRepair. (2) PMend performs better than its counterpart PMend$_{nw}$, which is consistent with Figures 8(i) to 8(o).

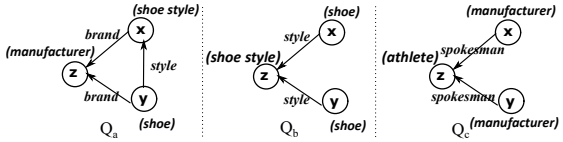


Figure 9: Real-life GQRs

7.3 Case Study

In the last decade, e-commence companies, *e.g.,* Alibaba [53] and Amazon [15] have built product graphs to answer questions about products and their related knowledge. For instance, the shoes of brand Under Armour should be returned when a customer searches for "Stephen Curry Shoes", since Curry is a spokesman of Under Armour.

Model GFix can be used to clean such graphs. Consider three GQRs with patterns shown in Fig. 9: (1) $\varphi_a = Q_a[\bar{x}](\emptyset \rightarrow x.\mathsf{upper} = y.\mathsf{upper})$, saying that if shoe y has shoe style x and both x and y refer to manufacturer z, then x and y have the same upper; (2) $\varphi_b = Q_b[\bar{x}](x.\mathsf{color} = y.\mathsf{color}, x.\mathsf{size} = y.\mathsf{size} \rightarrow x.\mathsf{id} = y.\mathsf{id})$, *i.e.,* two shoes can be identified by their colors and sizes if they have the same style; and (3) $\varphi_c = Q_c[\bar{x}](\emptyset \rightarrow x.\mathsf{id} = y.\mathsf{id})$, *i.e.,* two manufacturers can be identified if they have the same athlete as the spokesman.

When a new manufacturer or shoe style is added to the product graph, it is put into V_0, and the GQRs above are enforced in the online mode to fix errors related to the uppers of shoes, and to remove those duplicate shoes and manufacturers that are possibly created by different sellers. In the product graph, entities related to shoes typically have common structures and attributes, *e.g.,* GQR φ_a still applies for other properties in shoe styles besides upper.

The users of GFix in the online mode are typically people in charge of maintaining the product graph. They only need to manually check a limited number of values for the shoe styles, and are knowledgeable enough to validate the styles and manufactures of shoes. The changes made by them are also periodically inspected by domain experts. The validated changes are included in Γ as ground truth.

Using the accumulated ground truth, the offline mode is invoked periodically to "clean" the whole set of shoes. This process can be applied to other products analogously.

8. CONCLUSION

We have made a first effort to clean graphs with certainty, by proposing GFix. We have extended GEDs [24] to express positive and negative rules. We have settled the complexity of fundamental problems for deducing certain fixes. We have developed (parallel scalable) algorithms underlying GFix. Our experiments have verified that the method is promising.

One topic for future work is to fix errors in connection with invalid and missing edges, which introduce challenges to rule-based method since it is hard for the graph patterns in the rules to simultaneously catch such edge errors.

Acknowledgments. The authors are supported in part by ERC 652976, 973 Program 2014CB340302, NSFC 61421003, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Lu is also supported in part by NSFC 61602023.

9. REFERENCES

[1] DBpedia. *http://wiki.dbpedia.org*.

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

[3] W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in RDF. In *SDKB*, 2010.

[4] Y. Altowim and S. Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, 2017.

[5] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[6] A. Arioua and A. Bonifati. User-guided repairing of inconsistent knowledge bases. In *EDBT*, 2018.

[7] A. Assadi, T. Milo, and S. Novgorodov. DANCE: data cleaning with constraints and experts. In *ICDE*, 2017.

[8] I. Bhattacharya and L. Getoor. Entity resolution in graphs. *Mining graph data*, 2006.

[9] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[10] S. R. Buss and L. Hay. On truth-table reducibility to SAT. *Inf. Comput.*, 91(1):86–102, 1991.

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

[12] Y. Cheng, L. Chen, Y. Yuan, and G. Wang. Rule-based graph repairing: Semantic and efficient repairing methods. In *ICDE*, 2018.

[13] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[14] A. Cortés-Calabuig and J. Paredaens. Semantics of constraints in RDFS. In *AMW*, 2012.

[15] X. L. Dong. Challenges and innovations in building a product knowledge graph. In *KDD*, 2018.

[16] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *PVLDB*, 8(12):1590–1601, 2015.

[17] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

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

[19] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2):6:1–6:48, 2008.

[20] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *ICDE*, 2010.

[21] W. Fan, C. Hu, X. Liu, and P. Lu. Discovering graph functional dependencies. In *SIGMOD*, 2018.

[22] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[23] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2):213–238, 2012.

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

[25] W. Fan, X. Wang, and Y. Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 7(12):1083–1094, 2014.

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

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

[28] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[29] S. Hao, N. Tang, G. Li, and J. Li. Cleaning relations using knowledge bases. In *ICDE*, 2017.

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

[31] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, 2016.

[32] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, 2008.

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

[34] L. Kolb, A. Thor, and E. Rahm. Load balancing for MapReduce-based entity resolution. In *ICDE*, 2012.

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

[36] G. Lausen, M. Meier, and M. Schmidt. SPARQLing constraints for RDF. In *EDBT*, 2008.

[37] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[38] C. Papadimitriou and S. Zachos. Two remarks on the power of counting. *TCS*, pages 269–275, 1982.

[39] T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *TKDE*, 27(5):1316–1329, 2015.

[40] N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining quantitative and logical data cleaning. *PVLDB*, 9(4):300–311, 2015.

[41] V. Rastogi, N. N. Dalvi, and M. N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[42] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[43] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Bießmann, and A. Grafberger. Automating large-scale data quality verification. *PVLDB*, 11(12):1781–1794, 2018.

[44] S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 7(11):987–998, 2014.

[45] H. Spakowski. *Completeness for parallel access to NP and counting class separations*. PhD thesis, University of Düsseldorf, Germany, 2005.

[46] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A core of semantic knowledge. In *WWW*, 2007.

[47] V. Verroios, H. Garcia-Molina, and Y. Papakonstantinou. Waldo: An adaptive human interface for crowd entity resolution. In *SIGMOD*, 2017.

[48] K. W. Wagner. Bounded query classes. *SICOMP*, 19(5):833–846, 1990.

[49] S. E. Whang and H. Garcia-Molina. Joint entity resolution on multiple datasets. *VLDB J.*, 22(6):773–795, 2013.

[50] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

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

[52] G. Zhang, D. Jimenez, and C. Li. Maverick: Discovering exceptional facts from knowledge graphs. In *SIGMOD*, 2018.

[53] C. Zhou, Y. Liu, X. Liu, Z. Liu, and J. Gao. Scalable graph embedding for asymmetric proximity. In *AAAI*, 2017.