THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

Towards Effective Analysis of Big Graphs: From Scalability to Quality

Chao Tian

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2017

Abstract

This thesis investigates the central issues underlying graph analysis, namely, scalability and quality.

We first study the incremental problems for graph queries, which aim to compute the changes to the old query answer, in response to the updates to the input graph. The incremental problem is called bounded if its cost is decided by the sizes of the query and the changes only. No matter how desirable, however, our first results are negative: for common graph queries such as graph traversal, connectivity, keyword search and pattern matching, their incremental problems are unbounded. In light of the negative results, we propose two new characterizations for the effectiveness of incremental computation, and show that the incremental computations above can still be effectively conducted, by either reducing the computations on big graphs to small data, or incrementalizing batch algorithms by minimizing unnecessary recomputation.

We next study the problems with regards to improving the quality of the graphs. To uniquely identify entities represented by vertices in a graph, we propose a class of keys that are recursively defined in terms of graph patterns, and are interpreted with subgraph isomorphism. As an application, we study the entity matching problem, which is to find all pairs of entities in a graph that are identified by a given set of keys. Although the problem is proved to be intractable, and cannot be parallelized in logarithmic rounds, we provide two parallel scalable algorithms for it.

In addition, to catch numeric inconsistencies in real-life graphs, we extend graph functional dependencies with linear arithmetic expressions and comparison predicates, referred to as NGDs. Indeed, NGDs strike a balance between expressivity and complexity, since if we allow non-linear arithmetic expressions, even of degree at most 2, the satisfiability and implication problems become undecidable. A localizable incremental algorithm is developed to detect errors using NGDs, where the cost is determined by small neighbors of nodes in the updates instead of the entire graph.

Finally, a rule-based method to clean graphs is proposed. We extend graph entity dependencies (GEDs) as data quality rules. Given a graph, a set of GEDs and a block of ground truth, we fix violations of GEDs in the graph by combining data repairing and object identification. The method finds certain fixes to errors detected by GEDs, *i.e.,* as long as the GEDs and the ground truth are correct, the fixes are assured correct as their logical consequences. Several fundamental results underlying the method are established, and an algorithm is developed to implement the method. We also parallelize the method and guarantee to reduce its running time with the increase of processors.

Lay Summary

Graphs are becoming more and more pervasive in various domains. As an example, it has been recognized that social network and knowledge base analysis should be incorporated into search engines, whose principal goal has been to help people find what they are looking for. Social networks produce an immense amount of data about what people like and knowledge bases provide semantic relations among objects. It is natural to improve searches by capitalizing on these graph-structured data. However, to incorporate graph data into search engines, techniques must be in place to address the following.

(1) How to efficiently conduct graph analysis? Social graphs typically have millions of nodes and billions of edges. Moreover, graph queries are expensive on large graphs, and worse still, real-life graphs are constantly changed. These call for revisions of graph querying and new techniques to cope with the sheer size of graph data.

(2) How to improve the quality of search results? Search engines often return irrelevant, duplicated or spam results. The scale of the problem is already severe, and the incorporation of graph data would only make it worse if the graphs are dirty themselves. These highlight the need for new techniques to improve the quality of graphs.

In response to the need, in this thesis, we develop a set of techniques in the area of graph data management. More specifically, we investigate the effectiveness of incremental graph computations, which helps to reduce computations on possibly big graph to small changes. In addition, we propose a class of keys for graphs. Extending conventional keys for relations and XML, these keys find applications in object identification, knowledge fusion and social network reconciliation. Furthermore, another class of graph dependencies is studied, referred to as NGDs, to catch semantic inconsistencies with numeric values involved. Indeed, such numeric errors are commonly found in knowledge bases and social networks. We finally present a technique to clean graphs, *i.e.,* fixing erroneous attribute values and resolving duplicate entities, by employing a set of data quality rules defined on graphs.

Acknowledgements

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chao Tian*)

Table of Contents

Chapter 1

Introduction

It has been recognised that graphs are an important source of big data, and the quest for analyzing graphs arises from different aspects, *e.g.,* knowledge discovery, transportation networks, mobile networks, social marketing, computer vision, and intelligence analysis. To make practical use of big graphs, however, we have to cope with (1) their quantity (volume), and (2) their quality (velocity) as well. In light of these, this thesis investigates various issues in analyzing graphs, ranging from querying big graphs to improving data quality.

1.1   Background

(1) Incremental graph computations. For a class Q of graph queries, *the incremental problem* aims to find an algorithm $\mathcal{T}_{\Delta}$ that, given a query $Q \in Q$, a graph G, query answers $Q(G)$ and updates ΔG to G as input, computes changes ΔO to $Q(G)$ such that

$$Q(G \oplus \Delta G) = Q(G) \oplus \Delta O.$$

Here $S \oplus \Delta S$ denotes applying updates ΔS to S, when S is either graph G or query result $Q(G)$. That is, $\mathcal{T}_{\Delta}$ answers Q in response to ΔG by computing changes to the (old) output $Q(G)$. We refer to $\mathcal{T}_{\Delta}$ as an *incremental algorithm* for Q, in contrast to *batch algorithms* $\mathcal{T}$ that given Q, G and ΔG, recompute $Q(G \oplus \Delta G)$ starting from scratch.

The need for incremental computations is evident. Real-life graphs G are often big, *e.g.,* the social graph of Facebook has billions of nodes and trillions of edges [GBDS14]. Graph queries are expensive, *e.g.,* subgraph isomorphism is NP-complete (cf. [Pap94]). Moreover, real-life graphs are constantly changed. It is often too costly to recompute $Q(G \oplus \Delta G)$ starting from scratch in response to frequent ΔG.

These highlight the need for incremental algorithms $\mathcal{T}_\Delta$: we use a batch algorithm $\mathcal{T}$ to compute $Q(G)$ once, and then employ incremental $\mathcal{T}_\Delta$ to compute changes ΔO to $Q(G)$ in response to ΔG. The rationale behind this is that in the real world, changes are typically small, *e.g.,* less than 5% on the entire Web in a week [NCO04]. When ΔG is small, ΔO is often also small, and is much less costly to compute than $Q(G \oplus \Delta G)$, by making use of previous computation $Q(G)$. In addition, incremental computations are crucial to parallel query processing [FWW14a] that partitions a big G, partially evaluates queries on the fragments at different processors, treats messages among the processors as *updates*, and conducts iterative computations incrementally to reduce the cost. But can we ensure that the incremental $\mathcal{T}_\Delta$ is more efficient than the batch $\mathcal{T}$?

(2) Identifying entities. Keys provide an invariant connection between a real-world entity and its representation in a database. They are fundamental to relational databases: data models, conceptual design, and prevention of update anomalies. They are found in almost every database textbook. Keys have also been extensively studied for XML (*e.g.,* [BDF$^+$01]), and are part of XML Schema (W3C). They are instrumental in XML data transformation (publishing, shredding) and cleaning.

For all the reasons that keys are essential to relations and XML, keys are also needed for graphs in identifying entities. The need is evident when relations are represented as graphs [ARS09, BG07, RDG11, MAS14], and for citations of "digital objects" of graph structures [BS10]. They are also important to emerging applications such as knowledge fusion and knowledge base expansion [DGH$^+$14, DMG$^+$14, PKS$^+$10], to deduplicate entities and to fuse information from different sources that refers to the same entity. Another application is social network reconciliation, to reconcile user accounts across multiple social networks [KL14]. However, keys for graphs are more challenging than conventional keys.

(3) Catching and fixing inconsistencies. A variety of dependencies have recently been studied for graphs [LMS08, FWX16, CP12, ACP10, YH11, HZZ14, FL17, CFP$^+$14]. These dependencies are often defined in terms of graph patterns, and aim to capture inconsistencies among entities in a graph. They are useful in, *e.g.,* knowledge acquisition, knowledge base enrichment, and spam detection in social networks.

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

In addition, although these dependencies can detect errors commonly found in real-

life graphs, they do not tell us how to fix the errors.

1.2 Contributions and Thesis Organization

The following contributes are made in this thesis.

- The effectiveness of incremental graph computations is studied in Chapter 2.
  - We show that no bounded incremental algorithms exist for RPQ (regular path queries), SCC (strongly connected components), and KWS (keyword search) (Section 2.2), *i.e.,* their costs cannot be expressed as polynomial functions in the sizes of the queries and the changes to the inputs and outputs. We establish these impossibility results either by elementary proofs or by reductions from incremental graph problems that are already known unbounded. To the best of our knowledge, this work gives the first proofs by reductions for unbounded graph incremental computations.
  - We characterize localizable incremental computations and relative boundedness in Sections 2.3 and 2.4, respectively. We show that the incremental computations above are either localizable (KWS and ISO (subgraph isomorphism)) or relatively bounded (RPQ and SCC). That is, while these incremental computations are unbounded, they can still be effectively conducted with performance guarantees.
  - As a proof of concept, we develop localized incremental algorithms for KWS and ISO (Section 2.3), and bounded incremental algorithms for RPQ and SCC relative to their batch algorithms (Section 2.4). We also develop optimization techniques for processing batch updates. These extend the small library of existing incremental graph algorithms that have performance guarantees.
- Chapter 3 investigates keys for graphs, from specifications and semantics to applications.
  - We propose a class of keys for graphs (Section 3.1). We define keys in terms of *graph patterns*, to specify topological constraints and value bindings needed for identifying entities. Moreover, keys may be *recursively defined*: to identify a pair of entities, we may need to decide whether some other entities can be identified. We interpret keys by means of graph pattern matching via subgraph isomorphism. These make such keys more expressive than our familiar keys for relations and XML.

– We study *entity matching*, an essential application of keys for graphs (Section 3.2). Given a graph G and a set Σ of keys for graphs, *entity matching* is to find all pairs of entities (vertices) in G that can be identified by keys in Σ. We formalize the problem by revising the chase [AHV95] studied in the classical dependency theory. While entity matching is in PTIME (polynomial time) for relations and XML with traditional keys, we show that its decision problem is NP-complete for graphs. Worse still, recursively defined keys pose new challenges. We show that entity matching does not have the polynomial-fringe property (PFP) [ABC$^+$11], and cannot be solved in logarithmic parallel computation rounds. Nonetheless, we show that entity matching is within reach in practice, by providing parallel scalable algorithms.

– We develop a MapReduce algorithm for entity matching (Section 3.3). As opposed to subgraph isomorphism, entity matching with recursively defined keys requires a fixpoint computation, and in each round, multiple isomorphism checking for each entity pair. We show that the algorithm is *parallel scalable*, *i.e.,* its worst-case time complexity is $O(t(|G|,|\Sigma|)/p)$, where $t(,)$ is a function in $|G|$ and $|\Sigma|$, and p is the number of processors used. It guarantees to take proportionally less time with the increase of p, which is *not* warranted by many parallel algorithms. We also develop optimization methods to process recursively defined keys.

– We give another algorithm in the vertex-centric asynchronous model of [LGK$^+$12] (Section 3.4). This algorithm not only checks different entity pairs in parallel, but also inspects different mappings in parallel when checking each entity pair, via asynchronous message passing. It reduces unnecessary costs inherent to the I/O bound and the synchronization policy ("blocking" of stragglers) of MapReduce. We show that the algorithm is also parallel scalable. Moreover, we propose optimization techniques to reduce message passing.

• The numeric graph dependency is introduced in Chapter 4, for detecting numeric errors.

– We propose a class of numeric graph dependencies, referred to as NGDs (Section 4.2). NGDs are a combination of (a) a pattern Q to identify entities by graph homomorphism, and (b) an attribute dependency $X \rightarrow Y$ on the entities identified. They extend graph functional dependencies

(GFDs [FWX16, FL17]) by supporting linear arithmetic expressions and built-in comparison predicates $=, \neq, <, \leq, >, \geq$. We show that NGDs are able to catch numeric inconsistencies commonly found in real-life graphs. Moreover, they subsume GFDs [FWX16, FL17] and relational conditional functional dependencies (CFDs [FGJK08]) as special cases. Thus they are able to capture all inconsistencies that can be detected by GFDs and CFDs, besides numeric errors that are beyond the capacity of GFDs and CFDs.

– We study two classical problems for reasoning about NGDs (Section 4.3), in which the *satisfiability* problem is to decide whether a given set Σ of NGDs has a model *i.e.,* a graph satisfying Σ, and the *implication* problem is to decide whether a set Σ of NGDs entails another NGD φ, *i.e.,* for all graphs G that satisfy Σ, G also satisfies φ. (a) We show that the increased expressive power of NGDs comes with a price. Their satisfiability and implication problems become Σ_2^p-complete and Π_2^p-complete, as opposed to coNP-complete and NP-complete for GFDs, respectively [FWX16, FL17]. The complexity bounds are robust: they remain Σ_2^p-hard and Π_2^p-hard, respectively, even when only equality $=$ is used, in the absence of $\neq, <, \leq, >, \geq$, or when no arithmetic operations are used at all. These tell us that unless P = NP, it is harder to reason about NGDs than about GFDs. (b) We also show that if we expand NGDs by allowing non-linear arithmetic expressions, then both problems become undecidable, even when the degree of the arithmetic expressions is at most 2, and even in the absence of comparison predicates $\neq, <, \leq, >, \geq$. The undecidability results justify the choice of linear arithmetic expressions. That is, NGDs strike a balance between expressivity and complexity when arithmetic and comparison are a must.

– We develop techniques for detecting inconsistencies in real-life graphs, numeric or not, by employing NGDs as data quality rules (Sections 4.4 and 4.5). (a) We show that the *validation problem* is coNP-complete for NGDs, to decide whether a given graph satisfies a set of NGDs. The complexity is the same as for GFDs [FWX16, FL17]. That is, NGDs do not complicate the process of error detection. Better still, the parallel algorithms developed in [FWX16] for detecting errors with GFDs can be readily extended to NGDs, retaining the same complexity. (b) In light of this, we focus on incremental inconsistency detection in graphs, a problem that has not been studied by previous work, to the best of our knowledge (Section 4.4).

Given a graph G and a set Σ of NGDs, suppose that we have already identified a set $\mathsf{Vio}(\Sigma, G)$ of violations of Σ in G, *i.e.,* entities in G that violate at least one NGD in Σ. We want to find *changes* $\Delta\mathsf{Vio}$ to $\mathsf{Vio}(\Sigma, G)$, such that $\mathsf{Vio}(\Sigma, G \oplus \Delta G) = \mathsf{Vio}(\Sigma, G) \oplus \Delta\mathsf{Vio}$, where $X \oplus \Delta X$ denotes X updated by ΔX. Thus we can use (an extension of) the batch algorithms of [FWX16] to compute $\mathsf{Vio}(\Sigma, G)$ once, and then incrementally compute changes $\Delta\mathsf{Vio}$ in response to ΔG. (c) While desirable, the incremental detection problem is nontrivial. We show that the problem is also coNP-complete, even when both graphs G and updates ΔG have *constant sizes* (Section 4.4).

– In response to the practical need, we develop two algorithms for incremental error detection with NGDs (Section 4.5), which make incremental error detection feasible in large-scale graphs. One is a sequential *localizable* algorithm IncDect. It incrementalizes subgraph search by *update-driven evaluation*. Its cost is determined by the d_Σ-neighbors of nodes in ΔG, where d_Σ is the maximum diameter of the patterns in Σ [FHT17]. In practice, Σ is much smaller than G, and so is d_Σ. It reduces the computations on a (possibly big) graph G to smaller d_Σ-neighbors of those nodes in ΔG. The other one is a parallel algorithm PIncDect. We show that it is *parallel scalable relative to* IncDect: its cost is $O(t(|G|, |\Sigma|, |\Delta G|)/p)$, where p is the number of processors used, and $t(|G|, |\Sigma|, |\Delta G|)$ is the cost of IncDect. That is, PIncDect guarantees to reduce running time when more processors are used. We propose a *hybrid strategy* to split skewed work units and dynamically balance workload, based on cost estimation, to balance computation and communication.

• Finally, Chapter 5 proposes a rule-based method, referred to as Analogist, to generate certain fixes to semantic inconsistencies in graphs, from fundamental results to practical algorithms.

– We propose a class of data quality rules (Section 5.1). We extend graph entity dependencies (GEDs) of [FL17] such that we can simultaneously (a) repair data, *i.e.,* fix attribute values, by using graph functional dependencies [FWX16] along the same lines as conditional functional dependencies (CFDs [FGJK08]) for repairing relations, (b) identify objects, *i.e.,* determine whether two vertices in a graph refer to the same entity, by using recursively defined keys of Chapter 3, and moreover, (c) deduce entities that do not match to reduce false positives, by adding a form of forbidding

constraints with inequality.

– Employing GEDs, we propose Analogist, a method to clean graphs with certain fixes (Section 5.2). Given a graph G, a set Σ of GEDs and a block Γ of ground truth (confirmed attribute values and entity matches), Analogist finds fixes to the violations of Σ in G by chasing G with (Σ, Γ). We show that Analogist is Church-Rosser, *i.e.,* the chase converges at the same set of fixes regardless of the order of GEDs applied. Moreover, its fixes are *certain*, as logical consequences of Σ and Γ. That is, the fixes are assured correct as long as the rules of Σ and ground truth of Γ are correct. It integrates data repairing and object identification in the same process. It propagates changes to correlated and co-occurred entities in G.

– We settle three fundamental problems for graph cleaning with certain fixes (Section 5.3). (a) The *consistency problem* is to determine whether Σ and Γ have no conflict, *i.e.,* the set of rules and ground truth are not dirty themselves. (b) The *certain fix problem* is to decide whether a fix can be found by the chase. (c) The *coverage problem* is to decide whether Σ and Γ suffice to fix all violations of Σ in G, and yield a unique repaired graph that satisfies Σ. We establish the combined complexity and data complexity of these problems, ranging over PTIME (polynomial time), coNP-complete, and NP-complete and $P_{\parallel}^{NP}$-complete, comparable to or slightly harder than their counterparts for cleaning relations with certain fixes [FLM^{+}12].

– We develop an algorithm Clean to implement Analogist (Section 5.4). While the chase is Church-Rosser, the order of rules applied has big impact on the efficiency of the method. We propose two strategies to make the method practical: (a) precedence graphs on GEDs to determine the order of rules applied, and (b) incremental expansion of certain fixes. These substantially reduce redundant computations, notably the costly graph homomorphism checking.

– We develop a parallel algorithm PClean to clean large-scale graphs with certainty (Section 5.5). We show that PClean is *parallel scalable relative to* Clean [KRS90]. It adopts a workload partition strategy to evenly distribute the work of each chase step across available processors. Hence the method is able to scale with large real-life graphs G, since we can add more processors when G grows big.

Remark. It is worth mentioning that (partial) results in Chapter 2 have been published in [FHT17], and results in Chapter 3 are published in [FFTD15]. The results in Chapters 4 and 5 are taken from two submitted papers under review.

1.3   State of the Art

The proposed work in this thesis is fundamentally different from previous approaches, which are categorized as follows.

1.3.1   Graph Computations and View Maintenance

Bounded incremental algorithms.   Proposed in [TR81], the notion was studied for graph algorithms in [RR96a, RR96b, FWW13]. A number of incremental algorithms have been developed for graphs [ZGM98, RR96a, RR96b, Sah07, BCN08, SNS09, FWW13, RZ04, HK97, Lac13, HKM$^+$12] (see [DEGI10] for a survey). However, their costs are typically studied in terms of *amortized* analysis for averaged operation time of a sequence of unit updates to G, not in the size of changes that is inherent to the incremental problem itself. To the best of our knowledge, bounded algorithms are only in place for the shortest path problems, single-source or all pairs, with positive lengths [RR96a, RR96b]. It is known that the incremental problem is unbounded for subgraph isomorphism ISO [FWW13], and for single-source reachability to all vertices (SSRP under unit edge deletions, but bounded under unit insertions) [RR96a].

As the notion of boundedness is often too strong, a weaker standard was introduced in [FWW13], based on a notion of affected area $\mathsf{AFF}^\forall$. Intuitively, $\mathsf{AFF}^\forall$ covers not only changes ΔO, but also data that is necessarily checked to detect ΔO by *all* incremental algorithms for Q, encoded in auxiliary structures. An incremental algorithm is *semi-bounded* [FWW13] if (a) its cost can be expressed as a polynomial in $|\mathsf{AFF}^\forall|$, $|Q|$ and $|\Delta G|$, and (b) the size of the auxiliary structure is bounded by a polynomial in $|G|$. The incremental problem for graph simulation is shown semi-bounded [FWW13].

The work in Chapter 2 differs from the prior work in the following. (a) We establish new unboundedness results for RPQ, SCC and KWS, and a new form of reductions as proof techniques. (b) We propose measures for the effectiveness of incremental graph algorithms. In contrast to [RR96a, RR96b, FWW13], localizable algorithms are characterized by d_Q-neighbors of ΔG instead of ΔO or $\mathsf{AFF}^\forall$. Relative boundedness is defined in terms of the affected area AFF relative to a specific algorithm $\mathcal{T}$, as opposed

to AFF$^\forall$ for *all* incremental algorithms for Q (semi-boundedness). (c) We develop incremental algorithms for RPQ, SCC, KWS and ISO with performance guarantees under the new measures, although they are unbounded.

Locality of graph computations. There have been batch algorithms that capitalize on the data locality of queries, for (parallel) subgraph isomorphism (*e.g.,* [FWW14b, FWX16]). Incoop [BWR$^+$11], a generic MapReduce framework for incremental computations, also makes use of the locality of previously computed results in its scheduling algorithm to prevent straggling. To the best of our knowledge, the study of localizable incremental algorithms in Chapters 2 and 4 is the first effort to characterize the effectiveness of incremental algorithms in terms of locality.

Relative boundedness. There has also been work on incrementalizing batch algorithms, notably self-adjusting computations [Aca05, Bha15]. The idea is to track the dependencies between data and function calls as a dynamic dependency graph [ABH02], upon which functions that are affected by the changes in the input can be identified and recomputed. Memorization [PT89] is used to record and reuse the results of function calls when possible. It is a general-purpose, language-centric technique for programs to automatically respond to modifications to their data. In contrast, relative boundedness (Chapter 2) is to characterize whether it is *feasible* to incrementalize a given batch algorithm $\mathcal{T}$ with cost measured in the size of affected area AFF inspected by $\mathcal{T}$, not in terms of function calls.

View maintenance. Related is also view maintenance for updating materialized views, which has been studied for relational data [GMS93, GJM96, CGL$^+$96], object-oriented databases [KR98], and semi-structured data modeled as graph [ZGM98, AMR$^+$98]. Various methods have been proposed, *e.g.,* an algebraic approach of [BGMS13] for XML views and the use of key constraints [GJM96] for relations. However, few of them have provable performance guarantees, and fewer can be applied to graph queries. In particular, the techniques of [ZGM98, AMR$^+$98] are developed for views specified as selection paths, and do not apply to graph queries studied in this thesis. In contrast, in Chapter 2, we study the boundedness of incremental graph problems and provide algorithms that are localizable or relatively bounded.

1.3.2   Data Dependencies

Keys. Relational keys are defined over a relation schema in terms of a set of attributes [AHV95]. XML keys are specified in terms of path expressions in the absence

of schema [BDF$^+$01].

In contrast to traditional keys, keys for graphs (Chapter 3) (a) are defined in terms of *graph patterns*, specifying constraints on both topological structure and value bindings, in the absence of schema; (b) they are interpreted based on *graph pattern matching*, with both value equality and node identity; and (c) they can be *recursively defined*. These keys are useful in emerging applications besides their traditional use.

To the best of our knowledge, the only prior work on keys for graphs is [PSS13], which specifies keys for RDF data in terms of a combination of object properties and data properties defined over OWL ontology. Such keys differ from keys of this work in that they (a) cannot be recursively defined, (b) do not enforce topological constraints imposed by graph patterns, and (c) adopt the unique name assumption via URIs, which is often too strong for entity matching.

Dependencies for graphs. Dependencies have been studied for RDF [LMS08, ACP10, CP12, YH11, HZZ14, FFTD15], and for generic graphs [FWX16, FL17]. They are used to (1) map relations to RDF [CFP$^+$14, LMS08], (2) detect erroneous triples in RDF graphs [YH11, HZZ14] and inconsistencies in property graphs [FWX16], (3) identify objects [FFTD15], (4) repair vertex labels [SCYC14], and (5) enrich knowledge bases with association rules [GTHS13]. This line of work started from [LMS08]. It extends RDF vocabulary to define keys, foreign keys and functional dependencies (FDs). Using triple patterns with variables, [ACP10, CP12] interpret FDs with triple embedding and homomorphism. Based on value-clustered property, a class of FDs was also formulated in [YH11] with path patterns; these FDs were extended in [HZZ14] to support CFDs. [GTHS13, CGWJ16] study a class of first-order Horn clause on binary predicates as soft constraints to facilitate knowledge base reasoning, expansion and cleaning.

Closer to this work are graph functional dependencies (GFDs) studied for general property graphs [FWX16, FL17]. GFDs are formulated in [FWX16] in terms of (a) a graph pattern Q that is interpreted via subgraph isomorphism, and (b) an extension of an FD that is imposed on the entities identified by Q, carrying constant and variable literals. GFDs subsume relational CFDs [FGJK08] when tuples are represented as nodes in a graph. GFDs are extended to graph entity dependencies (GEDs) in [FL17], by supporting literals with node identities to express keys of [FFTD15]. GEDs subsume GFDs, and interpret graph pattern matching in terms of graph homomorphism.

This work defines NGDs (Chapter 4) by extending GFDs, and interprets pattern

matching by graph homomorphism following [FL17]. It differs from [FWX16, FL17] in the following. (a) Unlike GFDs and GEDs, NGDs support both arithmetic operations and comparison predicates. (b) We settle fundamental problems (satisfiability, implication and validation) for NGDs. We show that their satisfiability and implication problems are Σ_2^p-complete and Π_2^p-complete, respectively, while these problems are NP-complete and coNP-complete for GFDs and GEDs. (c) We develop (parallel) incremental error detection algorithms with performance guarantees. As far as we know, no prior work has studied dependencies for capturing numeric errors in graphs, established their complexity, or developed incremental error detection algorithms for graphs, sequential or parallel.

In addition, we adopt the GEDs of [FL17] in Chapter 5 because (a) GEDs can express GFDs of [FWX16] and (recursively defined) keys of Chapter 3, and allow us to support data repairing and object identification; and (b) GEDs are defined for general property graphs, not limited to RDF. We extend the GEDs of [FL17] by supporting negative rules with inequality. As observed in [FLM$^+$11, ARS09], negative rules improve the accuracy of object identification by reducing false positives.

Dependencies on numeric data. The need for detecting numeric errors has long been recognized, and several dependency classes have been studied for relational data for this purpose [BBFL08, FFP10, FPL$^+$01, GKK$^+$09, KSSV09, RSSS98, FFG14, SC11]. Metric functional dependencies [KSSV09] and sequential dependencies [GKK$^+$09] extend FDs by supporting (numeric) metrics and intervals on ordered data, respectively. Differential dependencies [SC11] constrain distances of numeric attribute values among different tuples. However, none of these dependency classes supports arithmetic operations. Beyond these, aggregation constraints are defined in terms of aggregate functions (*e.g.,* max, min, sum, avg, count) [RSSS98, BBFL08]. However, it is undecidable to decide whether a set of aggregation constraints is satisfiable. There has also been work on repairing numeric data using constraints defined in terms of aggregate functions [FFP10] and disjunctive logic programming [FPL$^+$01]. Their satisfiability and implication problems are open, and the complexity is suspected high. An extension of CFDs was studied for relations in [FFG14], which supports linear arithmetic expressions and built-in predicates like we do in NGDs. Unfortunately, none of these numeric constraints is applicable to graph-structured data.

To the best of our knowledge, NGDs (Chapter 4) are among the first effort to incorporate arithmetic expressions into graph dependencies. In contrast to the prior numeric

constraints studied for relations, NGDs are a combination of a topological constraint defined in terms of pattern matching, and an attribute dependency defined with linear arithmetic expressions and built-in comparison predicates.

1.3.3   Data Cleaning

Entity resolution.   Entity resolution (*a.k.a.* entity matching, record linkage, etc.) is to identify records that refer to the same real-world entity. There has been a host of work on the topic, following iterative clustering [BGMG$^+$07, MAS14], learning-based [RDG11, KTR12a], rule-based methods [FGJ$^+$11, ARS09] (see [Chr12, GM12] for surveys).

Keys for graphs (Chapter 3) yield a *declarative and deterministic* method to provide an invariant connection between vertices and the real-world entities they represent, and fall in the rule-based approach. Prior rule-based methods mostly focus on *relational data*; this work is to define a primary form of constraints for *graphs*, namely, keys. The quality of matches identified by keys highly depends on keys discovered and used, although keys help us reduce false positives. We defer the topic of key discovery to future work, and focus primarily on the efficiency of applying such constraints.

One branch of entity resolution, called collective entity resolution [BG07, DHM05, RDG11], is to jointly determine entities for co-occurring references and propagate similarities of entities. Analogous to datalog rules [ARS09], keys for graphs extend this approach by providing recursively defined rules, based on *graph pattern matching*. This work addresses some of the emerging challenges highlighted in [GM12], by targeting graphs when data is "more linked", and by providing parallel scalable algorithms for "larger datasets". A topic for future work is to combine our approach with learning-based and similarity propagation methods [GM12], and thus to find high-quality keys and improve the quality of linkage results.

Finally, we remark that entity resolution is just *one of* the applications for keys for graphs, besides, *e.g.,* digital citations [BS10] and knowledge base expansion [DGH$^+$14].

Data cleaning. Error detection and cleaning algorithms have been developed for relations [FLTY12, VCSM14, PSC$^+$15] (see [FG12] for a survey), RDF [SSW09, WP14, KWA$^+$14] and general property graphs [FWX16, SCYC14]. There have been heuristic methods to find repairs with minimum costs to fix violations of FDs [BFFR05] and CFDs [CFG$^+$07]. [FLTY12] studies (incremental) detection of violations of

CFDs [FGJK08] in horizontally or vertically partitioned relations. GDR [YEN$^+$11] incorporates user feedback into automated repairing. QFix [WMW17] repairs predicates in restricted relational SPC queries, instead of data, in response to anomalous updates in OLTP. Rule-based methods and machine learning are combined in [PSC$^+$15] to ensure that the cleaned data differs minimally, in terms of both the distance from the original data, and the statistical distortion from an ideal relation. Based on CFDs [FGJK08] and matching dependencies [FGJ$^+$11], [FLM$^+$11] unifies entity resolution and data repairing to clean relations. Repairing relations with certain fixes is studied in [FLM$^+$12] with master data. Based on relational FDs, a continuous framework is developed in [VCSM14] to clean data that may change, using FDs that may also evolve. Consistency checking in RDF data is conducted by logical reasoning on existing and newly extracted knowledge [SSW09], or by unsupervised detection of numerical outliers [WP14]. [KWA$^+$14] detects errors in RDF data through test cases that are represented as SPARQL queries.

On graphs, [SCYC14] fixes vertex labels to make graphs satisfy neighborhood constraints, which allow only certain label pairs to appear on adjacent nodes. Batch algorithms are proposed in [FWX16] for catching violations of GFDs in graphs that are either replicated or fragmented across multiple processors. Collective graph identification [JLG16] infers "true" hidden graphs from the given observed networks (treated as "ground truth"), formulated as a probabilistic inference problem. It involves identifying observations that correspond to the same entity (entity resolution), inferring the existence of edges (link prediction) and inferring hidden labels of nodes (node labeling). Unsupervised clustering is studied for entity idnetification in bibliographic datasets [BG06].

Different from the prior work, in Chapter 4, (1) we provide incremental error detection algorithms that are localizable [FHT17] and relatively parallel scalable; as far as we know, none of the previous error detection algorithms is parallel scalable except the batch ones of [FWX16, FFTD15]; and (2) we propose a hybrid dynamic strategy to achieve relative parallel scalability. The strategy balances the workload at run time, at two levels: (a) it makes use of cost estimation to split and distribute stragglers, *i.e.,* work units that take much longer than the others, and (b) it monitors the status of processors and reassigns work units from a busy processor to those with a light load. While (b) is along the same lines as work stealing and shedding [HCD$^+$16, BL99], we find that it does not work very well alone unless in combination with (a).

Incremental detection of NGD violations is more intriguing than conventional

graph pattern matching: we have to compute violations that are newly introduced or removed by updates only. As a consequence, previous algorithms for parallel pattern matching, *e.g.,* [HAR11, LQLC15], cannot be applied directly in this context.

Moreover, to the best of our knowledge, this work is also the first effort to clean graphs with certain fixes (Chapter 5). (a) As opposed to [FFTD15, BG06], we support both data repairing and object identification. We aim to fix inconsistencies among attribute values of different entities, beyond node labeling [SCYC14] that is regarded as a task of collective classification of entities in [JLG16]. The method of [JLG16] targets hidden graph inference rather than graph cleaning, and is complementary to this work. While [FL17] studies chase of graphs, it does not consider graph cleaning. (b) In contrast to [FLM$^+$12], we define certain fixes as the result of chase and logical consequences of GEDs and ground truth; moreover, we support object identification in addition to repairing. As suggested by the experience of relational data cleaning [FLM$^+$11], we interleave object identification and data repairing to improve the accuracy of both. However, the methods of [FLM$^+$11, FLM$^+$12] are not able to fix inconsistencies that involve correlated or co-occurred entities. In fact, the prior methods for cleaning relations are not directly applicable to graphs. For instance, the repairing methods based on FDs and CFDs are not able to repair inconsistencies that involve (unboundedly many) interconnected entities. (c) We make the first efforts to settle fundamental problems underlying graph cleaning, such as the complexity of certain fix and coverage.

Parallel algorithms.   Parallel algorithms have been developed for entity resolution [KTR12b, AM17, FFTD15, CIK16], in which the technique of distributed blocking is widely used to reduce the search space. BigDansing [KIJ$^+$15] repairs relations on top of MapReduce-like frameworks. There has also been work on error detection in distributed relations [FGMM10] with CFDs, and in fragmented graphs [FWX16] using GFDs.

Our algorithms in Chapter 3 differ from previous ones in the following. (a) Entity matching is far more intriguing than conventional subgraph isomorphism, and the prior algorithms [KLCL13, GG14, SWW$^+$12, RvRH$^+$14] cannot be applied to entity matching. (b) For the same reasons, entity matching is more involved than record matching of [BGMG$^+$07, KTR12a, MAS14, RDG11] to identify tuples in relations, and than the task of [HNST12] that does not enforce topological constraints in the matching process. (c) We propose optimization strategies that have not been studied before.

Related to this work are also parallel algorithms for evaluating datalog [ABC$^+$11, SPSL13], which deal with recursive computation. However, entity matching with keys (Chapter 3) requires to identify *bijective functions* for subgraph isomorphism, which are more challenging to compute than relations in datalog. Worse still, we show that entity linking does not have PFP [ABC$^+$11], and is harder to be parallelized than, *e.g.,* transitive closures.

Moreover, none of existing approaches targets fixing inconsistencies in graphs. In Chapter 5, we present a simple partition strategy to balance the workload in graph cleaning, and guarantee relative parallel scalability that has not been accomplished by previous data cleaning methods.

Chapter 2

Incremental Graph Computations: Impossibility and Possibility

This chapter studies both the possibilities and impossibilities for the effectiveness of incremental graph computations.

As remarked in Chapter 1, incremental graph computations is desirable for querying real-life graphs. However, when ΔG is small and G is big, can we guarantee that it is more efficient to compute ΔO with incremental algorithm $\mathcal{T}_\Delta$ than to recompute $Q(G \oplus \Delta G)$ with batch algorithm $\mathcal{T}$? A traditional characterization is by means of a notion of *boundedness* proposed in [TR81] and extended to graphs in [RR96a, FWW13]. It measures the cost of $\mathcal{T}_\Delta$ in $|\mathsf{CHANGED}| = |\Delta G| + |\Delta O|$, the size of the changes in the input and output. We say that $\mathcal{T}_\Delta$ is *bounded* if its cost can be expressed as a polynomial function of $|\mathsf{CHANGED}|$ and $|Q|$. The incremental problem for Q is *bounded* if there exists a bounded $\mathcal{T}_\Delta$ for Q, and is *unbounded* otherwise.

Bounded $\mathcal{T}_\Delta$ allows us to reduce the incremental computations on big graphs to small graphs. Its cost is determined by $|\mathsf{CHANGED}|$ and query size $|Q|$, rather than by the size $|G|$ of the entire G. In the real world, $|Q|$ is typically small; moreover, $|\mathsf{CHANGED}|$ represents the updating cost that is *inherent to* the incremental problem itself, and is often much smaller than $|G|$. Hence bounded $\mathcal{T}_\Delta$ warrants efficient incremental computation no matter how big G is.

Undoable. No matter how desirable, we show that the incremental problem for Q is *unbounded* when Q ranges over graph traversal (RPQ, regular path queries), strongly connected components (SCC) and keyword search (KWS). The negative results hold when ΔG consists of a single edge deletion or insertion. Add to it the unboundedness of

graph pattern matching via subgraph isomorphism (ISO) [FWW13]. For these common queries, a bounded incremental algorithm is beyond reach. That is, by the standard of boundedness, incremental graph algorithms seem not very helpful.

Doable.   The situation is not so hopeless.   The boundedness of [TR81, RR96a, FWW13] is often too strong to evaluate incremental algorithms. To characterize the effectiveness of real-life incremental algorithms, we propose two alternative measures.

(1) Localizable computations. We say that the incremental problem for Q is *localizable* if there exists an incremental algorithm $\mathcal{T}_\Delta$ such that for $Q \in \mathcal{Q}$, G and ΔG, its cost is determined by $|Q|$ and the d_Q-neighbors of nodes in ΔG, where d_Q is decided by $|Q|$ only. In practice, Q is typically small, and so is d_Q. Hence it allows us to reduce the computations on (big) G to small d_Q-neighbors of ΔG.

We show that the incremental problems for KWS and ISO are localizable, although they are unbounded.

(2) Relative boundedness. We often want to incrementalize a batch algorithm $\mathcal{T}$ for Q. For a query $Q \in \mathcal{Q}$ and a graph G, we denote by $G_{(\mathcal{T},Q)}$ the part of data in G inspected by $\mathcal{T}$ when computing $Q(G)$. Given updates ΔG to G, denote by AFF the difference between $(G \oplus \Delta G)_{(\mathcal{T},Q)}$ and $G_{(\mathcal{T},Q)}$.

An incremental algorithm $\mathcal{T}_\Delta$ for Q is *bounded relative to* $\mathcal{T}$ if its cost is a polynomial in $|\Delta G|$, $|Q|$ and $|\text{AFF}|$. Intuitively, AFF indicates the necessary cost for incrementalizing $\mathcal{T}$, and $\mathcal{T}_\Delta$ incurs this minimum cost, not measured in $|G|$.

We show that RPQ and SCC are relatively bounded, *i.e.,* it is possible to incrementalize their popular batch algorithms $\mathcal{T}$ and minimize unnecessary recomputation of $\mathcal{T}$.

2.1   Incremental Graph Computations

We first present graph queries studied in this chapter, and then formulate their incremental problems.

We start with basic notations.

We consider directed *graphs G*, represented as (V, E, l), where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges in which (v, v') denotes an edge from v to v', and (3) each node v in V carries $l(v)$, indicating its label and content, as found in social networks and property graphs.

If (v,w) is edge in E, we refer to node w as a *successor* of v, and to node v as a *predecessor* of w.

Graph $G_s = (V_s, E_s, l_s)$ is a *subgraph of* G if $V_s \subseteq V$, $E_s \subseteq E$, and for each node $v \in V_s$, $l_s(v) = l(v)$.

Subgraph G_s is *induced by* V_s if E_s consists of all the edges in G such that their endpoints are both in V_s.

2.1.1 Graph Queries

We study the following four classes of graph queries.

RPQ. Consider directed graphs $G = (V, E, l)$ over a finite alphabet Σ of labels defined on the nodes in V. A *path* ρ from v_0 to v_n in G is a list $(v_0, \ldots, v_n)$, where for $i \in [0, n-1]$, (v_i, v_{i+1}) is an edge in G. The *length* of path ρ is n.

A *regular path query* Q is a regular expression as follows:

$$Q ::= \varepsilon \mid \alpha \mid Q \cdot Q \mid Q + Q \mid Q^*.$$

Here (a) ε denotes an empty path; (b) α is a label from Σ; (c) $\cdot$ and $+$ are concatenation and union operators, respectively; and (d) Q^* indicates zero or more occurrences of Q.

We use $L(Q)$ to denote the regular language defined by Q, *i.e.,* the set of all strings that can be parsed by Q. For a path $\rho = (v_0, \ldots, v_n)$ in G, we use $l(\rho)$ to denote the labels $l(v_0) \cdots l(v_n)$ of the nodes on ρ. A *match* of Q in G is a pair (v, w) of nodes such that there exists a path ρ from v to w having $l(\rho) \in L(Q)$. RPQ is stated as follows.

- Input: A directed graph G and a regular path query Q.
- Output: The set $Q(G)$ of all matches of Q in G.

It takes $O(|V||E||Q|^2 \log^2 |Q|)$ time to compute $Q(G)$ by using NFA (nondeterministic finite automaton) [MW95, HSW01], where $|Q|$ is the number of occurrences of labels from Σ in Q [HSW01].

SCC. A subgraph G_s of a directed graph G is a *strongly connected component* of G if it is (a) strongly connected, *i.e.,* for any pair (v, v') of nodes in G_s, there is a path from v to v' and vice versa, and (b) maximum, *i.e.,* adding any node or edge to G_s makes it no longer strongly connected.

We use $\mathsf{SCC}(G)$ to denote the set of all strongly connected components of G. The SCC problem is stated as follows.

- Input: A directed graph G.
- Output: $\mathsf{SCC}(G)$.

It is known that SCC is in $O(|V|+|E|)$ time [Tar72].

KWS. We consider keyword search with distinct roots in the same setting of [QYC$^+$14]. A keyword query Q is of the form $(k_1,\ldots,k_m)$, where each k_i is a keyword. Given a directed graph G and a bound b, *a match to Q in G at node r* is a tree $T(r,p_1,\ldots,p_m)$ such that (a) T is a subgraph of G, and r is the root of T, (b) for each $i \in [1,m]$, p_i is a node in T such that $l(p_i) = k_i$, *i.e.*, it matches keyword k_i, (c) $\mathrm{dist}(r,p_i) \leq b$, and (d) the sum $\Sigma_{i\in[1,m]}\mathrm{dist}(r,p_i)$ is the smallest among all such trees. Here for a pair (r,s) of nodes, $\mathrm{dist}(r,s)$ denotes the *shortest distance* from r to s, *i.e.*, the length of a shortest path from r to s. KWS is as follows.

- Input: A directed graph G, a keyword query $Q = (k_1,\ldots,k_m)$, and a positive integer b.
- Output: The set $Q(G)$ of all matches to Q at node r in G within b hops, for r ranging over all nodes in G.

It can be computed in $O(m(|V|\log|V|+|E|))$ time (cf. [YQC10]).

ISO. A *pattern query Q* is a connected graph (V_Q,E_Q,l_Q), in which V_Q and E_Q are the set of pattern nodes and directed edges, respectively, and each node u in V_Q has a label $l_Q(u)$.

A *match* of Q in G is a subgraph G_s of G that is isomorphic to Q, *i.e.*, there exists a *bijective function h* from V_Q to the set of nodes of G_s such that (a) for each node $u \in V_Q$, $l_Q(u) = l(h(u))$, and (b) (u,u') is an edge in Q iff $(h(u),h(u'))$ is an edge in G_s. The answer $Q(G)$ to Q in G is the set of all matches of Q in G. ISO is stated as follows.

- Input: A directed graph G and a pattern Q.
- Output: The set $Q(G)$ of all matches of Q in G.

It is NP-complete to decide whether $Q(G)$ is empty (cf. [Pap94]).

2.1.2   Incremental Query Answering

We next formalize incremental computation problems.

Updates. We consider *w.l.o.g.* the following *unit updates*:
- edge insertion: (insert e), possibly with new nodes, and
- edge deletion: (delete e).

A *batch update ΔG* to graph G is a sequence of unit updates.

Incremental problem. For a class Q of graph queries, the incremental problem is

symbols	notations
G	a (directed) graph (V, E, l)
Q	a query in a query class $\mathcal{Q}$
$Q(G)$	the answers to query Q in graph G
ΔG	updates to graph G (edge insertions, deletions)
$G \oplus \Delta G$	the graph obtained by updating G with ΔG
ΔO	updates to old output $Q(G)$ in response to ΔG
$\mathcal{T}$	a batch algorithm for a query class $\mathcal{Q}$
$\mathcal{T}_\Delta$	an incremental algorithm for $\mathcal{Q}$
AFF	changes to the area inspected by a batch algorithm $\mathcal{T}$
$|G|$	the size of G; similarly for $|\Delta G|$, $|Q|$
$\text{dist}(s, t)$	the shortest distance from node s to t
$G_d(v)$	the d-neighbor of node v in G

Table 2.1: Notations in Chapter 2: graphs and queries

stated as follows.

- ○ Input: Graph G, query $Q \in \mathcal{Q}$, old output $Q(G)$, and updates ΔG to the input graph G.
- ○ Output: Updates ΔO to the output such that

$$Q(G \oplus \Delta G) = Q(G) \oplus \Delta O.$$

We study the problem for RPQ, SCC, KWS and ISO.

The notations used in this chapter are summarized in Table 2.1.

2.2 Bounded Incremental Problems: Undoable

This section shows the following impossibility results.

Theorem 2.1: *The incremental problem is unbounded for*

- ○ *regular path queries (*RPQ*),*
- ○ *strongly connected components (*SCC*), and*
- ○ *keyword search (*KWS*),*

even under a unit edge deletion and a unit edge deletion. □

Together with the unboundedness of ISO [FWW13], Theorem 2.1 tells us that it is

impossible to find bounded incremental algorithms for all the graph query classes presented in Section 2.1. The negative results are rather robust: the incremental problems are already unbounded under unit updates.

Before we give a proof, we first review the notion of boundedness of [RR96a, FWW13], and introduce a form of Δ-reductions.

Boundedness. An incremental algorithm $\mathcal{T}_\Delta$ for a graph query class Q is *bounded* if its cost can be expressed as a polynomial of $|\mathsf{CHANGED}|$ and $|Q|$, where $|\mathsf{CHANGED}|$ $= |\Delta G| + |\Delta O|$. Following [RR96a, FWW13], we require $\mathcal{T}_\Delta$ to be *locally persistent*. Such $\mathcal{T}_\Delta$ may use (a) auxiliary structures associated with each node v of G, to keep track of intermediate results at v, and (b) pointers to its successors and predecessors. However, no global auxiliary information is allowed, such as pointers to nodes other than its neighbors; similarly for edges. The algorithm starts an update from the nodes or edges involved in ΔG, and traverses G following the edges of G. The choice of which edge to follow depends only on the information accumulated in the current processing of G since global information from prior passes is not maintained.

Reductions. We now introduce Δ-*reduction*. Consider two classes of graph queries Q_1 and Q_2. For $i \in [1,2]$, we represent an instance of (the computational problem for) Q_i as $I_i = (Q_i, G_i)$, where $Q_i \in Q_i$ and G_i is a graph.

A Δ-*reduction* from Q_1 to Q_2 is a triple (f, f_i, f_o) of functions such that for each instance $I_1 = (Q_1, G_1)$ of Q_1,

  (1) $f(I_1)$ is an instance $I_2 = (Q_2, G_2)$ of Q_2; and

  (2) for all updates ΔG_1 to G_1,

      (a) $f_i(\Delta G_1)$ computes updates ΔG_2 to G_2; and

      (b) $f_o(\Delta O_2)$ computes ΔO_1, where ΔO_i denotes updates to $Q_i(G_i)$ in response to ΔG_i for $i \in [1,2]$,

in polynomial-time (PTIME) in $|\Delta G_1| + |\Delta O_1|$ and $|Q_1|$.

Intuitively, f maps the instances of Q_1 to Q_2; f_i maps input updates ΔG_1 to ΔG_2, and f_o maps output updates O_2 back to O_1, both in PTIME in the size of Q_1 and changes in the input and output of instance (Q_1, G_1), where (Q_2, G_2) corresponds to (Q_1, G_1) via function f. Hence if Q_2 has a bounded incremental algorithm, then so does Q_1. Equivalently, if Q_1 is unbounded, neither is Q_2. That is, Δ-reduction preserves boundedness.

Lemma 2.2: *If there exists a Δ-reduction from Q_1 to Q_2 and if the incremental problem*

for Q_2 is bounded, then the incremental problem for Q_1 is also bounded. □

Proof of Lemma 2.2. Assume that there exists a bounded incremental algorithm $\mathcal{T}_\Delta$ for Q_2. We show that a bounded incremental algorithm $\mathcal{T}'_\Delta$ for Q_1 can be built from $\mathcal{T}_\Delta$ and the Δ-reduction (f, f_i, f_o) from Q_1 to Q_2. Given an instance $I_1 = (Q_1, G_1)$ of Q_1, we first compute a corresponding instance $f(I_1) = (Q_2, G_2)$ of Q_2. Then for each update ΔG_1 to G_1, $\mathcal{T}'_\Delta$ transforms it to $f_i(\Delta G_1)$ and invokes the bounded incremental algorithm $\mathcal{T}_\Delta$ on G_2, Q_2, $Q_2(G_2)$ and $f_i(\Delta G_1)$ to obtain ΔO_2, *i.e.,* the corresponding changes to $Q_2(G_2)$. Thereafter, it transforms the updates ΔO_2 back to ΔO_1 leveraging function f_o. As (f, f_i, f_o) is a Δ-reduction, it concludes that $f_o(\Delta O_2) = \Delta O_1$, where ΔO_1 denotes the updates to $Q_1(G_1)$ in response to ΔG_1, and $\mathcal{T}'_\Delta$ takes PTIME in $|\Delta G_1| + |\Delta O_1|$ and $|Q_1|$ to compute ΔO_1, *i.e.,* $\mathcal{T}'_\Delta$ is a bounded incremental algorithm for Q_1. From this Lemma 2.2 follows. □

Proof of Theorem 2.1. Based on Δ-reduction, we give nontrivial proofs for RPQ, SCC, and KWS one by one, which reveal the challenges to the development of incremental algorithms. For each query class, we need to give two proofs: one under a unit edge deletion, and the other under a unit insertion. Indeed, a problem may be unbounded under deletions (resp. insertions) but be bounded under insertions (resp. deletions). An example is SSRP, the single-source reachability problem to all vertices. It is to decide, given a graph G and a node v_s in G, whether there exists a path from v_s to v_t for all nodes v_t in G. It is known that SSRP is unbounded under unit edge deletions but bounded under unit edge insertions [RR96a].

RPQ. We consider first edge deletions, and then insertions.

(1) Deletions. We prove the unboundedness of the incremental problem for RPQ under a unit edge deletion by Δ-reduction from the single source reachability problem to all vertices (SSRP). Given a graph $G = (V, E, l)$ and a node $v_s \in V$, SSRP is to decide whether node v_i is reachable from v_s for all $v_i \in V$. The answer is expressed as Boolean value $r(v_i)$ associated with v_i. The incremental problem for SSRP is unbounded under unit edge deletions [RR96a].

Given an instance I_1 of SSRP, *i.e.,* a graph $G_1 = (V_1, E_1, l_1)$ and a distinguished node v_s in G_1, we construct an instance I_2 of RPQ, *i.e.,* a graph $G_2 = (V_2, E_2, l_2)$ and a regular path query Q_2, by using function f such that the reachability $r(v_i)$ from v_s to v_i in G_1 changes in response to ΔG_1 iff (if and only if) there exists a corresponding change in the output of Q_2 on G_2 in response to ΔG_2, where input and output updates
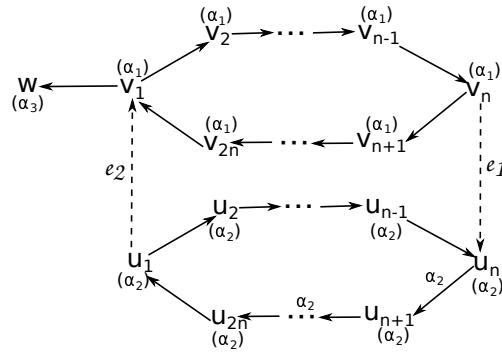
Figure 2.1: Unboundedness for RPQ and SCC

of the two instances are mapped by functions f_i and f_o, respectively (see Section 2.2).

More specifically, G_2 is constructed from G_1 with each node v_i replaced by v'_i. All the edges in G_1 remain unchanged, *i.e.*, $(v'_i, v'_j) \in E_2$ iff $(v_i, v_j) \in E_1$. Furthermore, $l_2(v'_i) = \alpha_1$ when $v'_i = v'_s$, and $l_2(v'_i) = \alpha_2$ otherwise, where v'_s corresponds to source node v_s in G_1. Query Q_2 is defined as $\alpha_1 \cdot (\alpha_2)^*$. Then one can verify that v_i is reachable from v_s in G_1 iff the node pair (v'_s, v'_i) is a match of Q_2 in G_2. Indeed, the source node of each match in $Q_2(G_2)$ must be v'_s since all paths having label α_1 originate from v'_s.

Given delete(v_i, v_j) in ΔG_1, function f_i returns corresponding (v'_i, v'_j) to be deleted from G_2, *i.e.*, $\Delta G_2 = f_i(\Delta G_1)$. Then the changes ΔO_2 to $Q_2(G_2)$ consist of node pairs (v'_s, v'_i) removed. Clearly, v'_i is no longer reachable from v'_s in G_2 and v_i is not reachable from v_s in G_1; hence ΔO_1 is the set of such $r(v_i)$ changed from true to false, which can be computed by $f_o(\Delta O_2)$ directly. Thus, a one-to-one mapping between the changes of I_1 and I_2 is obtained via linear-time functions f_i and f_o.

Putting these together, (f, f_i, f_o) is a Δ-reduction and RPQ is unbounded under a unit edge deletion by Lemma 2.2.

(2) Insertions. We next show that RPQ is unbounded under a unit edge insertion by contradiction. Consider graph G shown in Fig. 2.1 (excluding dotted edges), which consists of two cycles $(v_1, v_2), \ldots, (v_{2n-1}, v_{2n}), (v_{2n}, v_1)$ and $(u_1, u_2), \ldots, (u_{2n-1}, u_{2n})$, (u_{2n}, u_1), and an edge (v_1, w). Each node v_i in G has label α_1 for $i \in [1, 2n]$, while u_i is labeled α_2. Node w is labeled α_3 that is distinct from α_1 and α_2. Query Q is defined as $\alpha_1 \cdot (\alpha_1)^* \cdot \alpha_2 \cdot (\alpha_2)^* \cdot \alpha_3$. Denote by Δ_1 the insertion of $e_1 = (v_n, u_n)$, and by Δ_2 the insertion of $e_2 = (u_1, v_1)$. Let graph $G_1 = G \oplus \Delta_1$, $G_2 = G \oplus \Delta_2$ and $G_3 = G_1 \oplus \Delta_2$. One can verify that $Q(G) = Q(G_1) = Q(G_2) = \emptyset$, while $Q(G_3) = \{(v_i, w) \mid i \in [1, 2n]\}$.

Assume by contradiction that there exists a bounded incremental algorithm $\mathcal{T}_\Delta$ for RPQ under a unit edge insertion. Then $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_1)$ and $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_2)$ are

both in $O(1)$ time since only a unit update is applied to G and none of the outputs is affected for the fixed query Q. We next show that this leads to contradiction.

Let $T_s(G, \Delta G)$ denote the sequence of nodes visited in executing $\mathcal{T}_\Delta(G, Q, Q(G), \Delta G)$, referred to as its *trace*. Observe that $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_2)$ and $\mathcal{T}_\Delta(G_1, Q, Q(G_1), \Delta_2)$ must behave differently as the outputs of these two are different, in which $\mathcal{T}_\Delta(G_1, Q, Q(G_1), \Delta_2)$ computes $Q(G_3)$ exactly. This can happen only if $T_s(G, \Delta_2)$ and $T_s(G_1, \Delta_2)$ contain some node associated with different information in G and G_1 as $\mathcal{T}_\Delta$ traverses the graph from the nodes involved in Δ_2, *i.e.*, u_1 or v_1. Since G_1 is obtained by applying Δ_1 to G, these nodes must be included in $T_s(G, \Delta_1)$ with information updated. Observe that if a node v in G is visited during the execution of a locally persistent algorithm $\mathcal{T}_\Delta$ to process ΔG, then each node on some undirected path from the position of ΔG to v is also inspected by $\mathcal{T}_\Delta$. Denote by v_d the the first node having different information in $T_s(G, \Delta_2)$ and $T_s(G_1, \Delta_2)$. Then $T_s(G, \Delta_1)$ and $T_s(G, \Delta_2)$ include all the nodes on an undirected path from the position of Δ_1 to that of Δ_2 through v_d. However, the length of this path is $O(n)$, which contradicts the assumption that $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_1)$ and $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_2)$ both take constant time.

SCC. To comply with locally persistent algorithm, we define the output of SCC by using $\mathsf{SCC}(v)$ for each node v in G, where $\mathsf{SCC}(v)$ is the minimum id of the nodes that are in the same strongly connected component as v.

(1) Deletions. We show that the incremental problem for SCC is unbounded under a unit edge deletion also by Δ-reduction from SSRP. Given an instance I_1 of SSRP consisting of a graph $G_1 = (V_1, E_1, l_1)$ and a source node $v_s \in V_1$, function f is to construct an SCC instance I_2 of a graph $G_2 = (V_2, E_2, l_2)$ such that the reachability from v_s to v_i changes in G_1 iff $\mathsf{SCC}(v_i')$ in G_2 changes, in response to any edge deletion ΔG_1 to G_1 and its corresponding update ΔG_2 to G_2. Here v_i' denotes the corresponding node of v_i in G_2, which is determined by function f. More specifically, function f constructs graph G_2 as follows.

(a) Function f generates G_2 such that each node $v_i \in V_1$ is mapped to v_i' in V_2, and each edge (v_i, v_j) in E_1 is mapped to (v_i', v_j') in E_2. It maps the distinguished v_s in G_1 to v_s' with the minimum node id 1 in G_2.

(b) If G_{s_i} is a strongly connected component of G_1 *excluding* node v_s, then function f adds a node u_i together with the edges (u_i, v_j') and (v_j', u_i) to G_2 for each node v_j in G_{s_i}.

(c) All the nodes v_i' added to G_2 in (a) are directly connected to v_s', *i.e.*, (v_i', v_s') is

included in E_2 by f for each such v_i'.

Function f_i encodes unit delete(v_i, v_j) in ΔG_1 as the deletion ΔG_2 of (v_i', v_j') in G_2 if $v_j \neq v_s$, otherwise ΔG_2 is empty. Then ΔO_2 of SCC on $G_2 \oplus \Delta G_2$ only contains some node v_i' or u_i having SCC(v_i') (resp. SCC(u_i)) changed to a larger integer from 1. To see this, observe the following.

(a) A node v_i is reachable from v_s in $G_1 \oplus \Delta G_1$ if and only if v_i' and v_s' are in the same strongly connected component in $G_2 \oplus \Delta G_2$, i.e., SCC$(v_i') =$ SCC(v_s'). This is because there exists an edge (v_i', v_s') in G_2 that will never be deleted.

(b) For each set of nodes within a strongly connected component G_{s_i} in G_1 excluding v_s, their corresponding nodes are still in the same strongly connected component of $G_2 \oplus \Delta G_2$ together with another node u_i. This is because edges adjacent to u_i will not be affected by ΔG_1 via function f_i.

Therefore, changes ΔO_2 only contain some node v_i' or u_i that are separated from the strongly connected component containing v_s', which has id 1 since v_s' has the minimum id in G_2. Conversely, the corresponding v_i of such v_i' in G_1 is no longer reachable from v_s, which represents ΔO_1 and can be obtained by a liner-time function f_o. Obviously f_i and f_o are both one-to-one. Thus f, f_i and f_o make a Δ-reduction from SSRP, and SCC is unbounded under unit edge deletions.

(2) Insertions. We next show that the incremental problem for SCC is also unbounded under unit edge insertions. We use the same G, Δ_1, Δ_2, G_1, G_2 and G_3 as shown in Fig. 2.1, defined in the proof for RPQ above. It can be verified that SCC$(G) =$ SCC$(G_1) =$ SCC(G_2), where $\{v_i \mid i \in [1, 2n]\}$, $\{u_i \mid i \in [1, 2n]\}$ and $\{w\}$ constitute the node set of three strongly connected components. However, the first two are combined in G_3, i.e., $O(n)$ amount of SCC(v) values in G_3 are different from their counterparts in G.

Assume by contradiction that there exists a bounded incremental algorithm $\mathcal{T}_\Delta$. Then $\mathcal{T}_\Delta(G, \mathsf{SCC}, \mathsf{SCC}(G), \Delta_1)$ and $\mathcal{T}_\Delta(G, \mathsf{SCC}, \mathsf{SCC}(G), \Delta_2)$ both take constant time as $|\Delta_1| = |\Delta_2| = 1$, and compared with SCC$(G)$, SCC$(G_1)$ and SCC(G_2) remain unchanged.   Consider the executions of $\mathcal{T}_\Delta(G, \mathsf{SCC}, \mathsf{SCC}(G), \Delta_2)$ and $\mathcal{T}_\Delta(G_1, \mathsf{SCC}, \mathsf{SCC}(G_1), \Delta_2)$ that behave differently. Since G_1 is the same as G except the insertion of edge e_1, the different behaviors of $\mathcal{T}_\Delta$ on G and G_1 when processing Δ_2 can only be triggered if the traces $T_s(G, \Delta_2)$ and $T_s(G_1, \Delta_2)$ contain some node associated with different information in G and G_1, respectively. This could happen only

if these nodes are contained in the trace $T_s(G, \Delta_1)$ with information updated during the processing of Δ_1 on G. As $\mathcal{T}_\Delta(G, \mathsf{SCC}, \mathsf{SCC}(G), \Delta_1)$ is in $O(1)$ time, $T_s(G, \Delta_1)$ includes a constant number of nodes.

Now we construct a graph G_1' from G_1 by skipping the nodes in $T_s(G, \Delta_1)$. More specifically, for each node v_i in $T_s(G, \Delta_1)$, we add an edge (v_{i-1}, v_{i+1}) and leave out v_i along with the edges adjacent to it. Note that (v_{2n}, v_2) and (v_{2n-1}, v_1) are used for removing v_1 and v_{2n}, respectively, edges e_1 and (v_1, w) are also adjusted accordingly when removing v_n or v_1. The removal of u_i is processed similarly. The information residing at each node remains unchanged. Note that G_1' is not empty when n is sufficiently large. Denote by G_3' the graph $G_1' \oplus \Delta_2$. It follows that $\mathcal{T}_\Delta(G, \mathsf{SCC}, \mathsf{SCC}(G), \Delta_2)$ and $\mathcal{T}_\Delta(G_1', \mathsf{SCC}, \mathsf{SCC}(G_1'), \Delta_2)$ should behave the same, as the information in G and G_1' are not different enough to trigger different actions of $\mathcal{T}_\Delta$. However, the outputs $\mathsf{SCC}(G_2)$ and $\mathsf{SCC}(G_3')$ are different, in which only two connected components are in G_3' while G_2 has three. Thus a contradiction.

KWS. We now study the incremental problem of KWS.

(1) Insertions. We first prove that the incremental problem for KWS is unbounded under a unit edge insertion by Δ-reduction from ISO. It is known that the incremental problem for ISO remains unbounded under unit edge insertions [FWW13] when the pattern query Q is a tree.

The Δ-reduction (f, f_i, f_o) is defined as follows.

(a) Given an instance $I_1 = (Q_1, G_1)$ of ISO, where Q_1 is a tree rooted at node r, f computes an instance $I_2 = ((Q_2, b), G_2)$ of KWS such that subgraph G_s is a match of Q_1 in G_1 iff G_s' is a match to Q_2 in G_2 bounded by b, where G_s' is the corresponding subgraph of G_s in G_2. More specifically, we assume that all the nodes in $Q_1 = (V_{Q_1}, E_{Q_1}, l_{Q_1})$ have distinct labels, without affecting the correctness of the proof in [FWW13]. We define Q_2 as the collection of node labels in Q_1, *i.e.*, $Q_2 = \{l_{Q_1}(v) \mid v \in V_{Q_1}\}$. The bound b for Q_2 is the *longest* shortest distance from root r to other nodes in Q_1, *i.e.*, $b = \max_{v \in V_{Q_1}} \mathrm{dist}(r, v)$. Each node v in $G_1 = (V_{G_1}, E_{G_1}, l_{G_1})$ is mapped to v' in $G_2 = (V_{G_2}, E_{G_2}, l_{G_2})$, and $l_{G_1}(v) = l_{G_2}(v')$. For each edge (v_i, v_j) in G_1, (v_i', v_j') is included to E_{G_2} if and *only if* there exists an edge (u_i, u_j) in Q_1 such that $l_{Q_1}(u_i) = l_{G_1}(v_i)$ and $l_{Q_1}(u_j) = l_{G_1}(v_j)$.

(b) We map $\mathsf{insert}(v_i, v_j)$ in ΔG_1 to $\mathsf{insert}(v_i', v_j')$ to G_2 using function f_i if and only if there exists an edge (u_i, u_j) in Q_1 such that $l_{Q_1}(u_i) = l_{G_1}(v_i)$ and $l_{Q_1}(u_j) = l_{G_1}(v_j)$.
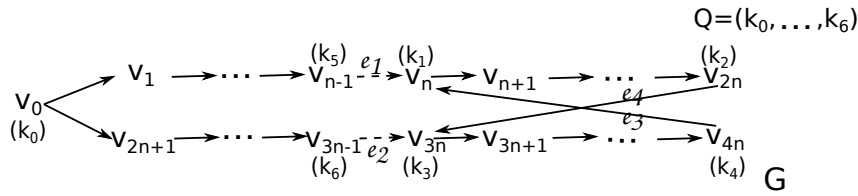
Figure 2.2: Unboundedness for KWS

Graph G_2 remain unchanged under other cases.

(c) Observe that edge insertion can only lead to new matches generated for both the two problems. Hence function f_o computes ΔO_1 for a given ΔO_2 that consists of new matches in $Q_2(G_2 \oplus \Delta G_2)$. More specifically, each new match G_s' in ΔO_2 is transformed to a new match G_s of Q_1 in $G_1 \oplus \Delta G_1$, where G_s is composed of the corresponding nodes of those in G_s' with the edges connecting them in $G_1 \oplus \Delta G_1$.

We next prove that functions f_o and f_i are one-to-one. First suppose that G_s is a match in $Q_1(G_1 \oplus \Delta G_1)$. Denote by G_s' the corresponding subgraph in $G_2 \oplus \Delta G_2$. Observe that (i) all keywords from Q_2 are covered by the node labels in G_s' as Q_2 is obtained by using the labels in Q_1; and (ii) the sum of distances from root r' of G_s' to those nodes containing keywords in G_s' is minimum among such trees rooted at r'; this is because the structure of the matches in $Q_2(G_2 \oplus \Delta G_2)$ with bound b is restricted by that of $Q_1(G_1 \oplus \Delta G_2)$, and hence any change to it will lead to v_i' unreachable from r' for some node v_i' matching keyword k_i in G_s'; this can formally be proved by induction on the size of Q_1. Conversely, if G_s' is a match to Q_2 in $G_2 \oplus \Delta G_2$, then one can verify that G_s is in $Q_1(G_1 \oplus \Delta G_1)$ also by induction on the size of Q_1.

Moreover, f_o and f_i take linear time in $|\Delta G_1| + |\Delta O_1|$ and $|Q_1|$, since f_o is a simple one-to-one mapping and f_i cheeks the labels in Q_1 for ΔG_1. Hence (f, f_i, f_o) is indeed a Δ-reduction from ISO, and KWS is unbounded in this case.

(2) Deletions. We show that the incremental problem for KWS is also unbounded under a unit edge deletion. Consider an instance of KWS shown in Fig. 2.2, where Q is a list $(k_0, \ldots, k_6)$ of 7 keywords. Graph $G = (V, E, l)$ consists of two paths $(v_0, v_1, \ldots, v_{2n})$ and $(v_0, v_{2n+1}, \ldots, v_{4n})$ of length $2n$, and there are two edges $e_3 = (v_{4n}, v_n)$ and $e_4 = (v_{2n}, v_{3n})$ connecting them. Node v_{i*n} carries label k_i for $i \in [0, 4]$, $l(v_{n-1}) = k_5$ and $l(v_{3n-1}) = k_6$. Other nodes in V carry label k' that does not occur in Q. Bound b is defined as $3n$. Denote by Δ_1 the deletion of edge $e_1 = (v_{n-1}, v_n)$, and by Δ_2 the deletion of $e_2 = (v_{3n-1}, v_{3n})$. We define graph $G_1 = G \oplus \Delta_1$, $G_2 = G \oplus \Delta_2$ and $G_3 = G_1 \oplus \Delta_2$.

It can be verified that $Q(G)$ has a single tree rooted at v_0, which can be obtained from G by removing two edges e_3 and e_4. The matches $Q(G_1)$ and $Q(G_2)$ can be constructed from $Q(G)$ with edge e_1 replaced by e_3 and e_2 replaced by e_4 respectively, and hence the size of the differences between $Q(G)$ and $Q(G_1)$ or $Q(G_2)$ is 2. However, $Q(G_3) = \emptyset$, *i.e.,* these exist changes of $O(n)$ size compared to $Q(G)$.

Assume by contradiction that there exists a bounded incremental algorithm $\mathcal{T}_\Delta$ for KWS under a unit edge deletion. Then $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_1)$ and $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_2)$ both take $O(1)$ time as the changes to the inputs and outputs are constants and $|Q| = 7$. Consider the executions of $\mathcal{T}_\Delta(G, Q, Q(G), \Delta_2)$ and $\mathcal{T}_\Delta(G_1, Q, Q(G_1), \Delta_2)$, having different outputs. The different behaviors of $\mathcal{T}_\Delta$ on G and G_1 when processing Δ_2 can be caused only if their corresponding traces $T_s(G, \Delta_2)$ and $T_s(G_1, \Delta_2)$ contain some node associated with different information in G and G_1, respectively. Since graph G_1 is the same as G except the deletion of edge e_1, these nodes are contained in trace $T_s(G, \Delta_1)$ with information updated during the processing of Δ_1 on G.

Denote by v_d the first node associated with different data in $T_s(G, \Delta_2)$ and $T_s(G_1, \Delta_2)$, which is also contained in $T_s(G, \Delta_1)$. Then the nodes on some undirected path from the position of e_1 to v_d must be visited during the execution of locally persistent algorithm $\mathcal{T}_\Delta$ to process Δ_1 on G. On the other hand, v_d must also be inspected when processing Δ_2 on G_1 and G; hence all the nodes on an undirected path from the position of e_1 to that of e_2 should be inspected by $\mathcal{T}_\Delta$ to process Δ_1 and Δ_2 on G. The length of this path is $O(n)$, which contracts the assumption that the two updates could both be processed in constant time. $\qquad\square$

2.3 Localizable Incremental Computations

Not all is lost. Despite Theorem 2.1, there exist efficient incremental algorithms for RPQ, SCC, KWS and ISO with performance guarantees under new characterizations for the effectiveness of incremental algorithms. In this section we introduce one of the standards, namely, localizable incremental computations. We first present the notion (Section 2.3.1). We then show that the incremental problems for KWS and ISO are localizable (Section 2.3.2 and 2.3.3, respectively).

2.3.1 Locality of Incremental Computations

We start with a few notations. (a) In a graph G, we say that a node v' is *within d hops* of v if $\text{dist}(v, v') \leq d$ by taking G as an undirected graph. (b) We denote by $V_d(v)$ the

set of all nodes in G that are within d hops of v. (c) The *d-neighbor* $G_d(v)$ of v is the subgraph of G induced by $V_d(v)$, in which the set of edges is denoted by $E_d(v)$.

Consider a graph query class Q. An incremental algorithm $\mathcal{T}_\Delta$ for Q is *localizable* if its cost is determined only by $|Q|$ and the sizes of the d_Q-neighbors of those nodes on the edges of ΔG, where d_Q is determined by the query size $|Q|$.

The incremental problem for Q is called *localizable* if there exists a localizable incremental algorithm for Q.

Intuitively, if $\mathcal{T}_\Delta$ is localizable, it can compute ΔO by inspecting only $G_{d_Q}(v)$, *i.e.,* nodes within d_Q hops of nodes v in ΔG. In practice, $G_{d_Q}(v)$ is often small. Indeed, (a) Q is typically small; *e.g.,* 98% of real-life pattern queries have radius 1, and 1.8% have radius 2 [GFMPdlF11]; hence so is d_Q; and (b) real-life graphs are often sparse; for instance, the average node degree is 14.3 in social graphs [BW13]. Hence, $\mathcal{T}_\Delta$ can reduce the computations on possibly big G to small $G_{d_Q}(v)$.

The main results of this section are as follows.

Theorem 2.3: *The incremental problem is localizable for* KWS *and* ISO *under batch updates.*   □

That is, while the incremental problems for KWS and ISO are unbounded, we can still effectively conduct their incremental computations by making big graphs "small".

As a constructive proof of Theorem 2.3, we next develop localizable incremental algorithms for KWS. The incremental algorithms for ISO are similar and are outlined in Section 2.3.3.

2.3.2   Localizable Incremental Algorithms for KWS

We first provide localizable algorithms for KWS under unit edge insertions and deletions. We then develop a localizable incremental algorithm for KWS to process batch updates.

Data structures. We start with an auxiliary structure. Recall that a KWS query consists of a list $Q = (k_1, \ldots, k_m)$ of keywords and an integer bound b. For each node v in graph G, we maintain a *keyword-distance* list kdist(v). Its entries are of the form (keyword, dist, next), where dist is the shortest distance from v to a node labeled keyword in Q, and next indicates the node on this shortest path next to v. A single shortest path is selected with a predefined order in case of a tie. Hence each root uniquely determines a match if it exists. During the traversal of G, kdist$(\cdot)$'s

---

Algorithm: IncKWS$^+$

Input: A graph G with kdist$(\cdot)$, keyword query Q and bound b,
        matches $Q(G)$, and an edge (v, w) to be inserted.
Output: The updated matches $Q(G \oplus \Delta G)$ and kdist lists.

1. **for each** k_i in Q with
        kdist$(w)[k_i]$.dist $< \min(\text{kdist}(v)[k_i].\text{dist} - 1, b)$ **do**
2.   kdist$(v)[k_i]$.dist $:= \text{kdist}(w)[k_i].\text{dist} + 1$;
3.   kdist$(v)[k_i]$.next $:= w$; queue $q_i := nil$; q_i.enqueue(v);
4.   **while** q_i is not empty **do**
5.   node $u := q_i$.dequeue$()$;
6.   **for each** predecessor u' of u such that
          kdist$(u)[k_i]$.dist $< \min(\text{kdist}(u')[k_i].\text{dist} - 1, b)$ **do**
7.   kdist$(u')[k_i]$.dist $:= \text{kdist}(u)[k_i].\text{dist} + 1$;
8.   kdist$(u')[k_i]$.next $:= u$; q_i.enqueue(u');
9. **for each** u_1'' and u_2'' involved in a changed kdist$(u)[k_i]$.next **do**
10.   replace (u, u_1'') with (u, u_2'') in all the matches of $Q(G)$ or
      add matches to $Q(G \oplus \Delta G)$ by including (u, u_2'');
11. **return** $Q(G \oplus \Delta G)$ (including revised $Q(G)$) and kdist$(\cdot)$;

Figure 2.3: Algorithm IncKWS$^+$

are updated, and $Q(G)$ is generated using these lists. Such keyword-distance lists are obtained after the execution of a batch algorithm. Indeed, existing batch approaches [HWYY07, BHN$^+$02, KPC$^+$05] for KWS traverse G to find shortest paths from nodes to others matching keywords in Q. While they vary in search and indexing strategies, they all maintain something like kdist$(\cdot)$.

(1) Unit insertions. Inserting an edge to graph G may shorten the shortest distances from nodes to those matching keywords in Q, which is reflected as changes to dist and next in the keyword-distance lists on G. Based on this, we present an incremental algorithm, referred to as IncKWS$^+$ and shown in Fig. 2.3, to process unit edge insertions.

Given ΔG consisting of insert(v, w), IncKWS$^+$ inspects whether it inflicts any change to shortest paths of existing matches; if so, it propagates the changes, revises kdist(v) entries for affected nodes v and updates the matches accordingly. It proceeds
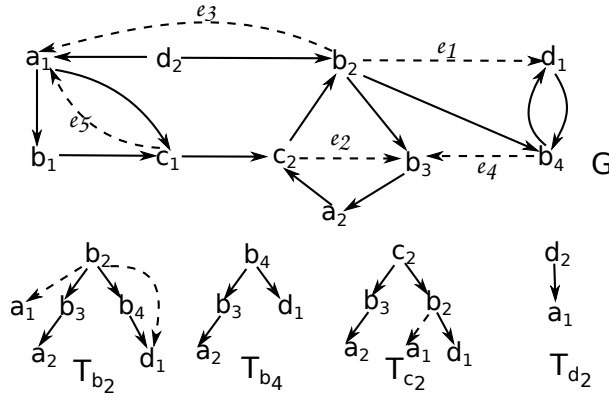
Figure 2.4: Example graph and matches of KWS

until no more revision is needed. The search is confined in the b-neighbors of nodes in ΔG, and hence localizable, where b is the bound in the KWS query. This is achieved by updating $\mathsf{kdist}(\cdot)$ only when there exists some shortest path of length within bound b being affected.

More specifically, IncKWS^+ first checks whether (v, w) is on a shorter path within the bound b from v to nodes labeled k_i in Q (line 1). If so, $\mathsf{kdist}(v)$ is adjusted by updating dist and next (lines 2-3). IncKWS^+ then propagates the change to the ancestors of v if their $\mathsf{kdist}(\cdot)$ entries are no longer valid, and updates the entries accordingly (lines 4-8). An FIFO (first-in-first-out) queue q_i is used to control the propagation, following BFS (breadth-first-search). Each time when a node u is dequeued from q_i (line 5), the predecessors of u are inspected to check whether u triggers updated shortest path from them within bound b, followed by updating their kdist entries if needed (lines 6-8). These predecessors may be inserted into queue q_i for further checking (line 8).

After revising the data structures, IncKWS^+ computes $Q(G \oplus \Delta G)$ based on the changes to next in $\mathsf{kdist}(\cdot)$ (lines 9-10), either by replacing some edges in existing matches, or by including new matches not in $Q(G)$. Note that all such affected edges are inside the $2b$-neighbors of ΔG.

Example 2.1: Figure 2.4 gives a graph G (with all solid edges and dotted e_2, e_5). Consider a KWS query $Q = (a, d)$ and bound 2. Two trees T_{b_2} and T_{d_2} in $Q(G)$ are shown in Fig. 2.4 (with solid edges only), rooted at b_2 and d_2, respectively.

When edge e_1 is added to G, denote by G_1 the graph after the insertion. IncKWS^+ finds that the shortest distance from b_2 to nodes matching d in G_1 is reduced to 1 from 2. Thus it updates the entries in $\mathsf{kdist}(b_2)[d]$ and propagates the change to b_2's predecessors c_2 and d_2. The propagation stops at c_2 since the shortest distance from

it to d nodes already reaches bound 2. The values of $\langle\text{dist},\text{next}\rangle$ in keyword-distance lists on G are updated, as shown below.

IncKWS$^+$	before insertion	after insertion
kdist$(b_2)[d]$	$\langle 2, b_4 \rangle$	$\langle 1, d_1 \rangle$
kdist$(c_2)[d]$	$\langle \perp, nil \rangle$	$\langle 2, b_2 \rangle$

Then IncKWS$^+$ revises T_{b_2} by replacing the path starting with edge (b_2, b_4) by (b_2, d_1) to get T'_{b_2} in $Q(G_1)$, and a new match T_{c_2} (solid edges in Fig. 2.4) is added to $Q(G_1)$. $\square$

Correctness & complexity. IncKWS$^+$ updates kdist$(\cdot)$ correctly: it revises only entries in which dist values are decreased, and checks all affected entries by propagating the changes. From this the correctness of IncKWS$^+$ follows.

IncKWS$^+$ is in $O(m(|V_b(w)| + |E_b(w)|) + |V_b(w)||E_{2b}(w)|)$ time. Updating kdist$(\cdot)$ takes $O(m(|V_b(w)| + |E_b(w)|))$ time in total (lines 1-8), where m is the number of keywords in Q. Observe the following: (a) each node with updated kdist is verified at most m times, one for each keyword k_i in Q to check the shortest path to nodes labeled k_i; and (b) only the data in $G_b(w)$ is inspected since change propagation stops as soon as the shortest distance exceeds b (lines 1, 6), *i.e.,* kdist$(\cdot)$ is partially updated for matches within bound b. Updating $Q(G)$ (lines 9-10) takes $O(|V_b(w)||E_{2b}(w)|)$ time since the roots of the affected matches are within b hops of w, and their edges to be adjusted are at most $2b$ hops away from w. Therefore, algorithm IncKWS$^+$ is localizable.

(2) Unit deletions. The incremental algorithm for processing unit delete(v, w) is shown in Fig. 2.5, denoted by IncKWS$^-$. In contrast to edge insertions, some shortest distances in kdist lists may be increased by delete(v, w). As a result, matches in $Q(G)$ will be changed or even removed. The main idea of IncKWS$^-$ is to identify those entries in kdist$(\cdot)$ that are *affected* by ΔG, and compute changes to dist and next. Similar to IncKWS$^+$, updating kdist$(\cdot)$'s is confined within the b-neighbors of ΔG by inspecting only those distances no longer than b. The identification and computation are separated into two phases in IncKWS$^-$.

After consulting whether (v, w) is on a shortest path from v to some node labeled keyword k_i within bound b (line 1), IncKWS$^-$ propagates the change to v's predecessors if needed with the help of a stack a_i, and each predecessor that may have an updated shortest path to nodes matching k_i is marked *affected w.r.t.* k_i (lines 3-6). The

---

Algorithm: IncKWS$^-$

Input: G with kdist$(\cdot)$, Q, b, $Q(G)$ as in IncKWS$^+$, and delete(v,w).

Output: The updated matches $Q(G \oplus \Delta G)$ and kdist lists.

1. **for each** k_i in Q with $w = $ kdist$(v)[k_i]$.next

     and kdist$(w)[k_i] < b$ **do**

2.   queue $q_i := nil$; stack $a_i := nil$; a_i.push(v); mark v *affected*;

3.   **while** a_i is not empty **do**

4.   node $u := a_i$.pop$()$;

5.   **for each** predecessor u' of u that $u = $ kdist$(u')[k_i]$.next

            and kdist$(u')[k_i] \leq b$ **do**

6. a_i.push(u'); mark u' *affected*;

7.   **for each** *affected* node u **do**

8.   compute dist and next for kdist$(u)[k_i]$ based on those

           u's successors that are *not affected*;

9. q_i.insert$(u, $kdist$(u)[k_i]$.dist$)$;

10.   **while** q_i is not empty **do**

11. $(u,d) := q_i$.pull_min$()$;

12.   **for each** predecessor u' of u with

          $d < \min($kdist$(u')[k_i]$.dist$-1, b)$ **do**

13.   kdist$(u')[k_i]$.dist $:= d + 1$; kdist$(u')[k_i]$.next $:= u$;

14. q_i.decrease$(u', $kdist$(u')[k_i]$.dist$)$;

15. **for each** u_1'' and u_2'' involved in a changed kdist$(u)[k_i]$.next **do**

16.   replace (u, u_1'') with (u, u_2'') in all the matches of $Q(G)$ or

      remove matches from $Q(G)$ by excluding (u, u_1'');

17. **return** $Q(G \oplus \Delta G)$ (updated $Q(G)$ above) and kdist$(\cdot)$;

Figure 2.5: Algorithm IncKWS$^-$

propagation is similar to that of IncKWS$^+$, by inspecting next values in kdist$(\cdot)$'s, and is conducted in b-neighbors of v. Then the *potential* dist and next values for kdist on those affected nodes are computed based on their successors that are *not affected w.r.t.* k_i (line 8), and affected nodes with their potential dist values (as the keys) are inserted into a priority queue q_i (line 9) to compute the exact dist values later. Indeed, the exact values of dist and next may depend on the affected successors, whose values also need

to be determined.

The exact values of dist and next are computed in the second phase of IncKWS^- (lines 10-14). For node u with minimum dist that is removed from q_i (line 11), IncKWS^- checks whether it leads to a new shortest path within bound b originated from predecessor u' of u, as described for IncKWS^+ (line 12). If so, dist and next in $\mathsf{kdist}(u')[k_i]$ are updated (line 13), and the key of u' in q_i is decreased (line 14).

The process continues until q_i becomes empty. Matches in $Q(G)$ are updated using the latest $\mathsf{kdist}(\cdot)$ lists (lines 15-16).

Example 2.2: Recall Q, G_1 and $Q(G_1)$ from Example 2.1. Suppose that e_2 is now removed from G_1. This makes the shortest path from c_2 to a_2 in T_{c_2} split, and IncKWS^- marks node c_2 affected with keyword a. Since the shortest distance from successor b_2 of c_2 to nodes matching a equals the bound 2, IncKWS^- concludes that node c_2 cannot be the root of a match, and removes T_{c_2} of Example 2.1 from $Q(G_1)$. □

Correctness & complexity. The correctness of IncKWS^- is verified just like for IncKWS^+, except that the exact values of $\mathsf{kdist}(v)$ may depend on multiple affected successors of v.

IncKWS^- runs in $O(m(|V_b(w)|\log|V_b(w)| + |E_b(w)|) + |V_b(w)||E_{2b}(w)|)$ time, including $O(|V_b(w)||E_{2b}(w)|)$ for updating matches in $Q(G)$ in addition to the cost for computing changes to $\mathsf{kdist}(\cdot)$'s. Its first phase (lines 1-9) takes $O(m(|V_b(w)| + |E_b(w)|))$ time since only the affected shortest paths of length bounded by b are identified. Its second phase (lines 10-14) takes $O(m(|V_b(w)|\log|V_b(w)| + |E_b(w)|))$ time, the same as computing b-bounded shortest path from each affected node to a single sink, *i.e.,* nodes labeled a specific keyword from Q, within the b-neighbors of updated (u, v).

(3) Batch updates. We next give an incremental algorithm, denoted by IncKWS (not shown), to process batch updates $\Delta G = (\Delta G^+, \Delta G^-)$, where ΔG^+ and ΔG^- denote edge insertions and deletions, respectively. We assume *w.l.o.g.* that there exist no delete e in ΔG^- and insert e in ΔG^+ for the same edge e, which can be easily detected.

Given batch updates ΔG, IncKWS inspects whether each unit edge deletion and insertion causes any change to existing matches, *i.e.,* whether some of existing shortest paths become invalid and new shortest paths within bound b have to be generated; if so, it propagates the changes and updates entries of keyword-distance lists of those

affected nodes. The algorithm updates the same entry at most once even if it is affected by multiple updates in ΔG, by *interleaving* different change propagation with the help of a global data structure to accommodate the effects of different unit updates. It works in three phases, as outlined below.

(a) IncKWS first identifies the affected nodes *w.r.t.* each keyword k_i in Q due to ΔG^- within the b-neighbors of ΔG^-, and computes their potential dist and next values, using the same strategy of IncKWS$^-$. Here all the affected nodes *w.r.t.* k_i and their potential dist values are inserted into a *single* priority queue q_i to further compute exact values.

(b) The algorithm then checks whether each insert(v, w) leads to the creation of a shorter path within bound b when neither v nor w is affected *w.r.t.* k_i by ΔG^-. Insertions with affected nodes are not considered since dist value at w may no longer be correct due to ΔG^-, or this edge has already been inspected to compute potential dist value for node v. If so, dist and next values are updated for the keyword-distance list on v. Unlike IncKWS$^+$ that propagates this change to ancestors of v directly, it inserts node v and the updated dist value into queue q_i to interleave insert(v, w) with other updates in ΔG when computing exact values.

(c) After these, IncKWS computes exact next and dist values of kdist$(\cdot)$, in the same way as we do in IncKWS$^-$ by making use of queue q_i. Note that all potential changes to kdist$(\cdot)$ caused by ΔG, including both deletions and insertions, are collected into the same q_i; in this way the algorithm guarantees that the exact value, *i.e.,* shortest distance, is decided at most once for each entry affected. Matches in $Q(G)$ are updated accordingly within the $2b$-neighbors of ΔG at last.

Example 2.3: Consider Q and G of Example 2.1, and batch updates ΔG that insert edges e_1, e_3, e_4 and delete e_2 and e_5.

   Given these, algorithm IncKWS first identifies the affected nodes c_1 and c_2 *w.r.t. a*, and finds that the potential value of the corresponding dist already exceeds the bound 2. Then it processes insertions; *e.g.,* the insertion of e_3 leads to decreased shortest distance from b_2 to a nodes, and the change is propagated to b_2's predecessor c_2 to compute the exact value of kdist$(c_2)[a]$, *i.e.,* IncKWS interleaves the insertion of e_3 and deletion of e_2 to decide the exact shortest distance from c_2 to a nodes. The other updates are handled similarly. Based on these, it replaces the two branches of T_{b_2} with (b_2, a_1) and (b_2, d_1), respectively, and adds match T_{b_4} in Figure 2.4. A new match T'_{c_2} is also included in $Q(G \oplus \Delta G)$, where path (c_2, b_3, a_2) in T_{c_2} of Example 2.1 is replaced

by (c_2, b_2, a_1).   □

Correctness & complexity. For the correctness of IncKWS, observe the following. (a) Each node that is affected *w.r.t.* keyword k_i by any unit update in ΔG is inspected. (b) The dist values for these nodes are monotonically increasing and correctly computed, similar to its counterpart in IncKWS$^-$.

IncKWS is in $O(m(|V_b(\Delta G)|\log|V_b(\Delta G)| + |E_b(\Delta G)|) + |V_b(\Delta G)||E_{2b}(\Delta G)|)$ time, where $V_b(\Delta G)$ (resp. $E_b(\Delta G)$) denote the nodes (resp. edges) of the union of b-neighbors of nodes involved in ΔG. Note that the final value of each affected node *w.r.t.* any keyword k_i is determined once by using the global priority queue q_i. The complexity analysis is similar to that of IncKWS$^-$, except that here the $2b$-neighbors of all the nodes involved in ΔG are possibly accessed for updating the matches and kdist$(\cdot)$'s.

Since the costs of IncKWS$^+$, IncKWS$^-$ and IncKWS are determined by m and the size of $2b$-neighbors of nodes involved in ΔG for a given bound b, they are all localizable.

Remark. Although the incremental algorithms for KWS are developed for a constant b, they can be readily extended to cope with b that varies. More specifically, when change propagation stops at node v due to the restriction of bound b, we can annotate v as a "breakpoint" *w.r.t.* b, and the set of all such breakpoints is stored as a "snapshot" of graph G *w.r.t.* b. When given a larger b', the snapshot of G *w.r.t.* b is firstly restored and each breakpoint is regarded as a unit update, either insertion or deletion, to the data graph, *i.e.,* as input to the incremental algorithm with b' in addition to ΔG, from where the change propagation continues. In this way, KWS queries with different b values can be answered using the same data structure, *i.e.,* keyword-distance list that is consistently updated. Indeed, we only need to store the snapshot of G *w.r.t.* the maximum b that is encountered.

2.3.3   Localizable Incremental Algorithms for ISO

Recall that given a pattern query Q and a graph G, ISO is to compute the set $Q(G)$ of all matches of Q in G, *i.e.,* all subgraphs of G that are isomorphic to Q. Observe that the deletion of an edge e may cause the removal of matches that include e from $Q(G)$. Conversely, insertion of $e = (v, w)$ may add new matches to $Q(G)$ and all these matches are within $G_{d_Q}(v)$ and $G_{d_Q}(w)$, where d_Q is the length of the longest shortest

path between any two nodes in Q when taken as undirected graph, *i.e.,* the *diameter* of Q.

Based on this, we outline a localizable incremental algorithm, denoted by IncISO, for ISO under batch updates (not shown). It works as follows. (1) Collect the set ΔG^- of all edge deletions in ΔG. For each edge deletion of e, remove those matches including e from $Q(G)$, by inspecting the d_Q-neighbors of the two nodes on e, where d_Q is the diameter of Q. (2) For the rest of updates in ΔG, *i.e.,* edge insertions ΔG^+, extract the union of d_Q-neighbors of the nodes involved in these edge insertions, denoted by $G_{d_Q}(\Delta G^+)$. (3) Invoke an existing batch algorithm (*e.g.,*VF2 [CFSV04]) for ISO to compute $Q(G_{d_Q}(\Delta G^+))$ all together rather than one by one, and add those matches to $Q(G)$ that are not in $Q(G)$.

Obviously, the cost of IncISO can be expressed as a function of $|Q|$ and $|G_{d_Q}(\Delta G)|$, instead of the size $|G|$ of the entire graph G. In other words, IncISO is localizable, and hence so is ISO. Note that $G_{d_Q}(\Delta G)$ also includes the d_Q-neighbors of nodes involved in edge deletions.

Putting this together with the algorithms presented in Sections 2.3.2, we complete the proof of Theorem 2.3.

In our experimental study, we compare IncISO with another algorithm IncISO$_n$, which applies the batch algorithm on d_Q-neighbor of each update one by one.

2.4   Relatively Bounded Incrementalization

We next introduce relative boundedness, another alternative characterization for the effectiveness of incremental computations. We first formalize the notion in Section 2.4.1. We then develop relatively bounded incremental algorithms for RPQ and SCC in Sections 2.4.2 and 2.4.3, respectively.

2.4.1   Relative Boundedness

Consider a batch algorithm $\mathcal{T}$ for a query class $\mathcal{Q}$ that is proven effective and being widely used in practice. For a query $Q \in \mathcal{Q}$ and a graph G, we denote by $G_{(\mathcal{T},Q)}$ the data inspected by $\mathcal{T}$ when computing $Q(G)$, including data in G and possibly auxiliary structures used by $\mathcal{T}$. For updates ΔG to G, we denote by AFF the difference between $(G \oplus \Delta G)_{(\mathcal{T},Q)}$ and $G_{(\mathcal{T},Q)}$, *i.e.,* the difference in the data inspected by $\mathcal{T}$ for computing $Q(G \oplus \Delta G)$ and for $Q(G)$.

An incremental algorithm $\mathcal{T}_\Delta$ for Q is *bounded relative to* $\mathcal{T}$ if its cost can be expressed as a polynomial function in $|\Delta G|$, $|Q|$ and $|\mathsf{AFF}|$ for $Q \in \mathcal{Q}$, graph G and updates ΔG. Note that the changes ΔO to $Q(G)$ are included in AFF.

Intuitively, we only incrementalize batch algorithms $\mathcal{T}$'s that have been verified effective. As batch algorithms have been studied for decades for graphs, a number of such algorithms are in place. When incrementalizing such algorithms, relative boundedness is to characterize the effectiveness of the incrementalization, *i.e.,* whether it minimizes unnecessary recomputation in response to updates ΔG. It suffices to develop $\mathcal{T}_\Delta$ bounded relatively to one of such $\mathcal{T}$'s.

Note that for a class $\mathcal{Q}$ of graph queries, one can find localizable incremental algorithms only if $\mathcal{Q}$ has the data locality, *i.e.,* to decide whether v is in the answer $Q(G)$ to a query Q, it suffices to inspect the d_Q-neighbor of v. However, many graph queries do not have the data locality, *e.g.,* RPQ and SCC. For such queries, we can explore relatively bounded incremental algorithms. Moreover, even when $\mathcal{Q}$ has the data locality, we want to find incremental algorithms that are both localizable and bounded relative to a practical batch algorithm of $\mathcal{Q}$. Such algorithms are particularly needed for large queries Q (*i.e.,* when diameter d_Q of Q is large).

We should remark that there are other alternative effectiveness characterizations for incremental graph algorithms, *e.g.,* a classification in terms of incremental complexity. We focus on localizability and relative boundedness in this chapter since they are easy to verify and use in practice.

The main results of this section are as follows.

Theorem 2.4: *There are bounded incremental algorithms for* RPQ *and* SCC *relative to their batch counterparts.* □

As a proof, we present relatively bounded algorithms for RPQ and SCC. As will be seen in Section 2.5, these algorithms are effective although none of the query classes is bounded.

2.4.2 Incrementalization for RPQ

We start with RPQ. Given a regular path query Q and a graph G, it is to compute the set $Q(G)$ of matches of Q in G, *i.e.,* pairs (v, w) of nodes in G such that v can reach w by following a path in the regular language defined by Q.

We incrementalize a batch algorithm $\mathsf{RPQ_{NFA}}$ [HSW01, MW95] for RPQ. We first
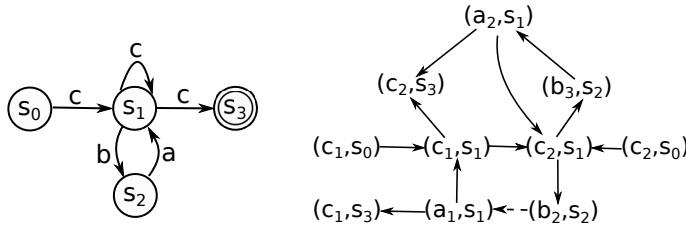
Figure 2.6: NFA M_Q and intersection graph of M_Q, G

review RPQ_{NFA} and identify its AFF. We then give a bounded incremental algorithm relative to RPQ_{NFA}.

Batch algorithm. Algorithm RPQ_{NFA} consists of two phases. Given Q and G, it first translates Q into an NFA M_Q (nondeterministic finite automaton) [HSW01], and then computes $Q(G)$ by traversing graph G based on the automaton M_Q [MW95]. Its time complexity is $O(|V||E||Q|^2 \log^2 |Q|)$.

More specifically, $M_Q = (S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is the *alphabet*, δ is the *transition function* that maps $S \times \Sigma$ to the set of subsets of S, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is the set of *accepting states*. There are other methods for constructing NFA, *e.g.,* the one based on partial derivatives [Ant96]. We adopt the algorithm of [HSW01] since it constructs smaller NFA than [Ant96] and takes less time.

After M_Q is in place, the second phase starts, traversing the *intersection graph* $G_I = (V_I, E_I, l_I)$ of G and M_Q [MW95]. Here $V_I = V \times S$, $l_I(v,s) = l(v)$, $E_I \subseteq V_I \times V_I$ and $((v,s),(v',s'))$ is in E_I if and only if $(v,v') \in E$ and $s' \in \delta(s,l(v'))$. RPQ_{NFA} conducts BFS from each node in G_I. Each node v in G is *marked* with a set $v.\text{pmark}(\cdot)$ of *markings*, where $v.\text{pmark}(u)$ is a set of states s in S, indicating that there exists a path ρ from u to v in G such that (u,s_0) reaches (v,s) following the corresponding path ρ_I of ρ in G_I. When node v is visited in state s, only the successor v' of v with $\delta(s,l(v')) \neq \emptyset$ are inspected in G. The markings prevent a node from being visited more than once in the same state. It includes (u,v) in $Q(G)$ if $v.\text{pmark}(u) \cap F \neq \emptyset$, *i.e.,* there exist state $s \in v.\text{pmark}(u)$ and a path ρ_I from (u,s_0) to (v,s) such that $l_I(\rho_I) \in L(Q)$.

Example 2.4: Consider an RPQ query $Q = c \cdot (b \cdot a + c)^* \cdot c$ over the graph G of Fig. 2.4. Its NFA M_Q and a fragment of the intersection graph G_I of G and M_Q are shown in Fig. 2.6 (excluding dotted edge $((b_2,s_2),(a_1,s_1))$).

RPQ_{NFA} traverses G_I and marks the nodes in G with states of M_Q. Note that there exist paths from (c_1,s_0) to (c_2,s_3) and from (c_2,s_0) to (c_2,s_3) in G_I; thus the accepting

state s_3 is included in markings $c_2.\mathsf{pmark}(c_1)$ and $c_2.\mathsf{pmark}(c_2)$. Therefore, (c_1, c_2) and (c_2, c_2) are returned by $\mathsf{RPQ_{NFA}}$.   □

Auxiliary structures. The marking $v.\mathsf{pmark_e}(u)$ is of the form $(\mathsf{state}, \mathsf{dist}, \mathsf{cpre}, \mathsf{mpre})$, where (a) dist is the shortest distance from (u, s_0) to (v, state) in G_I, (b) (v', s') is contained in $v.\mathsf{pmark_e}(u)[s].\mathsf{cpre}$ if there exists an entry in $v'.\mathsf{pmark_e}(u)$ for state s' such that $s \in \delta(s', l(v))$ and (v', v) is in G, *i.e.,* $v.\mathsf{pmark_e}(u)[s].\mathsf{cpre}$ stores predecessors of node (v, s) in G_I that are on a path starting from (u, s_0); and (c) (v', s') is in $v.\mathsf{pmark_e}(u)[s].\mathsf{mpre}$ if $v'.\mathsf{pmark_e}(u)[s'].\mathsf{dist} + 1 = v.\mathsf{pmark_e}[s].\mathsf{dist}$, *i.e.,* mpre keeps track of those predecessors on shortest paths. The auxiliary information is computed by $\mathsf{RPQ_{NFA}}$ without increasing its complexity.

Characterization of AFF. We identify AFF, *i.e.,* the difference between $G_{(\mathsf{RPQ_{NFA}}, \mathcal{Q})}$ and $(G \oplus \Delta G)_{(\mathsf{RPQ_{NFA}}, \mathcal{Q})}$, as changes to the markings. Indeed, the markings are the data that $\mathsf{RPQ_{NFA}}$ necessarily inspects, since updates to markings trigger different behaviors of $\mathsf{RPQ_{NFA}}$ when computing $Q(G \oplus \Delta G)$ and $Q(G)$. For instance, a change to dist in $v.\mathsf{pmark_e}(u)[s]$ indicates that (v, s) is reached in BFS through a different path from (u, s_0) and state s is included in $v.\mathsf{pmark}(u)$ in $\mathsf{RPQ_{NFA}}$ at a different level of the BFS tree.

Incremental algorithm. Based on markings, we develop incremental algorithms for RPQ that are bounded relative to $\mathsf{RPQ_{NFA}}$. The boundedness is accomplished by updating markings only when necessary, *i.e.,* when there exists corresponding difference between the data inspected by $\mathsf{RPQ_{NFA}}$. For unit edge deletions and insertions, the algorithms are similar to their counterparts for KWS (Section 2.3.2), guided by changes to dist. Below we just present an algorithm for processing batch updates $\Delta G = (\Delta G^+, \Delta G^-)$.

The algorithm is denoted as IncRPQ and shown in Fig. 2.7. It first invokes procedure identAff (not shown) to identify a set $\mathsf{aff_s}$ of (v, u, s) triples, where $v.\mathsf{pmark_e}(u)[s].\mathsf{dist}$ is no longer valid due to edge deletions ΔG^- (line 1). Similar to how $\mathsf{IncKWS^-}$ identifies affected entries of keyword-distance lists (Section 2.3.2), identAff checks the values of mpre and cpre in markings. For example, if $v.\mathsf{pmark_e}(u)[s].\mathsf{mpre}$ becomes empty, it checks whether (v, s) is in $v'.\mathsf{pmark_e}(u)[s'].\mathsf{mpre}$ for each successor v' of v and $s' \in \delta(s, l(v'))$. If so, (v, s) is removed, and identAff continues to check the successors of v'. Transition function δ of M_Q is needed here to decide the states of markings. After these, IncRPQ updates the

---

Algorithm: IncRPQ

Input: A graph G with $\mathsf{pmark}_e(\cdot)$, regular path query Q and
          NFA M_Q, matches $Q(G)$, and batch updates $(\Delta G^+, \Delta G^-)$.
Output: The updated matches $Q(G \oplus \Delta G)$ and markings $\mathsf{pmark}_e(\cdot)$.

1.   set $\mathsf{aff}_s := \mathsf{identAff}(G, \mathsf{pmark}_e(\cdot), \Delta G^-)$; queue $q := nil$;
2.   **for each** (v, u, s) in aff_s **do**
3.   update dist, mpre for $v.\mathsf{pmark}_e(u)[s]$ based on its cpre;
4. $q.\mathsf{insert}((v, u, s), v.\mathsf{pmark}_e(u)[s].\mathsf{dist})$;
5.   **for each** edge insertion of (v, w) in ΔG^+ **do**
6.   **if** edge (v, w) leads to a smaller $w.\mathsf{pmark}_e(u)[s].\mathsf{dist}$ for
             node u and state s **and** (v, u, s) is not in aff_s **then**
7.   update dist, mpre, cpre for $w.\mathsf{pmark}_e(u)[s]$;
8. $q.\mathsf{insert}((w, u, s), w.\mathsf{pmark}_e(u)[s].\mathsf{dist})$;
9.   update $\mathsf{pmark}_e(\cdot)$ based on queue q and NFA M_Q;
10. update $Q(G)$ to get $Q(G \oplus \Delta G)$;
11. **return** $Q(G \oplus \Delta G)$ and $\mathsf{pmark}_e(\cdot)$;

Figure 2.7: Algorithm IncRPQ

corresponding (potential) dist values of triples in aff_s based on the current cpre, *i.e.,* the remaining candidate predecessors after removing affected entries. These triples with new dist values are inserted into priority queue q (lines 2-4) for computing exact values of the markings later on.

Thereafter, IncRPQ processes insertions in ΔG^+ by checking whether they yield smaller dist values in some markings (lines 5-6), and update them accordingly (line 7). Again, the updated triples are added to queue q (line 8). IncRPQ determines exact markings based on queue q (line 9) following a monotonically increasing order of updated dist, similar to IncKWS, while NFA M_Q is used to guide the propagation. By grouping updated triples in queue q, the algorithm reduces redundant computations when processing ΔG.

Finally, given the updated markings, $Q(G \oplus \Delta G)$ is computed by taking new pairs of nodes marked with accepting states in F and removing invalid ones from $Q(G)$ (line 10). Algorithm IncRPQ returns $Q(G \oplus \Delta G)$ and revised markings for future processing

of subsequent updates (line 11).

Example 2.5: Recall batch updates ΔG to G from Example 2.3. These inflict the deletion of $((c_2, s_1), (b_3, s_2))$ and insertion of $((b_2, s_2), (a_1, s_1))$ to the intersection graph G_I of Example 2.4. IncRPQ first finds that triple (b_3, c_2, s_2) is affected by the deletion. The change is propagated to the decedents of (b_3, s_2) in G_I, and potential values of $\langle \text{dist}, \text{mpre} \rangle$ for affected entries are computed. After these, it decides exact values after processing insertions; some are shown below.

IncRPQ	before updates	after updates
$b_3.\text{pmark}_e(c_2)[s_2]$	$\langle 2, \{(c_2, s_1)\} \rangle$	$\langle \bot, nil \rangle$
$a_2.\text{pmark}_e(c_2)[s_1]$	$\langle 3, \{(b_3, s_2)\} \rangle$	$\langle \bot, nil \rangle$
$c_2.\text{pmark}_e(c_2)[s_3]$	$\langle 4, \{(a_2, s_1)\} \rangle$	$\langle 5, \{(c_1, s_1)\} \rangle$
$c_1.\text{pmark}_e(c_2)[s_3]$	$\langle \bot, nil \rangle$	$\langle 4, \{(a_1, s_1)\} \rangle$
$c_1.\text{pmark}_e(c_1)[s_3]$	$\langle \bot, nil \rangle$	$\langle 5, \{(a_1, s_1)\} \rangle$

Note that although the previous path from (c_2, s_0) to (c_2, s_3) is split due to edge deletion, accepting state s_3 remains in marking $c_2.\text{pmark}_e(c_2)$ since another path connecting these two nodes in G_I is formed as a result of insertions. Indeed, IncRPQ combines the processes for the two updates, *i.e.,* $\text{delete}(c_2, b_3)$ and $\text{insert}(b_2, a_1)$ to compute exact value of $c_2.\text{pmark}_e(c_2)[s_3]$. Based on the exact values, it adds (c_2, c_1) and (c_1, c_1) to obtain $Q(G \oplus \Delta G)$, as accepting state s_3 is included in the corresponding markings. □

Correctness & complexity. One can verify that IncRPQ correctly updates markings by induction on the number of changed entries. IncRPQ is in $O(|\text{AFF}| \log |\text{AFF}|)$ time. Indeed, (a) affected triples are added to set aff_s and queue q at most once by BFS traversal; (b) each of procedure identAff (line 1), computing potential values (lines 2-4) and processing edge insertions (lines 5-8) takes $O(|\text{AFF}|)$ time by using M_Q and cpre, where to compute potential values, $O(|\text{AFF}|)$ predecessors are processed directly via cpre, without inspecting the entire neighbors; and (c) computing the latest values of markings (line 9) needs $O(|\text{AFF}| \log |\text{AFF}|)$ time by using heaps for queue q, just like fixing dist values for affected nodes in IncKWS (Section 2.3.2). Note that $|Q|$ is counted in $|\text{AFF}|$. All these steps have costs bounded by a function of $|\text{AFF}|$. Hence IncRPQ is bounded relative to RPQ_{NFA}.

2.4.3 Incrementalization for SCC

We next investigate the incremental problem for SCC. Given a graph G, it is to compute SCC(G), *i.e.,* the set of all strongly connected components in G. In the sequel we abbreviate a strongly connected component as an scc.

We incrementalize Tarjan's algorithm [Tar72] for SCC. We refer to the batch algorithm as Tarjan. Below we first review the basic idea of Tarjan, and identify its AFF.

Batch algorithm. Tarjan traverses a directed graph G via repeated DFS (depth-first search) to generate a spanning forest $\mathcal{F}$, such that each scc corresponds to a subtree of a tree T in $\mathcal{F}$ with a designated *root*. It reduces SCC to finding the roots of corresponding subtrees in $\mathcal{F}$.

More specifically, each node v in G is assigned a unique integer v.num, denoting the order of v visited in the traversal. The edges of G fall into four classes by DFS: (a) *tree arcs* that lead to nodes not yet discovered during the traversal of G; (b) *fronds* that run from descendants to their ancestors in a tree T; (c) *reverse fronds* that are from ancestors to descendants in a tree; and (d) *cross-links* that run from one subtree to another. In addition, v.lowlink is maintained for each node v, representing the *smallest* num of the node that is in the *same* scc as v and is reachable by traversing zero or more tree arcs followed by *at most one* frond or cross-link. It determines whether node v is the root of the subtree corresponding to an scc by checking whether v.lowlink = v.num, and if so, generates the scc accordingly. It uses a stack to store nodes that have been reached during DFS but have not been placed in an scc. A node remains on the stack after it has been visited if and only if there exists a path in G from it to some node earlier on the stack (see [Tar72] for details).

Example 2.6: Figure 2.8 depicts the DFS forest $\mathcal{F}$ obtained by applying Tarjan on graph G of Fig. 2.4. Each node is annotated with its (num, lowlink). There are four scc's. The corresponding contracted graph G_c (see below) is also shown in Fig. 2.8 (solid edges), where node i refers to scc$_i$ in G.   □

Auxiliary structures. To incrementalize Tarjan, we maintain the values of num and lowlink of each node after traversing G, and annotate each edge with the type that it falls into. Besides, a *contracted graph* G_c of G is constructed by contracting each scc into a single node. The graph G_c maintains a counter for the number of cross-links from one node (*i.e.,* scc) to another. Each node v in G_c has a *topological rank* $r(v)$,
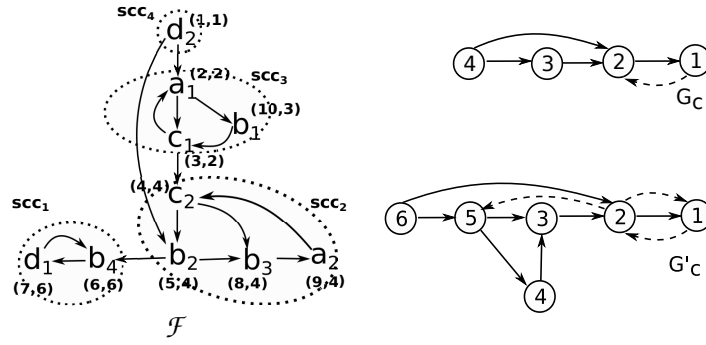
Figure 2.8: DFS forest of G and contracted graphs

initially the order of the scc to which v corresponds in the output sequence of Tarjan. Indeed, the topological sorting of scc's is a byproduct of Tarjan as nodes of each scc are popped from the stack recursively. These auxiliary structures can be computed by slightly revising Tarjan without increasing its complexity or changing its logic.

It is shown that $r(v) > r(v')$ if (v, v') is a cross-link in G_c [Tar72], an invariant property on which we will capitalize.

Characterization of AFF. The affected area AFF includes the following: (a) changes to lowlink and num of nodes when computing $\text{SCC}(G \oplus \Delta G)$, since accurate lowlink and num values determine the correctness of Tarjan; (b) v's successors for each node v whose v.lowlink changes, since the lowlink value of v is determined by comparing with lowlink or num of its successors; and (c) the neighbors of v for each node v whose v.num changes, since these neighbors are affected in this case and are necessarily checked by Tarjan.

We next give bounded incremental algorithms relative to Tarjan, under unit insertions, deletions, and batch updates.

(1) Unit insertions. Inserting an edge may result in combining two or more scc's into a single one. This happens if and only if a cycle is formed with the corresponding nodes of these scc's in the contracted graph after the insertion.

Employing the contracted graph G_c, we propose incremental algorithm IncSCC^+, shown in Fig. 2.9, to process unit insertion of edge (v, w). Intuitively, IncSCC^+ checks whether (v, w) inflicts a cycle in G_c by leveraging the topological ranks assigned on nodes in G_c, and combines some of the scc's in $\text{SCC}(G)$ when necessary to get $\text{SCC}(G \oplus \Delta G)$. It separates different types of (v, w), and makes use of topological ranks based on the invariant property mentioned above. Relatively boundedness is guaranteed since every change to the rank of an scc inspected by the algorithm corresponds

---

Algorithm: IncSCC^{+}

Input: A graph G with num$(\cdot)$, lowlink$(\cdot)$, contracted graph G_c,
        SCC(G) and an edge (v, w) to be inserted.

Output: SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$ and G_c.

1. **if** v and w are within the same scc (tree) T **then**
2. $T := T \oplus \Delta G$; update num$(\cdot)$, lowlink$(\cdot)$ for T;
3. **if** $r(\text{scc}(v)) > r(\text{scc}(w))$ **then** update G_c;
4. **if** $r(\text{scc}(v)) < r(\text{scc}(w))$ **then**
5.   aff$_r$:= DFS$_f(G_c, w, r(\text{scc}(v)))$; aff$_l$:= DFS$_b(G_c, v, r(\text{scc}(w)))$;
6.   **if** Tarjan$(\text{aff}_l \cup \text{aff}_r, v)$ has *non-singleton* cycle C **then**
7.   merge the corresponding components of nodes in C;
8.   update num$(\cdot)$, lowlink$(\cdot)$ for the new component;
9.   **else** reallocRank$(\text{aff}_l, \text{aff}_r)$;
10. **return** SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$, and G_c;

Figure 2.9: Algorithm IncSCC^{+}

to a change of lowlink or num, and thus is in AFF.

More specifically, if v and w are within the same scc T, then nothing changes for the other scc's. In this case, IncSCC^{+} only applies ΔG to T to get SCC$(G \oplus \Delta G)$, and computes the changes to num and lowlink, by applying Tarjan on the changed parts (lines 1-2). Otherwise consider the topological ranks of scc(v) and scc(w) in G_c, where scc(v) (resp. scc(w)) refers to the corresponding scc node to which v (resp. w) belongs. Consider the following cases.

(a) If $r(\text{scc}(v)) > r(\text{scc}(w))$, then no new scc is generated, and IncSCC^{+} only updates the graph G_c by inserting edge $(\text{scc}(v), \text{scc}(w))$ or increasing the counter of edges connecting their corresponding scc's (line 3). As the order of topological ranks in G_c is not affected in this case, it concludes that graph G_c is still acyclic and SCC$(G \oplus \Delta G) =$ SCC(G).

(b) If $r(\text{scc}(w)) > r(\text{scc}(v))$, *i.e.,* if the order of these two ranks becomes "incorrect", IncSCC^{+} identifies the *affected area* aff$_l$ and aff$_r$, two subgraphs of G_c induced by nodes whose ranks are no longer valid, through a bi-directional search. It invokes pro-

cedure $\mathsf{DFS_f}$ to conduct a forward DFS traversal from w to find nodes with topological ranks greater than that of v, followed by a backward traversal $\mathsf{DFS_b}$ from v to find nodes having ranks less than that of w (lines 4-5). If a cycle C is formed in the affected area after edge insertion, the corresponding scc's of the nodes in C are merged into one to obtain $\mathsf{SCC}(G \oplus \Delta G)$; this is followed by updating num and lowlink values in the new scc (lines 6-8). Otherwise, although the output is unaffected, it reallocates the topological ranks of nodes in the affected area such that $r(v) > r(v')$ when (v, v') is in G_c, using procedure reallocRank (not shown) (line 9), *i.e.,* the relationship of topological ranks in G_c still holds after reallocation. Procedure reallocRank sorts the previous ranks of those nodes in $\mathsf{aff_l}$ and $\mathsf{aff_r}$, and reassigns them in an ascending order, first $\mathsf{aff_r}$ and then $\mathsf{aff_l}$. Indeed, nodes in $\mathsf{aff_r}$ should have lower ranks than those in $\mathsf{aff_l}$ due to the edge insertion. The order on the ranks within $\mathsf{aff_r}$ and $\mathsf{aff_l}$ is unchanged.

Example 2.7: Continuing with Example 2.6, consider insertion of edge $e_4 = (b_4, b_3)$ into G. Observe that the topological ranks $r(\mathsf{scc}(b_4)) < r(\mathsf{scc}(b_3))$ in G_c; thus IncSCC^+ identifies the affected area that consists of nodes 1 and 2 and forms a cycle. Then scc_1 and scc_2 are merged to get the output. □

Correctness & complexity. The correctness of IncSCC^+ is warranted by the following properties: (a) scc's are merged in response to an edge insertion if and only if they form a cycle in the contracted graph; and (b) the topological ranks of the nodes on any path in G_c decrease monotonically.

IncSCC^+ is in $O(|\mathsf{AFF}| \log |\mathsf{AFF}|)$ time. The cost for updating lowlink and num by Tarjan on the affected parts of each scc is $O(|\mathsf{AFF}|)$. Besides this, it only visits those nodes in the contracted graph with updated ranks, and their neighbors (line 5). The number of nodes visited does not exceed $|\mathsf{AFF}|$ since there must be changes to num and lowlink values in the scc's that they refer to. Cycle detection (line 6) is done in $O(|\mathsf{AFF}|)$ time and rank reallocation (line 9) takes $O(|\mathsf{AFF}| \log |\mathsf{AFF}|)$ time via sorting by using heaps. Hence IncSCC^+ is bounded relative to Tarjan.

(2) Unit deletions. When edge (v, w) is deleted from G, an scc may be split into multiple ones. However, the output is unchanged if v still reaches w after the deletion. We give an incremental algorithm for SCC under unit deletions, denoted by IncSCC^- (Fig. 2.10). Intuitively, it examines the reachability from v to w by using num and lowlink maintained, and computes new scc's in $\mathsf{SCC}(G \oplus \Delta G)$ when v no longer reaches w within the same scc. The reachability checking is done as a byproduct of change

---

Algorithm: IncSCC$^-$

Input: A graph G with num$(\cdot)$, lowlink$(\cdot)$, contracted graph G_c,
        SCC(G) and an edge (v, w) to be deleted.
Output: SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$ and G_c.

1.   **if** scc$(v) \neq$ scc(w) **then** update G_c if needed;
2.   **if** (v, w) is a *reverse frond* within a tree T **then** $T := T \oplus \Delta G$;
3.   **if** (v, w) is a *frond* or *cross-link* within a tree T **then**
4.   **if** chkReach$(T, \Delta G, v)$ **then** $T := T \oplus \Delta G$;
5.   **else** replace T with Tarjan$(T \oplus \Delta G, T.\text{root})$; update G_c;
6.   **if** (v, w) is a *tree arc* within a tree T **then**
7.   **if** chkReach$(T, \Delta G, v)$ **and** w is discovered in
          updateDFS(e, T) for a *selected* edge e **then** $T := T \oplus \Delta G$;
8.   **else** replace T with Tarjan$(T \oplus \Delta G, T.\text{root})$; update G_c;
9.   **return** SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$, and G_c;

Figure 2.10: Algorithm IncSCC$^-$

propagation to num and lowlink, from which relatively boundedness is obtained.

As shown in Figure 2.10, to delete an edge (v, w), algorithm IncSCC$^-$ first checks whether the two endpoints belong to different components, and updates the contracted graph G_c if needed, *i.e.,* deletion of an edge or decrement the counter in G_c (line 1). Then it distinguishes the following cases based on the type of (v, w) within an scc (tree) T to check whether v still reaches w after edge deletion.

(a) Reverse frond (line 2). In this case, IncSCC$^-$ just deletes the edge from T and SCC$(G \oplus \Delta G)$ is obtained immediately without further operations.

(b) Frond or cross-link (lines 3-5). Procedure chkReach (see below) is invoked by IncSCC$^-$ to decide whether v still reaches the root r of T after edge deletion, from which it determines the reachability from v to w, and updates T if so, or replaces T with newly computed scc's in SCC$(G \oplus \Delta G)$.

(c) Tree arc (lines 6-8). Algorithm IncSCC$^-$ also checks the reachability from v to the root of T. Note that node v could no longer reach w even if the answer is positive. Hence it conducts a DFS traversal (procedure updateDFS, not shown) starting from a

selected edge (v',w) if it exists, or from (v',w'), where w' is a descendant of w while v' is not in T, and v' has the largest num of such edges; it updates the num and lowlink values in $T \oplus \Delta G$ as in Tarjan (lines 6-7). If w is encountered in this process, then root r still reaches w and so does v; in this case only tree T is updated (line 7). Otherwise new scc's in $\mathsf{SCC}(G \oplus \Delta G)$ are generated by using Tarjan on the affected scc (line 8).

Procedure chkReach. We next show how to check the reachability from node v to the root of tree T after deleting an edge (v,w) adjacent to it. The checking is done as a byproduct of the propagation of changes to lowlink values of v's ancestors in T. More specifically, when $\mathsf{lowlink}(v)$ equals $\mathsf{num}(w)$ (frond or cross-link) or $\mathsf{lowlink}(w)$ (tree arc), procedure chkReach computes the new value of $\mathsf{lowlink}(v)$ based on its successors other than w, and this change is propagated to its ancestors using a strategy similar to that adopted by IncKWS^- (Section 2.3.2). Once it encounters a node v' with updated $\mathsf{lowlink}(v')$ such that $\mathsf{num}(v') = \mathsf{lowlink}(v')$, it concludes that v no longer reaches the root in $T \oplus \Delta G$.

Example 2.8: Consider deleting edge $e_5 = (c_1, a_1)$ from G of Fig. 2.4, which is a frond in scc_3 (see Example 2.6). Since the lowlink value of c_1 increases to 3 and equals its num after deletion, procedure chkReach concludes that c_1 no longer reaches root a_1 of scc_3. In light of this, IncSCC^- computes new scc's on affected scc_3 to update the output, *i.e.,* scc_3 is split into three components. The contracted graph G'_c after the deletion is also shown in Fig. 2.8 (solid edges). $\square$

Correctness & complexity. (1) The correctness of IncSCC^- is warranted by the following. (a) The output remains unaffected if v still reaches w after deleting (v,w). (b) Reverse fronds have no impact on the reachability. (c) If $\mathsf{num}(v') = \mathsf{lowlink}(v')$ while v' is not the root of a tree, then v' no longer reaches the root, which is also the invariant property of algorithm Tarjan. (2) IncSCC^- runs in $O(|\mathsf{AFF}|)$ time, and is thus bounded relative to Tarjan. It only visits nodes that either are in new scc's, or have updated num or lowlink values and their neighbors by procedures chkReach and updateDFS, at most once. All these nodes are covered by AFF. Moreover, each node visited is involved in value comparison for constant times. Updating contracted graph G_c is also done within $O(|\mathsf{AFF}|)$ time. Therefore, IncSCC^- is bounded relative to Tarjan under unit edge deletion.

(3) Batch updates. We now present algorithm IncSCC to process $\Delta G = (\Delta G^+, \Delta G^-)$, shown in Fig. 2.11. It handles multiple updates in groups instead of one by one, to

---

Algorithm: IncSCC

Input: A graph G with num$(\cdot)$, lowlink$(\cdot)$, contracted graph G_c,
         SCC(G) and batch updates $(\Delta G^+, \Delta G^-)$.
Output: SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$ and G_c.

1. **for each** scc in SCC(G) **do**
2.   iteratively update num$(\cdot)$, lowlink$(\cdot)$ for each *intra-component*
          update involved in scc;
3.   **if** there is node v no longer reaches w in scc **then**
4.   replace scc with Tarjan$(\text{scc}, \text{scc.root})$; update G_c;
5. update G_c for remaining *inter-component* edge deletions;
6. identify affected area aff on G_c for *inter-component* insertions;
7. **for each** non-singleton cycle C in Tarjan(aff, v) **do**
8.   merge the components corresponding to C;
9.   update num$(\cdot)$, lowlink$(\cdot)$ for the new component;
10. reallocate topological ranks to nodes in aff;
11. **return** SCC$(G \oplus \Delta G)$ and updated num$(\cdot)$, lowlink$(\cdot)$, and G_c;

Figure 2.11: Algorithm IncSCC

reduce redundant cost. IncSCC consists of two steps.

(a) IncSCC first processes *intra-component* updates, where the endpoints of an updated edge are in the same scc. All updates to the same scc are grouped and processed together. It starts with edge insertions, and adjusts values of num and lowlink following IncSCC$^+$. Inserted edges are processed following a descending order determined by the num values of their source nodes. Then, following the same processing order, IncSCC$^-$ is invoked to handle deletions grouped together, to reduce redundant updates to num and lowlink values. After these, Tarjan is called on the affected scc's at most once to generate new scc's in SCC$(G \oplus \Delta G)$.

(b) IncSCC then handles *inter-component* updates, for edge updates in which the endpoints fall in different scc's. After updating G_c with deletions, forward and backward traversals are performed to find the affected areas for all inter-component insertions, similar to IncSCC$^+$. However, IncSCC stores these areas in a global structure aff, and

checks the existence of cycles formed by nodes from this global affected area, instead of processing unit updates one by one. Components are merged, and $\mathsf{num}(\cdot)$ and $\mathsf{lowlink}(\cdot)$ are revised, along the same lines as IncSCC^{+} to get $\mathsf{SCC}(G \oplus \Delta G)$.

Finally, topological ranks are reallocated if needed, and $\mathsf{SCC}(G \oplus \Delta G)$ is returned (see Fig. 2.11).

Example 2.9: Consider batch updates ΔG of Example 2.3. The intra-component deletions of e_2 and e_5 are firstly handled. Since $e_2 = (c_2, b_3)$ is a reverse frond in scc_2, IncSCC just deletes it from scc_2. Deletion of e_5 is processed as described in Example 2.8. Thereafter, the remaining three inter-component insertions in ΔG are handled by retrieving the affected area on contracted graph G_c'. Note that nodes 1 to 5 are covered by affected area aff that constitutes an scc in G_c', hence all the previous scc's in $\mathsf{SCC}(G)$ except scc_4 (d_2) are merged to obtain $\mathsf{SCC}(G \oplus \Delta G)$ in IncSCC. □

Correctness & complexity. The correctness of IncSCC follows from the correctness of IncSCC^{+} and IncSCC^{-}. IncSCC takes $O(|\mathsf{AFF}|(|\Delta G| + \log|\mathsf{AFF}|))$ time. Indeed, processing intra-component updates needs $O(|\Delta G||\mathsf{AFF}|)$ time since each update to the auxiliary structures in AFF is checked at most $|\Delta G|$ times; and handling inter-component updates takes $O(|\Delta G||\mathsf{AFF}| + |\mathsf{AFF}|\log|\mathsf{AFF}|)$ time, where each node with updated ranks in G_c is accessed by at most $|\Delta G|$ different bi-directional searches; the time for final rank reallocation is in $O(|\mathsf{AFF}|\log|\mathsf{AFF}|)$ as all such nodes are collected in aff. Thus IncSCC is bounded relative to Tarjan.

2.5 Experimental Evaluation

Using real-life and synthetic data, we conducted three sets of experiments to evaluate the impacts of (1) the size $|\Delta G|$ of batch updates; (2) the complexity of queries Q for KWS, RPQ and ISO (see below); and (3) the size $|G|$ of graphs on our incremental algorithms, compared with their batch counterparts and some existing dynamic algorithms.

Experimental setting. We used the following datasets.

Graphs. We used two real-life graphs: (a) DBpedia, a knowledge graph [DBp] with 4.3 million nodes, 40.3 million edges and 495 labels; and (b) *LiveJournal* (liveJ in short), a social network [SNA] with 4.9 million nodes, 68.5 million edges and 100 labels. We also designed a generator to produce synthetic graphs G, controlled by the number of

nodes $|V|$ (up to 50 million) and number of edges $|E|$ (up to 100 million), with labels drawn from an alphabet Σ of 100 symbols.

Updates ΔG are randomly generated for real-life and synthetic data, controlled by size $|\Delta G|$ and a ratio ρ of edge insertions to deletions. We use $\rho = 1$ unless stated otherwise, *i.e.,* the size of the data graphs G remain stable.

Query generators. We randomly generated 30 queries of KWS, RPQ and ISO with labels drawn from the graphs. More specifically, (1) KWS queries are controlled by the number m of keywords and bound b; (2) RPQ queries are controlled by the size (recall size $|Q|$ of a regular path query from Section 2.1.1) and the numbers of occurrences of $\cdot$, $+$ and Kleene $*$; and (3) ISO queries are controlled by the number of nodes $|V_Q|$, the number of edges $|E_Q|$ and the diameter d_Q, *i.e.,* the length of longest shortest path between any two nodes in Q when taken as an undirected graph.

Algorithms. We implemented the following algorithms, all in Java. (1) Incremental algorithms (a) IncKWS (Section 2.3.2), IncRPQ (Section 2.4.2), IncSCC (Section 2.4.3) and IncISO (Section 2.3.3); (b) IncKWS$_n$, IncRPQ$_n$, IncSCC$_n$ and IncISO$_n$, which process unit updates in batch ΔG one by one by calling their algorithms for unit updates developed in this work; (c) DynSCC that combines the incremental algorithm in [HKM$^+$12] to process insertions and decremental algorithm in [Lac13] for deletions. (2) Batch algorithms BLINKS [HWYY07] for KWS, RPQ$_{NFA}$ for RPQ, Tarjan for SCC, and VF2 [CFSV04] for ISO.

We did the experiments on an Amazon EC2 r3.4xlarge instance, powered by Intel Xeon processor with 2.3GHz, with 122 GB memory and 320GB SSD storage. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Impact of $|\Delta G|$. We first evaluated the impact of $|\Delta G|$ on the performance of IncKWS, IncRPQ, IncSCC and IncISO, compared with (a) their batch counterparts, and (b) incremental IncKWS$_n$, IncRPQ$_n$, IncSCC$_n$ and IncISO$_n$, and DynSCC for SCC. We conducted the experiments (a) on real-life graphs by varying $|\Delta G|$ from 2.2*M* to 17.6*M* in 2.2M increments over DBpedia and from 3.7*M* to 29.6*M* in 4M increments over liveJ, which account for 5% to 40% of each graph; and (b) synthetic G with $|G| = (50M, 100M)$ by varying $|\Delta G|$ from 7.5*M* to 60*M* in 7.5M increments, *i.e.,* 5% to 40% of $|G|$, for SCC; the results for KWS, RPQ and ISO on synthetic graphs are consistent with their counterparts on real-life graphs, and hence are not reported here.

(a) Varying ΔG, KWS (DBpedia)

(b) Varying ΔG, KWS (liveJ)

(c) Varying ΔG, RPQ (DBpedia)

(d) Varying ΔG, RPQ (liveJ)
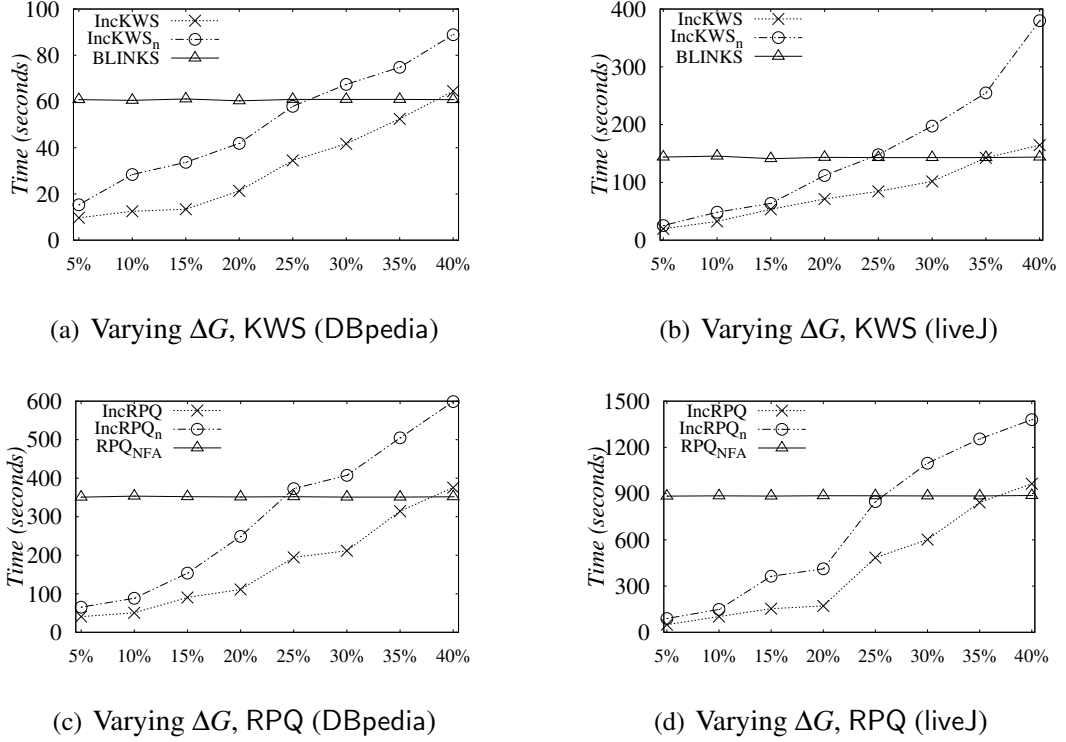
Figure 2.12: Effectiveness of incremental graph computations (KWS and RPQ)

(1) KWS. Fixing $m = 3$ and $b = 2$, we report the performance of IncKWS on DBpedia and liveJ in Figures 2.12(a) and 2.12(b), respectively. We find the following. (a) IncKWS outperforms BLINKS from 6.3 times to 2.8 times over DBpedia, and from 7.3 times to 2 times over liveJ, when $|\Delta G|$ varies from 5% to 20% of $|G|$. In fact, IncKWS does better than BLINKS when $|\Delta G|$ is up to 35% and 30% of $|G|$, respectively. These verify the effectiveness of localizable incremental algorithm IncKWS. (b) IncKWS is from 1.6 to 2 and 1.3 to 1.7 times faster than $IncKWS_n$ in the same setting. This validates the effectiveness of our optimization strategies on batch updates. (c) The larger $|\Delta G|$ is, the slower IncKWS and $IncKWS_n$ are, as expected. However, when $|\Delta G|$ increases, the gap between the performance of IncKWS and $IncKWS_n$ gets larger, which is more evident on liveJ. That is, IncKWS scales better with $|\Delta G|$. In contrast, BLINKS is indifferent to $|\Delta G|$. (d) IncKWS is efficient: it takes 12 and 32 seconds over DBpedia and liveJ, respectively, when $|\Delta G|$ is 10% of $|G|$, as opposed to 61 and 146 seconds by BLINKS. (e) The ratio ρ of insertions to deletions in ΔG has no impact on the performance of IncKWS, by varying ρ while keeping $|\Delta G|$ unchanged (not shown).
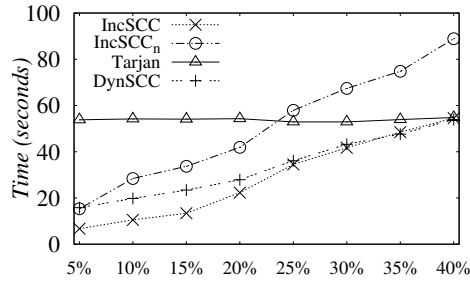
(a) Varying ΔG, SCC (DBpedia)
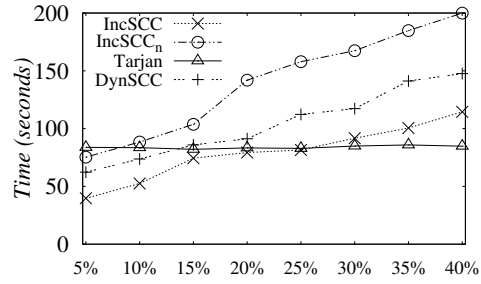
(b) Varying ΔG, SCC (liveJ)
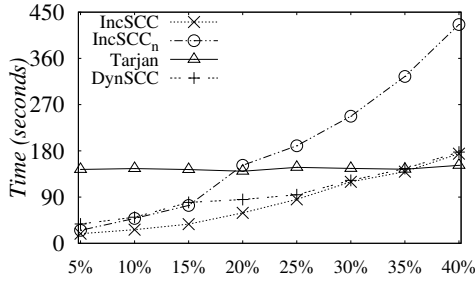
(c) Varying ΔG, SCC (Synthetic)
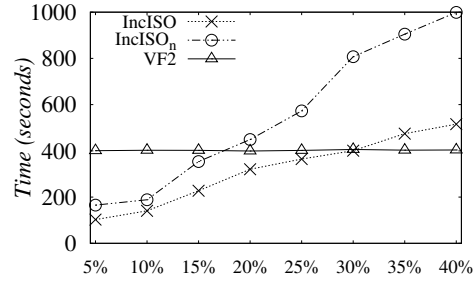
(d) Varying ΔG, ISO (DBpedia)
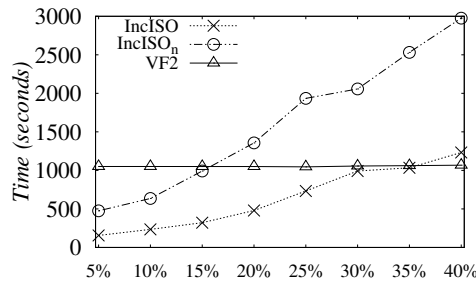
(e) Varying ΔG, ISO (liveJ)

Figure 2.13: Effectiveness of incremental graph computations (SCC and ISO)

(2) RPQ. We then evaluated the relatively bounded algorithm IncRPQ. Fixing $|Q| = 4$, Figures 2.12(c) and 2.12(d) show that (a) IncRPQ is from 8.6 to 3.2 times faster than RPQ_{NFA} on DBpedia, and from 12.7 to 4.1 times faster on liveJ, when $|\Delta G|$ varies from 5% to 20% of $|G|$. (b) IncRPQ consistently does better than $IncRPQ_n$. The improvement is on average 2.3 times when $|\Delta G|$ is about 15% of $|G|$. (c) IncRPQ scales better with $|\Delta G|$ than $IncRPQ_n$, especially when $|\Delta G|$ is large. (d) IncRPQ is insensitive to ρ.

(3) SCC. Figures 2.13(a), 2.13(b) and 2.13(c) report the performance for SCC over

DBpedia, liveJ and synthetic graphs, respectively. We find the following. (a) IncSCC is from 8 to 1.5, 2.3 to 1.2, and 7.7 to 1.7 times faster than Tarjan over DBpedia, liveJ and synthetic graphs, respectively, when $|\Delta G|$ varies from 5% to 25% of $|G|$. These verify the effectiveness of incrementalizing batch algorithm Tarjan. It is from 1.7 to 2.6, 1.9 to 2.1, and 1.4 to 2.2 times faster than $IncSCC_n$ in the same setting. (b) IncSCC performs better than DynSCC. For instance, IncSCC is on average 2.1 times faster than DynSCC when $|\Delta G|$ varies from 5% to 15% of $|G|$ over synthetic graphs. In particular, DynSCC does not do well with small $|\Delta G|$ due to its additional cost for maintaining dynamic data structures even when the output remains stable. (c) IncSCC works better on DBpedia than on liveJ since there are large scc's in liveJ, which take up to 77% of $|G|$, and need to be split in response to ΔG. (d) IncSCC is insensitive to ρ, similar to IncKWS and IncRPQ.

(4) ISO. Fixing $|Q| = (4,6,2)$, *i.e.,* pattern queries with 4 nodes, 6 edges and diameter 2, we evaluated localizable IncISO. As shown in Figures 2.13(d) and 2.13(e) on DBpedia and liveJ, respectively, (a) IncISO behaves better than VF2 and $IncISO_n$ when $|\Delta G|$ is no more than 25% of $|G|$; it is from 5.6 to 1.8 times faster than VF2 and from 2.4 to 2.6 times faster than $IncISO_n$, respectively, for $|\Delta G|$ from 5% to 25% of $|G|$. (b) The gap between the performance of IncISO and $IncISO_n$ gets larger when $|\Delta G|$ grows. (c) IncISO and $IncISO_n$ take longer to process edge insertions than deletions for the same $|\Delta G|$. This is because matches to be removed can be directly identified and hence, IncISO is faster for deletions. We also find that IncISO is insensitive to ρ.

(5) Unit updates. Using the same set of queries, we also evaluated the performance of these algorithms on processing unit updates, which consist of either a unit insertion or a unit edge deletion. As expected, the improvements of incremental algorithms are substantial. More specifically, IncKWS, IncRPQ, IncSCC and IncISO outperform their batch counterparts by 89 times, 221 times, 37 times, and 393 times on average, respectively (not shown). Moreover, IncSCC is 5.7 times faster than DynSCC on average.

Exp-2: Query complexity. We next evaluated the impact of queries Q, by varying different parameters of Q. We focused on KWS, RPQ and ISO, as SCC has a constant query. We fixed $|\Delta G| = 4.4M$, *i.e.,* 10% of $|G|$, and used DBpedia.

(1) KWS. We varied (m,b) from $(2,1)$ to $(6,5)$ for KWS queries. As shown in Figure 2.14(a), (a) the larger (m,b) is, the longer time is taken by all the algorithms, as expected. (b) IncKWS performs well on real-life queries. For queries with 4 keywords and bound 3, it takes 17 seconds over DBpedia, as opposed to 44 seconds by BLINKS.
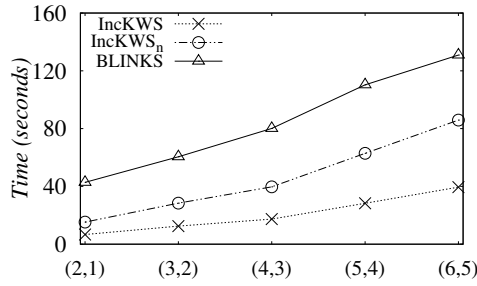
(a) Varying Q, KWS (DBpedia)
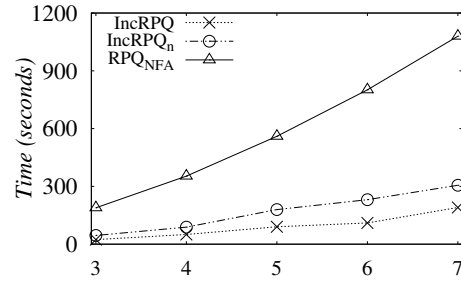
(b) Varying Q, RPQ (DBpedia)
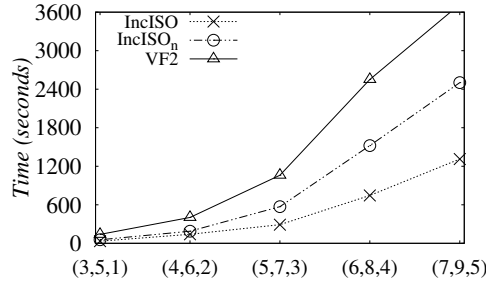
(c) Varying Q, ISO (DBpedia)

Figure 2.14: Efficiency of incremental graph computations

It works better on sparse DBpedia than on liveJ (not shown). (c) IncKWS outperforms the other algorithms, consistent with Fig. 2.12(a).

(2) RPQ. Varying $|Q|$ from 3 to 7, the results in Fig. 2.14(b) tell us the following. (a) IncRPQ is efficient: it returns answers within 190 seconds for all the queries, as opposed to 1080 seconds by RPQ_{NFA} and 326 seconds by $IncRPQ_n$. (b) The occurrences of Kleene $*$ have little impact on all the algorithms, as the size of NFA M_Q only depends on the number of node labels in Q. (c) IncRPQ outperforms RPQ_{NFA} and $IncRPQ_n$ on all the queries; this is consistent with Fig. 2.12(c).

(3) ISO. Varying $|Q| = (|V_Q|, |E_Q|, d_Q)$ from $(3, 5, 1)$ to $(7, 9, 5)$, we evaluated the impact of pattern queries. Figure 2.14(c) shows that all algorithms take longer over larger $|Q|$, as expected. However, (a) IncISO outperforms VF2 and $IncISO_n$ in all the cases, for the same reasons given above. (b) IncISO does well: it takes 290 seconds when $|Q| = (5, 7, 3)$, but VF2 and $IncISO_n$ take 1160 and 570 seconds, respectively.

Exp-3: Impact of $|G|$. We finally evaluated the impact of $|G|$ using synthetic graphs. Fixing $|\Delta G| = 15M$ and using the same set of queries tested in Exp-1, we varied $|G|$ with scale factors from 0.2 to 1. Figures 2.15(a), 2.15(b), 2.15(c) and 2.15(d) report

(a) Varying G, KWS (Synthetic)

(b) Varying G, RPQ (Synthetic)

(c) Varying G, SCC (Synthetic)

(d) Varying G, ISO (Synthetic)
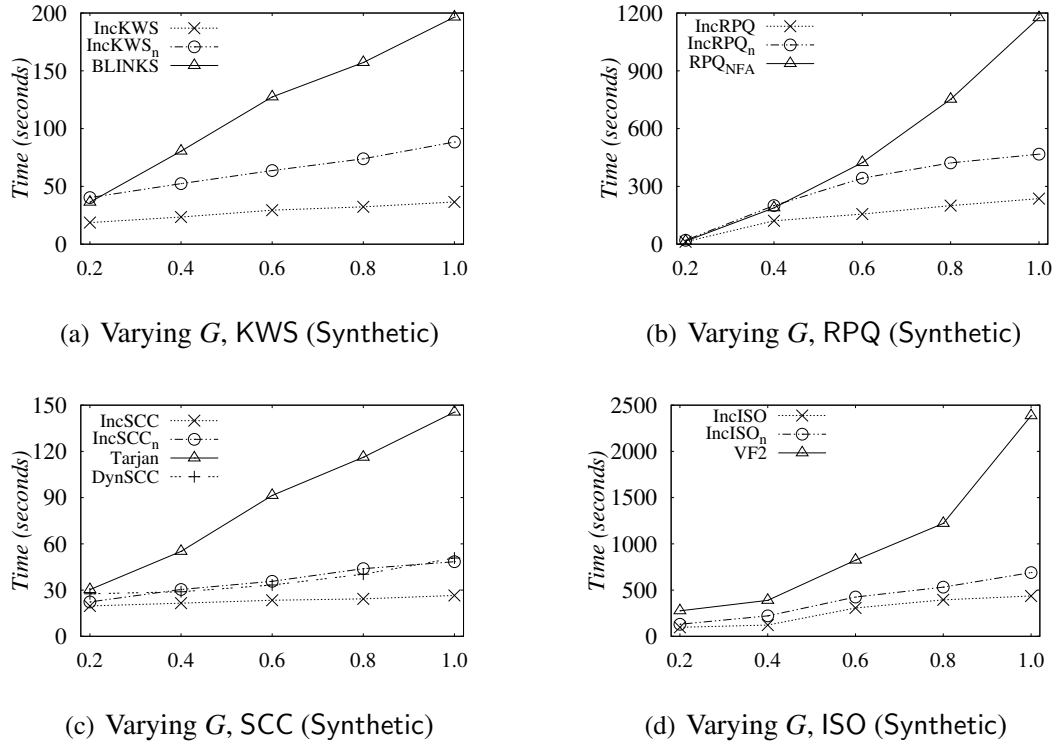
Figure 2.15: Scalability of incremental graph computations

the performance for KWS, RPQ, SCC and ISO, respectively. Observe the following. (a) All the incremental algorithms are less sensitive to $|G|$ compared with their batch counterparts. (b) Incremental algorithms scales well with $|G|$ and are feasible on large graphs.

Summary. From the experiments we find the following. (1) Incremental algorithms, either localizable or relatively bounded, are more effective than their batch counterparts in response to updates. When $|\Delta G|$ varies from 5% to 20% of $|G|$ for the three full-size graphs G, IncKWS, IncRPQ, IncSCC and IncISO outperform BLINKS, RPQ$_{NFA}$, Tarjan and VF2 from 6.9 to 2.4 times, 11.6 to 2.8 times, 3.4 to 1.7 times, and 7.9 to 2 times on average, respectively. They outperform the batch algorithms even when $|\Delta G|$ is up to 30%, 35%, 25% and 25% of $|G|$, respectively. (2) Incremental algorithms scale well with $|G|$ and are feasible on real-life graphs when ΔG is small, as commonly found in practice. For instance, IncKWS, IncRPQ, IncSCC and IncISO take 9, 42, 7 and 113 seconds, respectively, when updates account for 5% of DBpedia, as opposed to 62, 355, 54 and 427 seconds by their batch counterparts. (3) Our optimization strategies for batch updates effectively improve the performance by 1.6 times on average.

Chapter 3

Keys For Graphs

This chapter proposes keys for graphs, a declarative approach to provide the invariant connection between vertices in graphs and the real-world entities they refer to. We start with an example that demonstrates keys in graph structured data.

Example 3.1: We illustrate keys for graphs by using examples taken from various domains in knowledge bases. Note that require exact match in the examples for simplicity; however, we can easily relax the constraints to similarity match.

Music. Consider a knowledge base G_1 consisting of triples (s, p, o), indicating subject, predicate and object, respectively; *e.g.,* $(\text{album}, \text{recorded\_by}, \text{artist})$ says that an album is recorded by an artist. It is modeled as a graph in which s and o are nodes, connected by an edge from s to o labeled p.

One might think that name is a key for album. However, this is not the case. For instance, there are different albums recorded by the Beatles and John Farnham with the same name "Anthology 2" in Freebase. Indeed, the name of an album x uniquely identifies x only among all albums recorded by *the same* artist. Alternatively, an album can be identified by its name and its year of initial release. These yield two keys for albums: *An album can be uniquely identified by*

  Q_1: *its name and its primary recording artist,* or
  Q_2: *its name and its year of initial release.*

For the same reason, an artist may not be identified by its name. Indeed, there are 6 artists or bands named "Everest". Nonetheless, a key for artist can be given by incorporating one of the albums that the artist recorded.

  Q_3: *An artist can be identified by the name, and the album he or she recorded.*

These keys are depicted as *graph patterns* $Q_1(x)$, $Q_2(x)$ and $Q_3(x)$ in Fig. 3.1,
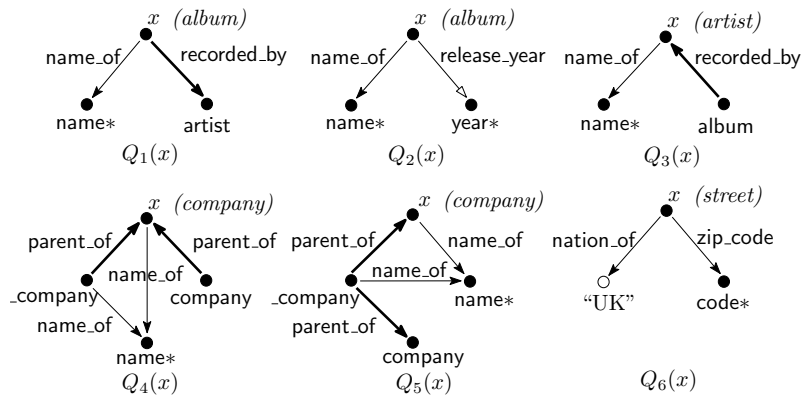
Figure 3.1: Keys for graphs as graph patterns

respectively, where *x* denotes an entity of a particular type to be identified. Intuitively, $Q_1(x)$ says that if two album entities x_1 and x_2 have the same name and are recorded_by the same artist, then x_1 and x_2 must be the same album; similarly for $Q_2(x)$ and $Q_3(x)$.

In contrast to keys for relations and XML, keys for graphs specify *topological constraints* with a graph pattern. Such keys (a) may consider not only *value equality* based on value bindings of properties, *e.g.,* name in Q_1, but also *node identity*, *e.g.,* the identity of artist node in Q_1; and (b) can be *recursively defined*, *e.g.,* to identify an album entity *x*, we may need to identify its primary artist *y*, while to identify an artist entity *y*, we need to identify one of its albums *x*.

Business. As another example, consider the domain for businesses. Typically we can identify a company in the US by its name and head-quarter location. However, there is often business merging and splitting, and very commonly the child company may carry the same name of the parent company without moving the head-quarter (*e.g.,AT&T* and *SBC* merged in 2005 and the new company carried the name of *AT&T*). To distinguish the parent company and the child company in this case, we need to encode the parent-child relationship in the key. That leads to the following two keys to identify companies in a knowledge base G_2, the former for the case of merging and the latter for splitting.

Q_4: *A company merged from a parent company of the same name can be identified by the company name and the other parent company.*

Q_5: *A company split from a parent company of the same name can be identified by the company name and another child company after splitting.*

These keys demonstrate another departure from traditional keys: (a) Q_4 and Q_5 are directed acyclic graphs (DAG), as shown in Fig. 3.1; and (b) they include properties of different entities, *e.g.,* $Q_4(x)$ for company incorporates both the name of the company

and the name of its parent company.

Address. To identify a street in the UK, one can use:

Q_6: *A street in the UK can be identified by its zip code*.

This key does not hold for streets in, *e.g.,* the US. As shown in Fig. 3.1, Q_6 is specified with *a constant* as a condition. This is another departure from conventional keys. ☐

Keys for graphs are a departure from conventional keys. To make practical use of such keys, several questions have to be answered. How should we define keys for graphs, from syntax to semantics? What is the complexity of identifying entities with keys? Is there any scalable algorithm to identify entities with keys in big graphs?

We contend that these keys provide an analogy of traditional keys for graph-structured data. Like relational and XML keys, they specify *the semantics of the data* and remain invariant regardless of changes to the data. They are important to not only traditional use of keys but also several emerging applications. Moreover, entity matching permits parallel scalable algorithms and is feasible in big graphs.

We focus on definition and application of keys in this chapter, and defer the study of key discovery by, *e.g.,* path-identification [LMC11] or communication theory [Guh14], to future work.

3.1 Specifying Keys with Graph Patterns

In this section, we formally define keys for graphs.

3.1.1 Graphs and Graph Pattern Matching

We start with (RDF) graphs, patterns and pattern matching.

(RDF) Graphs. Assume a set $\mathcal{E}$ of entities, a set $\mathcal{D}$ of values, a set $\mathcal{P}$ of predicates (labels), and a set Θ of types. Each entity e in $\mathcal{E}$ has a *unique ID* and a *type* in Θ.

An *(RDF) graph G* is a set of triples $t = (s, p, o)$, where *subject s* is an entity in $\mathcal{E}$, p is a *predicate* in $\mathcal{P}$, and *object o* is either an entity in $\mathcal{E}$ or a value d in $\mathcal{D}$. It can be represented as a directed edge-labeled graph (V, E), also denoted by G, such that (a) V is the set of nodes consisting of s and o for each triple $t = (s, p, o)$; and (b) there is an edge in E from s to o labeled p for each triple $t = (s, p, o)$.

We consider two types of equality:
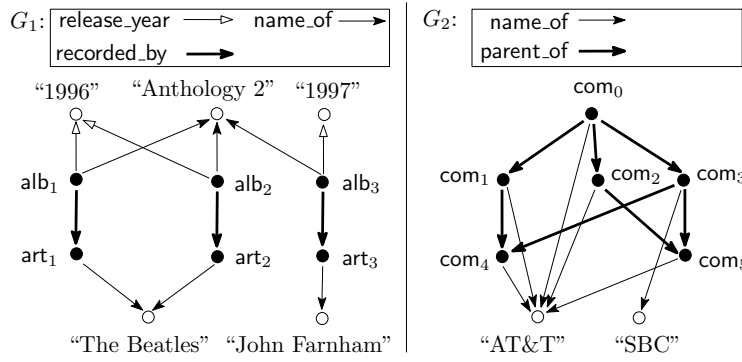
Figure 3.2: Fragments of two knowledge graphs

(a) *node identity* on $\mathcal{E}$: $e_1 \Leftrightarrow e_2$, if entities e_1 and e_2 have the same ID, *i.e.,* they refer to the same entity; and

(b) *value equality* on $\mathcal{D}$: $d_1 = d_2$ if they are the same value.

In G, e_1 and e_2 are represented as the same node if $e_1 \Leftrightarrow e_2$; similarly for values d_1 and d_2 if $d_1 = d_2$.

Example 3.2: Two (RDF) graphs G_1 and G_2 are shown in Fig. 3.2. (1) Graph G_1 represents a fragment of a knowledge base consisting of artists and their albums. For instance, in triple (art$_1$, name_of, "The Beatles"), subject art$_1$ is an entity of type *artist*, and object "The Beatles" is a value; in G_1, both are represented as nodes, and the triple is presented as an edge labeled name_of from art$_1$ to "The Beatles".

(2) Graph G_2 depicts a set of triples for companies. It tells us that, *e.g.,* "AT&T" (com$_4$ of type *company*) has parent companies "AT&T" (com$_1$) and "SBC" (com$_3$).   □

Graph patterns. A *graph pattern* $Q(x)$ is a set of triples (s_Q, p_Q, o_Q), where s_Q is a *variable z*, o_Q is either a value *d* or a variable *z*, and p_Q is a predicate in $\mathcal{P}$. Here *z* has one of three forms: (a) *entity variable y*, to map to an entity, (b) *value variable y∗*, to map to a value, and (c) *wildcard _y*, to map to an entity. Here s_Q can be either *y* or *_y*, while o_Q can be *y*, *y∗* or *_y*. Entity variables and wildcard carry a *type*, denoting the type of entities they represent. In particular, *x* is a designated variable in $Q(x)$, denoting an entity.

As will be seen shortly when we define keys, we enforce node identity ($\Leftrightarrow$) on variables *y*, and value equality ($=$) on *y∗*; for a wildcard *_y*, we just require the existence of an entity with the type of *_y* without checking its node ID or value. Value *d* in $Q(x)$ indicates a *value binding* condition.

A graph pattern can also be represented as a graph such that two variables are

represented as the same node if they have the same name of y, $y*$ or $\_y$; similarly for values d. We assume *w.l.o.g.* that $Q(x)$ is *connected*, *i.e.,* there exists an *undirected* path between x and each node in $Q(x)$.

Example 3.3: Six graph patterns are depicted in Fig. 3.1. For instance, $Q_4(x)$ represents triples $(x, \mathsf{name\_of}, \mathsf{name}*)$, $(\_\mathsf{company}, \mathsf{name\_of}, \mathsf{name}*)$, $(\_\mathsf{company}, \mathsf{parent\_of}, x)$ and $(\mathsf{company}, \mathsf{parent\_of}, x)$. Here x is the designated variable (type *company*), $\mathsf{name}*$ is a value variable, $\mathsf{company}$ is an entity variable and $\_\mathsf{company}$ is a wildcard for *company*. In Q_6, "UK" is a constant value (*i.e., d*) as a condition. □

A *valuation* of $Q(x)$ *in a set S of triples* is a mapping ν from $Q(x)$ to S that preserves values in $\mathcal{D}$ and predicates in $\mathcal{P}$, and maps variables y and $\_y$ to entities of *the same type*. More specifically, for each triple (s_Q, p_Q, o_Q) in $Q(x)$, there exists (s, p, o) in S, written as $(s_Q, p_Q, o_Q) \mapsto_\nu (s, p, o)$ or simply $(s_Q, p_Q, o_Q) \mapsto (s, p, o)$, where

(a) $\nu(s_Q) = s$, $p = p_Q$, $\nu(o_Q) = o$;

(b) o is an entity if o_Q is a variable y or $\_y$; it is a value if o_Q is $y*$, and $o = d$ if o_Q is a value d; and

(c) entities s and s_Q have the same type; similarly for entities o and o_Q if o_Q is y or $\_y$.

We say that ν is a *bijection* if ν is one-to-one and onto.

Graph pattern matching. Consider a graph G and an entity e in G. We say that G *matches* $Q(x)$ *at e* if there exist a set S of triples in G and a valuation ν of $Q(x)$ in S such that $\nu(x) = e$, and ν is a bijection between $Q(x)$ and S. We refer to S as a *match* of $Q(x)$ in G at e under ν.

Intuitively, ν is an isomorphism from $Q(x)$ to S when $Q(x)$ and S are depicted as graphs. That is, we adopt *subgraph isomorphism* for the semantics of graph pattern matching.

Example 3.4: Consider $Q_4(x)$ of Fig. 3.1, G_2 of Fig. 3.2, and a set S_1 of triples in G_2: $\{(\mathsf{com}_1, \mathsf{name\_of}, \text{"AT\&T"}), (\mathsf{com}_4, \mathsf{name\_of}, \text{"AT\&T"}), (\mathsf{com}_1, \mathsf{parent\_of}, \mathsf{com}_4), (\mathsf{com}_3, \mathsf{parent\_of}, \mathsf{com}_4)\}$. Then S_1 is a match of $Q_4(x)$ in G_2 at com_4, which maps variable x to com_4, $\mathsf{name}*$ to "AT\&T", wildcard $\_\mathsf{company}$ to com_1, and company to com_3. □
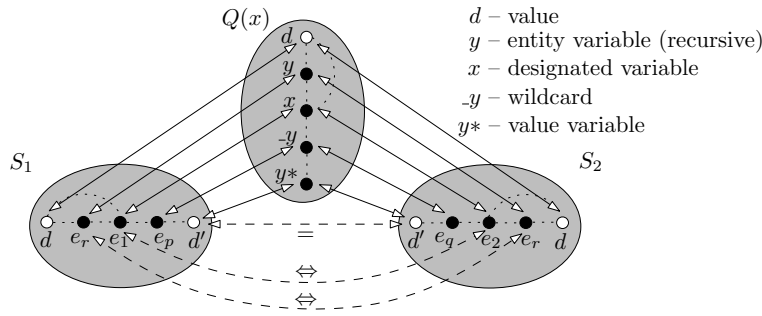
Figure 3.3: The semantics of keys for graphs

3.1.2 Keys for Graphs

Keys. A *key for entities of type* τ is a graph pattern $Q(x)$, where x is a designated entity variable of type τ.

Intuitively, it says that in a graph G, for entities e of type τ, the conditions specified in $Q(x)$ uniquely identify e. For example, Q_1 and Q_2 of Example 3.1 are keys for *album*, Q_3 is a key for *artist*, and Q_4 and Q_5 are keys for *company*.

To give the semantics of keys, we use the following notion. Consider matches S_1 and S_2 of $Q(x)$ at e_1 and e_2 in G under ν_1 and ν_2, respectively. We say that S_1 *coincides with* S_2, denoted by $S_1(e_1) \cong_Q S_2(e_2)$, if a bijection μ between S_1 and S_2 can be derived from ν_1 and ν_2, preserving node identity and value equality. That is, for each (s_Q, p_Q, o_Q) in $Q(x)$ such that $(s_Q, p_Q, o_Q) \mapsto_{\nu_1} (s_1, p_1, o_1)$ and $(s_Q, p_Q, o_Q) \mapsto_{\nu_2} (s_2, p_2, o_2)$, it satisfies conditions:

   (a) if s_Q is a variable y that is distinct from x, then $s_1 \Leftrightarrow s_2$; similarly for o_Q; and

   (b) if o_Q is a variable $y*$, then $o_1 = o_2$.

When s_Q is a wildcard $\_y$, we *do not* require that $s_1 \Leftrightarrow s_2$, *i.e.*, s_1 and s_2 may be distinct entities; similarly for o_Q.

We say that G *satisfies* key $Q(x)$, denoted by $G \models Q(x)$, if *for all entities* e_1 and e_2 in G, if *there exist* matches S_1 and S_2 of $Q(x)$ such that $S_1(e_1) \cong_Q S_2(e_2)$, then $e_1 \Leftrightarrow e_2$.

Based on graph patterns, we next define keys for graphs.

As shown in Fig. 3.3, the key says that if there exist S_1 and S_2 verifying that e_1 and e_2 satisfy the conditions of $Q(x)$, respectively, and if $S_1(e_1) \cong_Q S_2(e_2)$, then e_1 and e_2 must have the same ID, *i.e.,* they are the same entity.

Example 3.5: Continuing with Example 3.4, one can see that $G_2 \not\models Q_4(x)$. Consider S_1 of Example 3.4, and a match S_2 of $Q_4(x)$ at com_5: $\{(\text{com}_2, \text{name\_of, "AT\&T")},$ $(\text{com}_5, \text{name\_of, "AT\&T")}, (\text{com}_2, \text{parent\_of}, \text{com}_5), (\text{com}_3, \text{parent\_of}, \text{com}_5)\}$. Then

Symbols	Notations
$\mathcal{E}, \mathcal{P}, \mathcal{D}$	entities, predicates and data values, respectively
$G, Q(x)$	(RDF) graph and graph pattern, respectively
$e_1 \Leftrightarrow e_2$	node identity
$d_1 = d_2$	value equality
$y, y*, \_y$	variables for entities, values and wildcards, resp.
$\mapsto_v, \mapsto$	mapping from (s_Q, p_Q, o_Q) to (s, p, o)
$S_1(e_1) \cong_Q S_2(e_2)$	match S_1 at e_1 coincides with S_2 at e_2
$G \models Q(x)$	G satisfies key $Q(x)$
$(G, \Sigma) \models (e_1, e_2)$	entities e_1 and e_2 are *identified* by keys in Σ
$|G|, |Q(x)|$	the size of graph G and $Q(x)$
$d(Q, x)$	the radius of $Q(x)$

Table 3.1: Notations in Chapter 3

$S_1(\mathsf{com}_4) \cong_{Q_4} S_2(\mathsf{com}_5)$ but com_4 and com_5 are distinct entities in G_2. Thus either com_4 or com_5 is a duplicate.

Similarly in G_1, either alb_1 or alb_2 is a duplicate (violation of Q_2), and either art_1 or art_2 is a duplicate (by Q_3). However, these are not very obvious since keys for *album* and *artist* are defined by mutual recursion. □

We say that a key $Q(x)$ is *recursively-defined* if it contains some variables y other than x, and is *value-based* otherwise. Intuitively, when $Q(x)$ is recursive, $e_1 \Leftrightarrow e_2$ depends on whether $e \Leftrightarrow e'$ for some other entities e and e' can be identified by variable y, which involves node identity that is determined by using (possibly other) keys. In contrast, when $Q(x)$ is value-based, it decides whether $e_1 \Leftrightarrow e_2$ simply based on value equality on relevant triples in S_1 and S_2.

Example 3.6: Keys Q_1, Q_3, Q_4 and Q_5 depicted in Fig. 3.1 are all recursive, while Q_2 and Q_6 are value-based. □

Remark. (1) For simplicity, we focus on keys defined in terms of value equality and node identity. Nonetheless, the results of this chapter remain intact when *similarity predicates* are used along the same lines as value equality. (2) Relational keys [AHV95] and XML keys [BDF$^+$01] can be readily expressed as value-based keys with patterns of a form of trees.

We will also use the following notations: (1) $|G|$ (resp. $|Q|$) denotes the number of triples in G (resp. $Q(x)$); (2) for a set Σ of keys, $|\Sigma| = \sum_{Q(x) \in \Sigma} |Q|$ and $\|\Sigma\|$ is its cardinality; and (3) the *radius* of $Q(x)$, denoted by $d(Q,x)$, is the longest distance between x and any node in $Q(x)$ when $Q(x)$ is treated as an *undirected* graph, ignoring the edge direction.

The notations of this chapter are summarized in Table 3.1.

3.2   The Entity Matching Problem

In the rest of the chapter we focus on *entity matching*, an important application of keys. We formalize the problem (Section 3.2.1) and establish its complexity (Section 3.2.2). Moreover, we show that in the presence of recursively defined keys, entity matching is hard to be parallelized (Section 3.2.3).

3.2.1   Entity Matching with Keys

Example 3.5 shows that $G_2 \not\models Q_4(x)$, since $S_1(\mathsf{com}_4) \cong_{Q_4} S_2(\mathsf{com}_5)$ but com_4 and com_5 are distinct. However, key $Q_4(x)$ tells us that com_4 and com_5 refer to the same entity and should be *identified*. To formalize this, we revise the classical chase [AHV95] by using keys as rules for entities in graphs.

Chase revisited. Consider a set Σ of keys and a graph G. We use Eq to denote *the equivalence relation* of a set of pairs (e, e') of entities in G of the same type that have been identified by keys in Σ, *i.e.,* Eq is reflexive, symmetric and transitive. We denote by Eq_0 the node identity relation $\Leftrightarrow$, *i.e.,* the set of pairs (e, e) for all entities e in G.

Consider a key $Q(x) \in \Sigma$ and matches S_1 and S_2 of $Q(x)$ at e_1 and e_2 in G under valuations v_1 and v_2, respectively. We define $S_1(e_1) \cong_Q^{\mathsf{Eq}} S_2(e_2)$ by using Eq instead of relation $\Leftrightarrow$ in the definition of $S_1(e_1) \cong_Q S_2(e_2)$. More specifically, for each triple (s_Q, p_Q, o_Q) in Q, if $(s_Q, p_Q, o_Q) \mapsto_{v_1} (s_1, p_1, o_1)$ and $(s_Q, p_Q, o_Q) \mapsto_{v_2} (s_2, p_2, o_2)$, then

(a) if s_Q is a variable y distinct from x, then $(s_1, s_2) \in \mathsf{Eq}$ (instead of $s_1 \Leftrightarrow s_2$); similarly for o_Q; and

(b) if o_Q is a variable $y*$, then $o_1 = o_2$.

We define *a chase step* of G by Σ at Eq as

$$\mathsf{Eq} \to_{(e_1, e_2)} \mathsf{Eq}',$$

where (e_1, e_2) is a pair of entities in G such that (a) $(e_1, e_2) \notin \mathsf{Eq}$, (b) there exist a

key $Q(x)$ in Σ and matches S_1 and S_2 of $Q(x)$ at e_1 and e_2, respectively, such that $S_1(e_1) \cong_Q^{\mathsf{Eq}} S_2(e_2)$; and (c) Eq' is the equivalence relation of $\mathsf{Eq} \cup \{(e_1, e_2)\}$.

Intuitively, when e_1 and e_2 are identified by using a key in Σ, Eq is expanded to Eq' by including (e_1, e_2). For instance, in G_1, $\mathsf{Eq}_0 \rightarrow_{(\mathsf{alb}_1, \mathsf{alb}_2)} \mathsf{Eq}_1$, where Eq_1 is the extension of node identity relation $\Leftrightarrow$ in G_1 by including $(\mathsf{alb}_1, \mathsf{alb}_2)$.

A *chasing sequence* of G by Σ is a sequence

$$\mathsf{Eq}_0, \mathsf{Eq}_1, \ldots, \mathsf{Eq}_k,$$

such that for all $i \in [0, k-1]$, there exists a pair (e_1, e_2) of entities in G, where $\mathsf{Eq}_i \rightarrow_{(e_1, e_2)} \mathsf{Eq}_{i+1}$. The sequence is *terminal* if no chase step by Σ is defined at Eq_k. We refer to Eq_k as the *result* of the chasing sequence.

Chasing of keys has the *Church-Rosser property*:

Proposition 3.1: *For any set Σ of keys and graph G, all terminal chasing sequences of G by Σ are finite and have the same result, regardless of how the keys are applied.* ☐

Proof: Consider a set Σ of keys and a graph G. Let $\mathcal{E}$ be the set of entities in G. Then any terminal chasing sequence of G by Σ is no longer than $|\mathcal{E}|^2$, by the definition of chasing sequences. Hence the sequence is finite.

Assume by contradiction that there exist two terminal chasing sequences $S = (\mathsf{Eq}_0, \ldots, \mathsf{Eq}_k)$ and $S' = (\mathsf{Eq}'_0, \ldots, \mathsf{Eq}'_l)$ of G by Σ that have different results. Assume *w.l.o.g.* that (e_1, e_2) is a pair of entities that is in Eq_k but not in Eq'_l, and let (e_1, e_2) first appear in Eq_{i+1}, *i.e.*, $\mathsf{Eq}_i \rightarrow_{(e_1, e_2)} \mathsf{Eq}_{i+1}$ by applying a key $Q(x)$ in Σ. Then $\mathsf{Eq}_i \subseteq \mathsf{Eq}'_l$ by the assumption. As a result, Eq'_l can be further expanded by applying $Q(x)$ to identify e_1 and e_2 by using the pairs in $\mathsf{Eq}_i \subseteq \mathsf{Eq}'_l$ that have been already identified, by the definition of chasing steps. This contradicts to the assumption that S_2 is terminal. ☐

We denote by $\mathsf{chase}(G, \Sigma)$ the result of a terminal chasing sequence of G by Σ. By Proposition 3.1, this notion is well-defined. We say that entities e_1 and e_2 in G are *identified* by Σ, denoted by $(G, \Sigma) \models (e_1, e_2)$, if $(e_1, e_2) \in \mathsf{chase}(G, \Sigma)$.

Example 3.7: Let $\Sigma_1 = \{Q_1(x), Q_2(x), Q_3(x)\}$ from Fig. 3.1, and $\Sigma_2 = \{Q_4(x), Q_5(x)\}$. Then in G_1 of Fig. 3.2, $(G_1, \Sigma_1) \models (\mathsf{alb}_1, \mathsf{alb}_2)$ by applying $Q_2(x)$, since alb_1 and alb_2 have the same name "Anthology 2" and were initially released in "1996". This is followed by $(G_1, \Sigma_1) \models (\mathsf{art}_1, \mathsf{art}_2)$ by applying $Q_3(x)$ to entities $\{\mathsf{art}_1, \mathsf{alb}_1\}$ and $\{\mathsf{art}_2, \mathsf{alb}_2\}$. Note that art_1 and art_2 are identified *after* we identify alb_1 and alb_2. This is because in contrast to $Q_2(x)$, $Q_3(x)$ is recursively defined: it has an entity variable album. That is, recursively defined keys impose dependency on entities.

In graph G_2 of Figure 3.2, from the discussion above it follows that $(G_2, \Sigma_2) \models$ (com$_4$, com$_5$) by $Q_4(x)$. Similarly, $(G_2, \Sigma_2) \models$ (com$_1$, com$_2$) by applying $Q_5(x)$ to {com$_1$, com$_0$, com$_3$} and {com$_2$, com$_0$, com$_3$}. Note that nodes com$_4$ and com$_5$ can be identified before we identify nodes com$_1$ and com$_2$, since the *wildcard* _company in $Q_4(x)$ does not require com$_1 \Leftrightarrow$ com$_2$. This is why we separate entity variable y from wildcard $\_y$. □

Problem. The *entity matching problem* is stated as follows.

- ○ *Input*: A set Σ of keys, and a graph G.
- ○ *Output*: chase(G, Σ).

3.2.2   The Complexity of Entity Matching

Given a set of keys on a relation R, it is in PTIME to find all pairs of tuples in R that are identified by the keys. In contrast, the entity matching problem is nontrivial. To see this, consider its decision problem, also referred to as entity matching, which is to determine, given Σ, G and a pair (e_1, e_2) of entities in G, whether $(G, \Sigma) \models (e_1, e_2)$.

Theorem 3.2: *Entity matching is* NP-*complete.*   □

Proof: The lower bound follows from Lemma 3.3 (to be shown later).

To prove the membership in NP, we present an NP-algorithm for the entity matching problem. In order to do it, we use the following notions. Given a graph G, a set Σ of keys and two entities e_1 and e_2, if $(G, \Sigma) \models (e_1, e_2)$, then there exists a key $Q(x)$ such that (e_1, e_2) is included in Eq by applying $Q(x)$. As a result, there must exist a *proof tree* $T_{(G, Q(x), e_1, e_2)}$ accordingly as follows:

- (1) each node u in $T_{(G, Q(x), e_1, e_2)}$ is labeled with $(e_i, e_j, Q_k(x), S_i, S_j, \nu_i, \nu_j)$, which encodes that S_i (resp. S_j) is a match of $Q_k(x)$ at e_i (resp. e_j) under valuation ν_i (resp. ν_j);
- (2) the root r of $T_{(G, Q(x), e_1, e_2)}$ is labeled with $(e_1, e_2, Q(x), S_1, S_2, \nu_1, \nu_2)$;
- (3) for each node u of $T_{(G, Q(x), e_1, e_2)}$ that is labeled with $(e_i, e_j, Q_k(x), S_i, S_j, \nu_i, \nu_j)$ and $e_i \not\Leftrightarrow e_j$, there are q edges $(u, u_1), \ldots, (u, u_q)$, where each u_l is labeled with $(e_i^l, e_j^l, Q_k^l(x), S_i^l, S_j^l, \nu_i^l, \nu_j^l)$, if and only if there are q entity variables $y_1, \ldots, y_q$ in $Q(x)$ other than x, and $\nu_i(y_l) = e_i^l$ and $\nu_j(y_l) = e_j^l$; that is, whether $(e_i, e_j) \in$ Eq depends on whether $(e_i^l, e_j^l) \in$ Eq for all $l \in [1, q]$;
- (4) for each leaf node v labeled with $(e_m, e_n, Q_w(x), S_m, S_n, \nu_m, \nu_n)$, either (a) $Q_w(x)$

is a value-based key, or (b) $e_m \Leftrightarrow e_n$.

Intuitively, $T_{(G,Q(x),e_1,e_2)}$ encodes a set E_{e_1,e_2} of pairs of entities in G such that $(e_1,e_2) \in$ chase(G,Σ) if each pair of entities in E_{e_1,e_2} can also be identified by Σ. Moreover, for each node u of $T_{(G,Q(x),e_1,e_2)}$ that is labeled with $(e_i, e_j, Q_k(x), S_i, S_j, v_i, v_j)$, the sub-tree $T^u_{(G,Q(x),e_1,e_2)}$ of $T_{(G,Q(x),e_1,e_2)}$ taking u as root corresponds to the set E_{e_i,e_j}.

Note that there may be more than one nodes labeled with the same pair of two entities (e,e') in G. In this case, when deciding whether $(G,\Sigma) \models (e_1,e_2)$, e and e' are checked multiple times whether $(e,e') \in$ Eq. That means in $T_{(G,Q(x),e_1,e_2)}$, there are redundant sub-trees with roots (e,e') and can be deleted. We next introduce a notion of *proof graph PG^T* that is constructed from $T_{(G,Q(x),e_1,e_2)}$, where PG^T encodes the set E_{e_1,e_2} given above and there do not exist two distinct nodes in PG^T that are labeled with the same pair of entities. More specifically, we construct PG^T as follows. For two nodes u and u' in $T_{(G,Q(x),e_1,e_2)}$ that are labeled with $(e_i, e_j, Q_k(x), S_i, S_j, v_i, v_j)$ and $(e_i, e_j, Q_k(x)', S'_i, S'_j, v'_i, v'_j)$, respectively, *i.e.,* u and v are labeled with the same pair of entities (e_i, e_j), considering the following two cases:

(i) if u and u' are connected by a path $u \to u_1 \to \ldots \to u_k \to u'$ in $T_{(G,Q(x),e_1,e_2)}$, where $k \geq 0$, we define a procedure F_c that replace the subgraph $T^u_{(G,Q(x),e_1,e_2)}$ with $T^{u'}_{(G,Q(x),e_1,e_2)}$; and

(ii) if u and u' are not connected and the level of u is less than or equal to the level of u', we define a procedure F_{nc} that add an edge from the parent of u' to u and delete the sub-tree $T^{u'}_{(G,Q(x),e_1,e_2)}$.

We first use procedure F_c to treat all pair of nodes of $T_{(G,Q(x),e_1,e_2)}$ in case (i) until no such pair of nodes exists; and then use F_{nc} to handle all pair of nodes in case (ii), and finally get proof graph PG_T. Obviously, PG_T encodes the set E_{e_1,e_2} given above, and there does not exist two distinct nodes which are labeled with the same pair of entities. Note that PG_T is a directed acyclic graph (DAG).

We next give the NP-algorithm which works as follows.

1. Guess a DAG G_f with no more than N^2 nodes, where N is the number of entities in G, and for each node u, guess a pair of entities e_i, e_j in G, a pattern $Q_k(x)$ in Σ, two subgraphs S_i and S_j that have the same size with $Q_k(x)$, two mappings v_i and v_j from $Q_k(x)$ to S_1 and S_2, respectively.

2. Staring from the nodes without out-edges, for each node u, where $e_i \nLeftrightarrow e_j$, check if S_i and S_j are two matches of $Q_k(x)$ under v_i and v_j at e_i and e_j, respectively, and $S_i(e_i) \cong^{\text{Eq}}_{Q_k} S_j(e_j)$. If so, return "yes"; otherwise reject the guess and go back to step 1.

As discussed above, we only need to consider such DAG G_f with no more than N^2 nodes. Moreover, step 2 is in PTIME. Thus the algorithm is in NP.   □

One might think that non-recursive keys would make our lives easier. Unfortunately, this simple case already embeds the subgraph isomorphism problem, which is NP-complete (cf. [GJ79]) and can be reduced to the simple case.

Lemma 3.3: *The entity matching problem remains* NP-*hard even when* Σ *consists of a single value-based key* $Q(x)$, *and when graph G is a* DAG *(directed acyclic graph).*   □

Proof: We show the NP-hardness by reduction from the subgraph isomorphism problem for directed graph. Given two directed graph $G = (V_G, E_G)$ and $H = (V_H, E_H)$, it is to decide whether G contains a subgraph G' isomorphic to H, *i.e.*, a subset $V_{G'} \subseteq V_G$ and a subset $E_{G'} \subseteq E_G$ such that $|V_{G'}| = |V_H|$, $|E_{G'}| = |E_H|$, and there exists a one-to-one function $f : V_H \to V_{G'}$ satisfying $(u, v) \in E_H$ if and only if $(f(u), f(v)) \in E_{G'}$. It's known that the subgraph isomorphism problem for directed graph is NP-complete, even when G is a acyclic directed graph (DAG) and H is a directed tree (cf. [GJ79]).

Given a DAG G and a tree H, we assume *w.l.o.g.* that there exists no isolated nodes in G or H, *i.e.*, for each node u in G (resp. H), there exists at least one edge (u, v) or (v, u) in G (resp. H). We define a graph G^0, a pair (e_1, e_2) of entities in G^0, a key $Q(x)$, and we show that G contains a subgraph G' isomorphic to H if and only if $(G^0, \{Q(x)\}) \models (e_1, e_2)$. More specifically, G^0 contains all triples (u, p, v) if $(u, v) \in E_G$, as well as those triples (e_1, p, u) and (e_2, p, u) for each u in V_G; $Q(x)$ contains all triples $(\_u, p, \_v)$ if $(u, v) \in E_H$ and $(x, p, \_r)$ where r is the root of H. Note that u, v, e_1 and e_2 are entities of the same type θ, and $\_u$, $\_v$ are wildcards of type θ as well. Obviously, $Q(x)$ is a value-based key (tree).

Assume that G contains a subgraph $G' = (V', E')$ isomorphic to H. Then G' is a tree. Let S_1 (resp. S_2) contain all triples (u, p, v) if $(u, v) \in E_{G'}$, as well as a triple (e_1, p, r) (resp. (e_2, p, r)), where r is the root of G'. Obviously, S_1 and S_2 are matches of $Q(x)$ in G^0 at e_1 and e_2, respectively, and $S_1 \cong_Q S_2$, *i.e.*, $(G^0, \{Q(x)\}) \models (e_1, e_2)$. Conversely, if e_1 and e_2 can be identified by $\{Q(x)\}$, *i.e.*, there must exist matches S_1 and S_2 of $Q(x)$ in G^0 at e_1 and e_2 respectively. Let G' consist of all edges (u, v) if (u, p, v) is in S_1 and $u \neq e_1$. Clearly, G' is a subgraph of G isomorphic to H.   □

3.2.3   Recursion and Parallelization

Recursively defined keys introduce challenges beyond subgraph isomorphism. As a result, it is hard to find an efficient parallel algorithm for entity matching. To see

this, recall that a datalog program has *the polynomial fringe property* (PFP) if all true facts have a proof tree such that the number of its leaves is polynomial in the data size (cf. [ABC$^+$11]). It is known that datalog programs with PFP can be processed in *polylog parallel computation rounds* via recursive doubling, *i.e.,* in $\log^k N$ rounds for a constant k, where N is the size of the input data. We say that a problem has PFP if it has an algorithm with PFP. It is also known that transitive closure (TC), for instance, has PFP. As a result, TC can be computed in logarithmic MapReduce rounds.

Unfortunately, entity matching is harder than TC. Recursively defined keys impose *dependency* on the order of entities to be processed. This leads to chains C of dependent entity pairs such that to identify a pair (e_1, e_2) in C, we have to either wait for pairs preceding (e_1, e_2) in C to be identified, or incur exponentially many possible matches. In contrast, TC can be computed "partially" in PTIME.

Theorem 3.4: *Entity matching (1) has no PFP, and (2) cannot be parallelized in logarithmic rounds, even on trees.* □

Proof: We prove Theorem 3.4(1) by giving a counterexample, where a set Σ_c of keys and a graph G_c are constructed as follows. Note that G_c is a tree.

(1) The set Σ_c consists of two recursive keys $Q_1(x)$ and $Q_2(x)$, and a value-based key $Q_3(x)$. Let $Q_1(x)$ contain three triples (x, p_1, y_1), (x, p_2, y_2) and $(\_y, p_1, x)$, $Q_2(x)$ also contain three triples $(\_y, p_1, y_1)$ and (y_1, p_2, y_2) and $(\_y, p_2, x)$ while $Q_3(x)$ is defined as $\{(x, p_3, \_y)\}$.

(2) Then we build a *tree* G_c with $(4n + 3)$ entities, e, e_0, e_1, ... e_{2n}, e'_0, e'_1 ... e'_{2n}. For each $i \in [1, 2n - 5]$, we add triples (e_i, p_1, e_{i+2}), (e_i, p_2, e_{i+3}), (e'_i, p_1, e'_{i+2}) and (e'_i, p_2, e'_{i+3}) if i is an odd number. The tree G_c also includes triples (e_0, p_1, e_1), (e_0, p_2, e_2), $(e_{2n-3}, p_3, e_{2n-1})$, (e_{2n-2}, p_3, e_{2n}), (e'_0, p_1, e'_1), (e'_0, p_2, e'_2), $(e'_{2n-3}, p_3, e'_{2n-1})$, $(e'_{2n-2}, p_3, e'_{2n})$, (e, p_1, e_0) and (e, p_1, e'_0). Note that all the entity variables and wildcards in Σ_c and entities in G_c are of the same type.

We depict Σ_c and G_c in Fig. 3.4(a) and 3.4(b), respectively. Note that all the entities in G_c and entity variables in Σ_c are of the same type.

Assume that entity matching has the polynomial fringe property. Then for each entity pair identified by a set of keys, there must exist a *proof tree* (see the proof of Theorem 3.2) such that the number of its leaves is polynomial in the size of the graph. From Σ_c and G_c defined above we can conclude that $(e_0, e'_0) \in \text{chase}(G_c, \Sigma_c)$. It can also be verified that the *proof tree* of $(G_c, \Sigma_c) \models (e_0, e'_0)$ has the form depicted
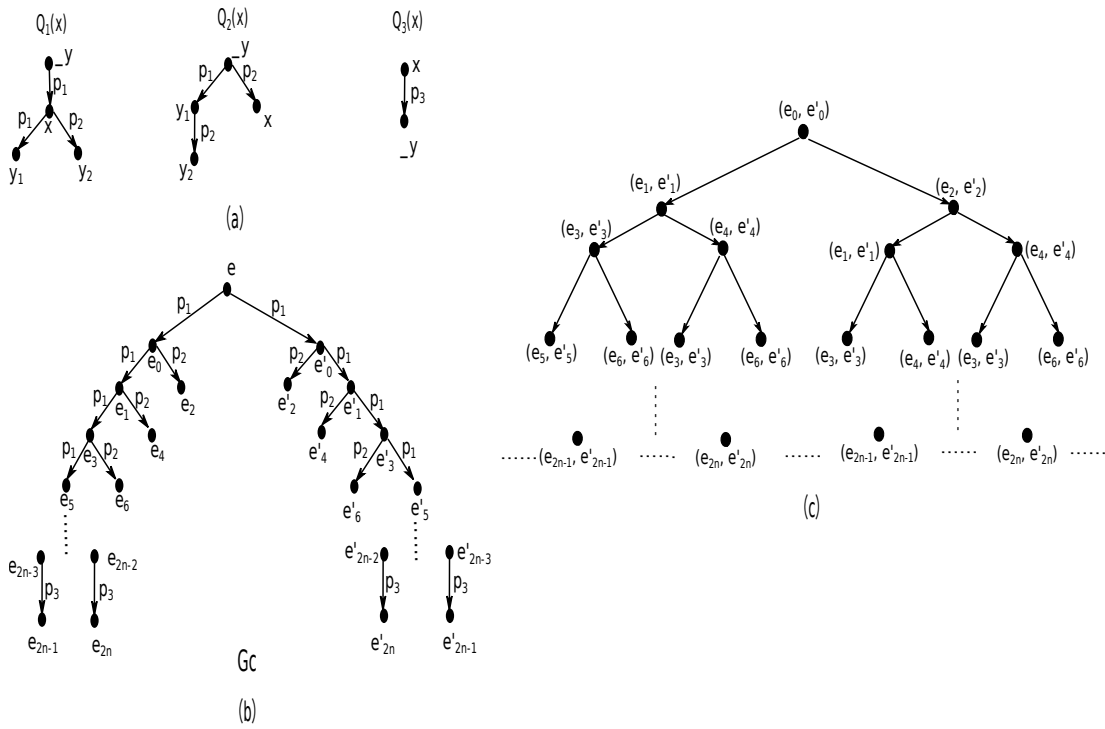
Figure 3.4: Counterexample of PFP on trees

in Fig. 3.4(c) and this tree is the *only* proof tree to identify e_0 and e'_0. To simplify the discussion, each node in the proof tree is denoted by its corresponding entity pair. Intuitively, to identify e_0 and e'_0, all the entity pairs (e_i, e'_i), where $i \in [0, 2n]$, must be checked. Moreover, each recursive key has two entity variables, and only two entity pairs are newly checked when the proof tree grows by a new level. Putting these together, the proof tree is a complete binary tree of depth $(n+1)$, and its size and the number of its leaves are both exponential in n, which contradicts the assumption.

Next we prove Theorem 3.4(2) by reduction from the *Monotone Circuit Value* problem [AP93], in which we are given a Boolean circuit consisting of a directed acyclic graph, whose nodes are INPUT, AND and OR gates. The INPUT gates have zero in-degree while AND and OR gates both of in-degree two. A gate in the circuit is designated to be the OUTPUT gate. The problem is to determine the value of the OUTPUT gate, *i.e.*, **true** or **false**. It is known that this problem is P-complete, even if all gates have out-degree two or less (cf. [AP93]). Given an instance C (we assume *w.l.o.g.* that all the gates in C have out-degree one) of the Boolean circuit, we construct a *tree G* as follows.

- ○ Suppose there are n gates in C, *i.e.*, $g_1, \ldots g_n$. We associate the output value of a gate g_l with a pair (e_l, e'_l) of entities in G such that the output of g_l is true if and
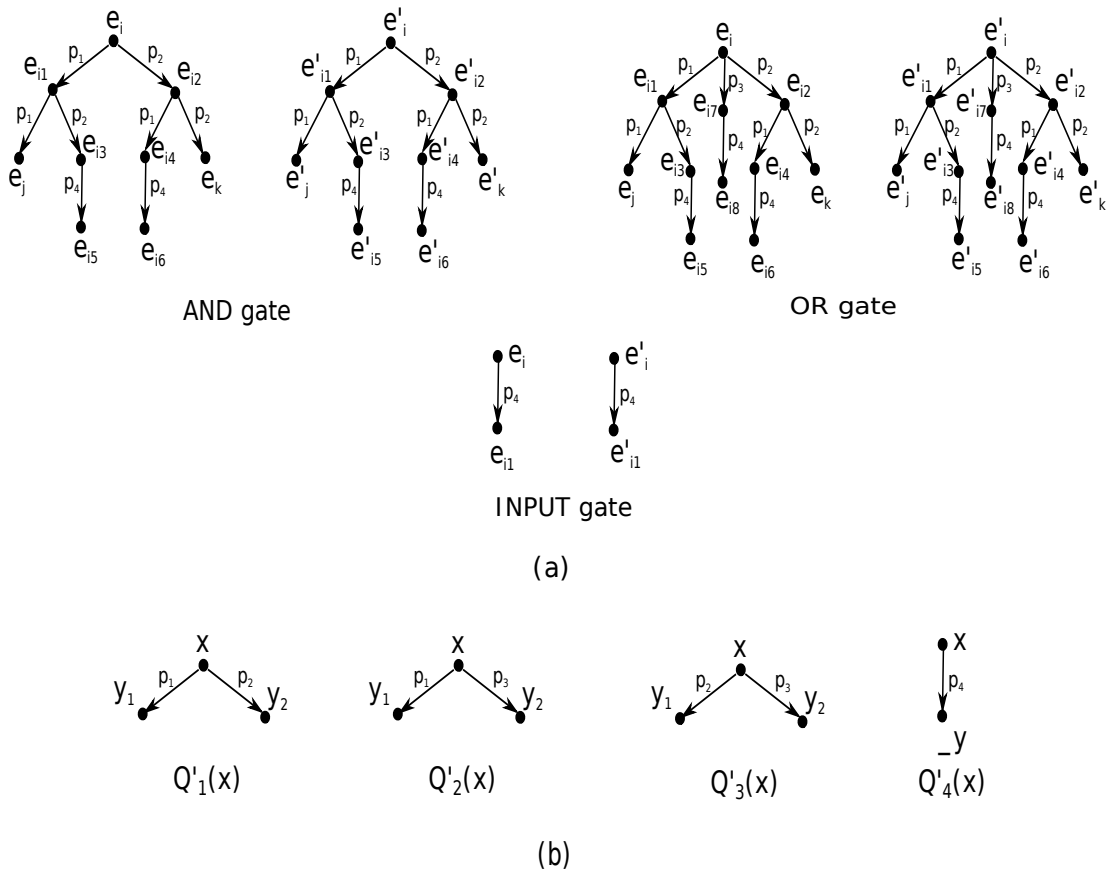
Figure 3.5: Reduction from Monotone Circuit Value

only if (e_l, e'_l) can be identified by a set Σ'_c of keys.

○ Each AND gate is encoded by two sets of triples, each has cardinality 8. For AND gate g_i, we add triples (e_i, p_1, e_{i1}), (e_i, p_2, e_{i2}), (e_{i1}, p_1, e_j), (e_{i1}, p_2, e_{i3}), (e_{i2}, p_2, e_k), (e_{i2}, p_1, e_{i4}), (e_{i3}, p_4, e_{i5}), (e_{i4}, p_4, e_{i6}) and another similar 8 triples except that each e is replaced by e'. Note that (e_j, e'_j) and (e_k, e'_k) corresponds to some other gates g_j and g_k, i.e., the input value of gate g_i.

○ Each OR gate is encoded by two sets of triples, each has cardinality 10. For OR gate g_i, we add triples (e_i, p_1, e_{i1}), (e_i, p_2, e_{i2}), (e_{i1}, p_1, e_j), (e_{i1}, p_2, e_{i3}), (e_{i2}, p_2, e_k), (e_{i2}, p_1, e_{i4}), (e_{i3}, p_4, e_{i5}), (e_{i4}, p_4, e_{i6}), (e_i, p_3, e_{i7}), (e_{i7}, p_4, e_{i8}) and another similar 10 triples except that each e is replaced by e'. Note that (e_j, e'_j) and (e_k, e'_k) corresponds to some other gates g_j and g_k, i.e., the input value of gate g_i.

○ Each INPUT gate with value true is encoded by two sets of triples, each has cardinality 1. For INPUT gate g_i with value true, we add triples (e_i, p_4, e_{i1}) and (e'_i, p_4, e'_{i1}). Note that we don't extend G for INPUT gate of false.

○ The set Σ'_c consists of three recursively defined keys $Q'_1(x)$, $Q'_2(x)$, $Q'_3(x)$ and a value-based key $Q'_4(x)$. Let $Q'_1(x)$ contain two triples (x, p_1, y_1) and (x, p_2, y_2), $Q'_2(x)$ have two triples (x, p_1, y_1) and (x, p_3, y_2), $Q'_3(x)$ contain two triples (x, p_2, y_1) and (x, p_3, y_2), while $Q'_4(x)$ is defined as $\{(x, p_4, \_y)\}$.

The gates and their corresponding triples in G are depicted in Fig. 3.5(a) and Σ'_c is shown in Fig. 3.5(b). Intuitively, G is constructed by connecting multiple sets of triples shown in Fig. 3.5(a) by using their subscripts (*i.e.*, i, j, k), and it is easy to add some dummy nodes to make G a tree since the circuit C is a DAG and each gate has out-degree one. Note that all the entities in G and entity variables and wildcards in Σ'_c are of the same type.

Then one can verify that the output value of gate g_l is true if and only if the entity pair (e_l, e'_l) can be identified by Σ'_c, *i.e.*, $(G, \Sigma'_c) \models (e_i, e'_i)$. Moreover, since G is constructed by connecting multiple copies of the fixed triples in Fig. 3.5(a), we can generate many indexed copies of the fixed triples in Fig. 3.5(a) and identify appropriate structure of G by using C, which involves basic manipulation of indices, and easy to be performed in $O(log|C|)$ space, which means the reduction is a log-space reduction. As monotone circuit value problem cannot be done in logarithmic rounds, neither can entity matching. □

When G is a tree, entity matching is tractable, as opposed to Lemma 3.3. However, it remains hard to be parallelized, as we have shown in Theorem 3.4.

Proposition 3.5: *On trees, entity matching is in* PTIME.   □

Proof: Given a tree G and a set Σ of keys, we propose a PTIME algorithm which works as follows.

1. For each pair (e_1, e_2) of entities of the same type, construct a *product graph* $G_p = (V_p, E_p)$ of G_1^d and G_2^d (d-neighbors of entity e_1 and e_2 respectively, see Section 3.3), where each node in V_p is a pair (s_1, s_2) taken from G_1^d and G_2^d respectively and (1) s_1 and s_2 are entities of the same type or $s_1 = s_2 = d$, (2) $((s_1, s_2), p, (o_1, o_2)) \in E_p$ if and only if $(s_1, p, o_1) \in G_1^d$ and $(s_2, p, o_2) \in G_2^d$. Note that condition (2) ensures G_p is also a tree. We use L to denote the set of all G_p.

2. Eq is used to keep track of entity pairs identified by Σ, initially it stores the set of pairs (e, e) for each entity e in G.

3. For each tree $Q(x) \in \Sigma$ and $G_p \in L$, compute subgraph isomorphism on $Q(x)$ and G_p, if there exists matches S_1 and S_2 (which are derived from the mapping) such that $S_1(e_1) \cong^{\text{Eq}}_Q S_2(e_2)$ (see Section 3.2), then add (e_1, e_2) to Eq and compute

transitive closure of Eq .

4. repeat step 3 until Eq no longer grows.

It is obvious that step 1 and 2 can be done in $O(|G|^4)$, and the size of L is polynomial in the size of G. Moreover subgraph isomorphism for trees can be solved in PTIME (cf. [GJ79]), and step 2 is conducted at most $||L||$ times since at least one pair is identified in each round. Putting these together, the algorithm is in PTIME.   □

Parallel scalability. Not all is lost. Despite Theorems 3.2 and 3.4, we will show that there are effective parallel algorithms for entity matching. To assess the effectiveness of parallel algorithms, we introduce a simple notion.

We say that an algorithm $\mathcal{A}$ for entity matching is *parallel scalable* if its worst-case time complexity is $O(t(|G|,|\Sigma|)/p)$, where p is the number of processors used by $\mathcal{A}$, and t is a function in $|G|$ and $|\Sigma|$, the size of the input. We assume *w.l.o.g.* that $p \ll |G|$ as commonly found in practice.

This suffices. For if $\mathcal{A}$ is parallel scalable, then for given G and Σ, the more processors are used (*i.e.,* the larger p is), the less time $\mathcal{A}$ takes. Indeed, $t(,)$ is independent of p. Hence entity matching is feasible in big G by increasing p. Many parallel algorithms do not have provable guarantee for speedup *no matter how many processors* are added.

3.3   A MapReduce Algorithm

We show that entity matching is feasible in big graphs.

Theorem 3.6: *There exist parallel scalable algorithms in MapReduce for entity matching.*   □

As a proof, we present a parallel scalable algorithm in Section 3.3.1, followed by optimization strategies in Section 3.3.2.

3.3.1   Algorithm and Parallel Scalability

The algorithm, referred to as $\mathsf{EM_{MR}}$ and shown in Fig. 3.6, takes as input a graph G and a set Σ of keys. It returns $\mathsf{Chase}(G,\Sigma)$, the set of all pairs (e_1,e_2) if $(G,\Sigma) \models (e_1,e_2)$.

As opposed to subgraph isomorphism algorithms, $\mathsf{EM_{MR}}$ has to compute the transitive closure (TC) of relation Eq, in which each step involves two subgraph isomorphism checks. By Theorem 3.4, this cannot be done in logarithmic rounds.

Driver: Driver$_{MR}$

Input: Graph G and a set Σ of keys.
Output: chase(G, Σ).

1. construct candidate set L and d-neighbor G^d for each e in L;
2. initialize a set Eq $:= \{(e,e) \mid e \in G\}$;
3. **repeat**
4.   call MapEM; ReduceEM;
            /* initially MapEM$((e_1, e_2), (false))$ for each $(e_1, e_2) \in L$ */
5. **until** Eq no longer changes;
6. **return** Eq;

Mapper: MapEM

Input: A key/value pair $((e_1, e_2), (\text{Flag}))$.
Output: Intermediate key/value pairs.

1. **if** Flag $= true$ or $(G_1^d \cup G_2^d, \text{Eq}, \Sigma) \models (e_1, e_2)$ **then**
2.   emit $((e_1), (e_1, e_2, true))$; emit $((e_2), (e_1, e_2, true))$;
3. **else** emit $((e_1), (e_1, e_2, false))$;

Reducer: ReduceEM

Input: A list of key/value pairs $((e), ([v_1, v_2, \cdots]))$.
Output: Key/value pairs $((e_1, e_2), (\text{Flag}))$.

1. initialize Eq(e) and $L(e)$ with $\emptyset$;
2. **for each** v_i in $[v_1, v_2, \cdots]$ **do**
3.   **if** $v_i = (e_1, e_2, true)$ **then** Eq$(e) :=$ Eq$(e) \cup \{(e_1, e_2)\}$;
4.   **if** $v_i = (e_1, e_2, false)$ **then** $L(e) := L(e) \cup \{(e_1, e_2)\}$;
5. Eq $:=$ Eq $\cup$ Eq(e);
6. **for each** (e_1, e_2) by joining pairs in Eq(e) and Eq, $(e_1, e_2) \notin$ Eq **do**
7.   emit $((e_1, e_2), (true))$; Eq $:=$ Eq $\cup \{(e_1, e_2)\}$;
8. **for each** $(e_1, e_2) \in L(e)$ and $(e_1, e_2) \notin$ Eq **do**
9.   emit $((e_1, e_2), (false))$;

Figure 3.6: Algorithm EM$_{MR}$

Nonetheless, $\mathsf{EM_{MR}}$ *combines* isomorphism checking and TC computation into the same MapReduce process. It ensures parallel scalability by leveraging the data locality of subgraph isomorphism (see below). Better still, it checks whether $(G, \Sigma) \models (e_1, e_2)$ *without* enumerating all isomorphic mappings, unlike conventional algorithms.

$\mathsf{EM_{MR}}$ starts with a set Eq consisting of (e, e) for all entities e in G, and a set L of *candidates, i.e.,* all entity pairs (e_1, e_2) having the same *type* on which at least one key in Σ is *defined*. We say that a key $Q(x)$ is defined on e if x and e have the same type. For all $(e_1, e_2) \in L$, it checks whether (e_1, e_2) is in Eq, or $(G, \Sigma) \models (e_1, e_2)$, *in parallel*. If so, it adds (e_1, e_2) to Eq, and incrementally extends the TC of Eq. Note that $(G, \Sigma) \models (e_1, e_2)$ *once* (e_1, e_2) can be identified by *one* key in Σ, no matter how many keys are defined on it. The process iterates until Eq no longer grows, *i.e.,* $\mathsf{chase}(G, \Sigma) = \mathsf{Eq}$. It takes at most $|\mathsf{Eq}|$ rounds of iterations.

To reduce the cost of checking whether $(G, \Sigma) \models (e_1, e_2)$, $\mathsf{EM_{MR}}$ capitalizes on the following notions.

(1) The d-neighbor G^d of entity e. Let d be the maximum radius of those keys $Q(x)$ in Σ that are defined on e, and V_d be the set of nodes in G that are within d-hops of e. The *d-neighbor* of e is the subgraph of G induced by V_d, consisting of nodes in V_d and edges of G connecting them.

To check $(G, \Sigma) \models (e_1, e_2)$, $\mathsf{EM_{MR}}$ inspects the *d*-neighbors (G_1^d, G_2^d) of (e_1, e_2), *not* the entire G. Indeed, one can verify the *data locality*: $(G, \Sigma) \models (e_1, e_2)$ iff $(G_1^d \cup G_2^d, \Sigma) \models (e_1, e_2)$.

We check $(G_1^d \cup G_2^d, \Sigma) \models (e_1, e_2)$ by using Eq computed so far (see Section 3.2), denoted by $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$.

(2) Transitivity closure (TC). $\mathsf{EM_{MR}}$ computes the TC of Eq with the following rule: if (e_1, e_1'), (e_2, e_2') and (e_1', e_2') are in Eq, then so is (e_1, e_2); similarly for (e_1', e_1) and (e_2', e_2).

Algorithm. We now present the details of $\mathsf{EM_{MR}}$. It is controlled by a non-MapReduce driver $\mathsf{Driver_{MR}}$. $\mathsf{Driver_{MR}}$ first identifies candidate set L (line 1). For each entity e appearing in L, it constructs d-neighbors G^d also in MapReduce, by revising breadth-first search starting from e, with bound d. To avoid the cost of shipping *invariant input* data in MapReduce, these d-neighbors G^d and the set Σ of keys are *cached physically* in the local disk of processors, along the same lines as Haloop [BHBE10]. In addition, it stores a "global variable" Eq in HDFS, to keep track of entity pairs identified by Σ, initially the set of pairs (e, e) for all e in G (line 2).

It then triggers MapEM with key/value pairs $((e_1, e_2), (\textit{false}))$ for all $(e_1, e_2) \in L$ (line 4), with (e_1, e_2) as its key. MapReduce functions MapEM and ReduceEM then iterate to expand Eq. $\mathsf{Driver_{MR}}$ *terminates* the process when there is no change to Eq (line 5), and return Eq as chase(G, Σ) (line 6).

Mapper. Given a key/value pair $((e_1, e_2), (\mathsf{Flag}))$, MapEM first checks whether Flag $= \textit{true}$ (*i.e.,* $(e_1, e_2) \in$ Eq) or $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$ (line 1) by invoking a procedure $\mathsf{Eval_{MR}}$ (to be presented shortly). If so, MapEM emits value $(e_1, e_2, \textit{true})$ with keys e_1 and e_2, for computing TC (line 2). Otherwise, it emits value $(e_1, e_2, \textit{false})$ with key e_1 only (line 3), indicating the result of checking in this round.

Reducer. The input to ReduceEM is (e, list), where list includes all *newly identified* and un-identified pairs, and are collected in Eq(e) and $L(e)$, respectively (lines 1-4). ReduceEM then adds Eq(e) to Eq (line 5) and *joins* pairs in Eq(e) and Eq (line 6), to compute TC following the rule we have seen earlier. For those *newly joined* pairs (e_1, e_2) not in Eq, ReduceEM emits $((e_1, e_2), \textit{true})$ to expand TC in the next round, and Eq is updated by including (e_1, e_2) (line 7). For each (e_1, e_2) in $L(e)$ but not in Eq (line 8-9), $((e_1, e_2), (\textit{false}))$ is emitted for checking in the next round. Note that for each pair $(e_1, e_2) \in$ Eq, if (e_1, e_2) is not newly identified in this round, (e_1, e_2) is no longer in the process.

One can verify the following by induction on the length of chasing sequences for $(G, \Sigma) \models (e_1, e_2)$ (see Section 3.2).

Proposition 3.7: *If* $(G, \Sigma) \models (e_1, e_2)$, *then* (e_1, e_2) *is identified by* $\mathsf{EM_{MR}}$ *following the shortest chasing sequence.*   □

Proof: We prove it by induction on the length l of the shortest chasing sequence. The case when $l = 1$ is trivial, as $\mathsf{EM_{MR}}$ checks all the candidate pairs in parallel in the first round and all pairs identified in this step will not be checked further. Now assume that the proposition holds when $l < k$. For $l = k$, suppose that there exists a shortest chasing sequence of length k to identify (e_1, e_2), *i.e.,* $\mathsf{Eq_0}, \mathsf{Eq_1}, \ldots, \mathsf{Eq_k}$. Then it can be verified that all the entity pairs in $\mathsf{Eq_{k-1}}$ have been identified by $\mathsf{EM_{MR}}$ before the chasing step $\mathsf{Eq_{k-1}} \rightarrow_{(e_1, e_2)} \mathsf{Eq_k}$ as these entity pairs have shortest chasing steps of length less than k. And $\mathsf{EM_{MR}}$ repeatedly checks this chasing step as long as some entity pairs in $\mathsf{Eq_{k-1}}$ on which (e_1, e_2) depends are newly identified. From which we can conclude that $\mathsf{EM_{MR}}$ identifies (e_1, e_2) following the shortest chasing sequence.   □

Example 3.8: Algorithm $\mathsf{EM_{MR}}$ works on G_1 and Σ_1 of Example 3.7 as follows.

$\mathsf{Driver_{MR}}$ triggers MapEM with $((\mathsf{alb}_i, \mathsf{alb}_j), (\textit{false}))$ and $((\mathsf{art}_i, \mathsf{art}_j), (\textit{false}))$, where $i, j \in [1, 3]$, $i < j$. Note that $d = 1$ for Q_1, Q_2 and Q_3 of Fig. 3.1.

Round 1. MapEM identifies alb_1 and alb_2 with key $Q_2(x)$ by procedure $\mathsf{Eval_{MR}}$. ReduceEM adds $(\mathsf{alb}_1, \mathsf{alb}_2)$ to Eq, and joins it with Eq. They emit key/value pairs as follows.

MapEM	Emitted pairs	ReduceEM	Emitted pairs
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$((\mathsf{alb}_1), (\mathsf{alb}_1, \mathsf{alb}_2, \mathsf{T}))$ $((\mathsf{alb}_2), (\mathsf{alb}_1, \mathsf{alb}_2, \mathsf{T}))$	alb_1	$((\mathsf{alb}_1, \mathsf{alb}_3), (\mathsf{F}))$
$(\mathsf{alb}_1, \mathsf{alb}_3)$	$((\mathsf{alb}_1), (\mathsf{alb}_1, \mathsf{alb}_3, \mathsf{F}))$	alb_2	$((\mathsf{alb}_2, \mathsf{alb}_3), (\mathsf{F}))$
$(\mathsf{alb}_2, \mathsf{alb}_3)$	$((\mathsf{alb}_2), (\mathsf{alb}_2, \mathsf{alb}_3, \mathsf{F}))$		
$(\mathsf{art}_1, \mathsf{art}_2)$	$((\mathsf{art}_1), (\mathsf{art}_1, \mathsf{art}_2, \mathsf{F}))$	art_1	$((\mathsf{art}_1, \mathsf{art}_2), (\mathsf{F}))$ $((\mathsf{art}_1, \mathsf{art}_3), (\mathsf{F}))$
$(\mathsf{art}_1, \mathsf{art}_3)$	$((\mathsf{art}_1), (\mathsf{art}_1, \mathsf{art}_3, \mathsf{F}))$	art_2	$((\mathsf{art}_2, \mathsf{art}_3), (\mathsf{F}))$
$(\mathsf{art}_2, \mathsf{art}_3)$	$((\mathsf{art}_2), (\mathsf{art}_2, \mathsf{art}_3, \mathsf{F}))$		

Round 2. MapEM identifies $(\mathsf{art}_1, \mathsf{art}_2)$ by key $Q_3(x)$, and ReduceEM updates Eq. Note that no key/value pair for $(\mathsf{alb}_1, \mathsf{alb}_2)$ is in this round since it is in Eq already.

MapEM	Emitted pairs	ReduceEM	Emitted pairs
$(\mathsf{alb}_1, \mathsf{alb}_3)$	$((\mathsf{alb}_1), (\mathsf{alb}_1, \mathsf{alb}_3, \mathsf{F}))$	alb_1	$((\mathsf{alb}_1, \mathsf{alb}_3), (\mathsf{F}))$
$(\mathsf{alb}_2, \mathsf{alb}_3)$	$((\mathsf{alb}_2), (\mathsf{alb}_2, \mathsf{alb}_3, \mathsf{F}))$	alb_2	$((\mathsf{alb}_2, \mathsf{alb}_3), (\mathsf{F}))$
$(\mathsf{art}_1, \mathsf{art}_2)$	$((\mathsf{art}_1), (\mathsf{art}_1, \mathsf{art}_2, \mathsf{T}))$ $((\mathsf{art}_2), (\mathsf{art}_1, \mathsf{art}_2, \mathsf{T}))$	art_1	$((\mathsf{art}_1, \mathsf{art}_3), (\mathsf{F}))$
$(\mathsf{art}_1, \mathsf{art}_3)$	$((\mathsf{art}_1), (\mathsf{art}_1, \mathsf{art}_3, \mathsf{F}))$	art_2	$((\mathsf{art}_2, \mathsf{art}_3), (\mathsf{F}))$
$(\mathsf{art}_2, \mathsf{art}_3)$	$((\mathsf{art}_2), (\mathsf{art}_2, \mathsf{art}_3, \mathsf{F}))$		

Round 3. There is no newly identified entity pair, and Eq is not updated; $\mathsf{Driver_{MR}}$ thus terminates the process, and returns $\mathsf{chase}(G_1, \Sigma_1)$ with $(\mathsf{alb}_1, \mathsf{alb}_2)$ and $(\mathsf{art}_1, \mathsf{art}_2)$. $\quad \square$

Procedure $\mathsf{Eval_{MR}}$. We next show how to check $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$ with a key $Q(x)$ in Σ, in MapEM. A naive method is to first enumerate all matches of $Q(x)$ at e_1 in G_1^d and e_2 in G_2^d by using a subgraph isomorphism algorithm (*e.g.,* VF2 [CFSV04], TurboIso [HLL13]), and then check whether any those matches *coincide* (see Section 3.1). This involves two calls for a subgraph isomorphism algorithm, each of exponential cost. In other words, it is *not practical* to conduct the checking by using *any existing algorithm*.

To reduce the cost, we propose algorithm $\mathsf{Eval}_{\mathsf{MR}}$ that *combines* the two processes of computing (isomorphic) mappings into *a single process*, and allows *early termination*, *i.e.,* $\mathsf{Eval}_{\mathsf{MR}}$ terminates as soon as (e_1, e_2) is identified.

$\mathsf{Eval}_{\mathsf{MR}}$ conducts search *guided by* $Q(x)$, to instantiate nodes in $Q(x)$ with candidate pairs (s_1, s_2) in (G_1^d, G_2^d). We use a vector m to record the instantiation, combining mappings ν_1 and ν_2 from variables or values of $Q(x)$ to entities or values in G_1^d and G_2^d, respectively, and mapping μ for coinciding the two (see Section 3.1). For each node s_Q in $Q(x)$, (a) either $m[s_Q] = (s_1, s_2)$ when $s_1 = \nu_1(s_Q)$, $s_2 = \nu_2(s_Q)$, and $s_1 = \mu(s_2)$; (b) or $m[s_Q] = \bot$ if s_Q has no match yet.

(1) Initialization. More specifically, $\mathsf{Eval}_{\mathsf{MR}}$ initializes m with $m[x] = (e_1, e_2)$ and $m[s_Q] = \bot$ for all the rest. It then instantiates nodes of m one by one, as follows.

(2) Feasibility checking. To extend m with $m[s_Q] = (s_1, s_2)$, it checks the following *feasibility conditions*.

   (1) Injective: s_1 and s_2 do not appear in m already.

   (2) Equality: (a) if s_Q is y, then $(s_1, s_2) \in \mathsf{Eq}$; (b) if s_Q is $y*$, then $s_1 = s_2$ (values); (c) if s_Q is $\_y$, then s_1 and s_2 are entities of the same type; and (d) if s_Q is d, then $s_1 = s_2 = d$ (values).

   (3) Guided expansion: for all triples $(s_Q, p_Q, o_Q) \in Q(x)$, if o_Q is already instantiated, *i.e.,* $m[o_Q] = (o_1, o_2)$, then $(s_1, p_Q, o_1) \in G_1^d$ and $(s_2, p_Q, o_2) \in G_2^d$; similarly for all triples (s'_Q, p_Q, s_Q) in $Q(x)$.

$\mathsf{Eval}_{\mathsf{MR}}$ sets $m[s_Q] = (s_1, s_2)$ if *all* feasibility conditions are satisfied. Otherwise, it *backtracks* with other instantiation. Intuitively, m encodes a *partial* injective mapping from nodes in $Q(x)$ to candidate pairs in (G_1^d, G_2^d).

(3) Verification. When m is *fully instantiated, i.e.,* it contains no $\bot$, $\mathsf{Eval}_{\mathsf{MR}}$ concludes that $(G_1^d \cup G_2^d, \mathsf{Eq}, \{Q(x)\}) \models (e_1, e_2)$ and returns *true*. It returns *false* if m cannot be fully instantiated. This is correct since the feasibility conditions check all "nodes" and "edges" in $Q(x)$.

Lemma 3.8: $(G, \{Q(x)\}) \models (e_1, e_2)$ *if and only if m can be fully instantiated by* $\mathsf{Eval}_{\mathsf{MR}}$, *using key* $Q(x)$.   □

Proof: When m is fully instantiated, $(G, \{Q(x)\}) \models (e_1, e_2)$ is assured by the following. (1) By the feasibility checking $\mathsf{EM}_{\mathsf{MR}}$ adopts, ν_1 (resp. ν_2) is a valuation of $Q(x)$ in the set S_1 (resp. S_2) of triples that m encodes, in which S_1 and S_2 can be easily constructed by combining m with the corresponding predicates and values in $Q(x)$. (2) The injective condition enforces ν_1 and ν_2 to be bijections. (3) The equality condition

coincides with the semantic of keys defined in Section 3.1, *i.e.,* $S_1(e_1) \cong_Q S_2(e_2)$. From these it follows that (e_1, e_2) is identified by $Q(x)$. Conversely, if e_1 and e_2 are identified by $Q(x)$, then all the feasibility checking steps are satisfied if m is expanded by selecting the images of s_Q in S_1 and S_2 respectively, where s_Q is a variable in $Q(x)$. This will proceed until m is fully instantiated. □

When Σ contains multiple keys, $\mathsf{Eval_{MR}}$ identifies common sub-structures of keys along the same lines as [LKDL12]. It terminates *once there exists a key $Q(x)$ that identifies (e_1, e_2).*

Example 3.9: Continuing with Example 3.8, $\mathsf{Eval_{MR}}$ identifies art_1 and art_2 with $Q_3(x)$ in round 2, after alb_1 and alb_2 are identified by $Q_2(x)$ in round 1. It initializes $m[x]$ = $(\mathsf{art}_1, \mathsf{art}_2)$, and extends m with $m[\mathsf{name}*] = (\text{"The Beatles", "The Beatles"})$, and $m[\mathsf{album}] = (\mathsf{alb}_1, \mathsf{alb}_2)$ after feasibility check. As m is fully instantiated, $\mathsf{Eval_{MR}}$ returns *true*. □

Parallel scalability. To complete the proof of Theorem 3.6, we show that $\mathsf{EM_{MR}}$ is parallel scalable. Let G^d_m be the largest d-neighbor of all entities in G, where d is determined by the radius of keys in Σ defined on each entity, and p be the number of processors used. Then for each round of $\mathsf{EM_{MR}}$, MapEM takes at most $O(t(|G^d_m|, |\Sigma|)|L|/p)$ time, and $\mathsf{ReduceEM}$ takes $O(|\mathsf{Eq}|^2/p)$ time, where $t(|G^d_m|, |\Sigma|)$ is the cost of $\mathsf{Eval_{MR}}$. Moreover, at most $O(|\mathsf{Eq}|)$ rounds are needed since in each round, at least one pair is identified. Furthermore, $\mathsf{Driver_{MR}}$ constructs all G_d's in $O((|G^d_m||L| + |\Sigma|)/p)$ time. Putting these together, $\mathsf{EM_{MR}}$ is parallel scalable.

3.3.2 Optimization Strategies

From the analysis above, we can see that the cost of algorithm $\mathsf{EM_{MR}}$ is dominated by (a) the length of L, (b) the size of d-neighbors, and (c) redundant MapReduce computation. Below we study optimization strategies to reduce the cost.

Reducing L. Each $(e_1, e_2) \in L$ involves (repeated) isomorphism checking. Thus we filter those pairs that cannot be identified as follows. Given a key $Q(x)$, we say that (e_1, e_2) *can be paired by $Q(x)$* if there exists a ternary relation P^Q on nodes of $(G^d_1, G^d_2, Q(x))$ such that (1) $(e_1, e_2, x) \in P^Q$, (2) for each triple $(s_1, s_2, s_Q) \in P^Q$, (a) s_1 and s_2 are entities with same type if s_Q is y or $\_y$, $s_1 = s_2$ if s_Q is $y*$, or $s_1 = s_2 = d$ if $s_Q = d$; and (b) for each $(s_Q, p_Q, o_Q) \in Q(x)$, there exist (s_1, p_Q, o_1) in G^d_1 and (s_2, p_Q, o_2) in G^d_2 such that $(s_Q, p_Q, o_Q) \mapsto (s_1, p_Q, o_1)$, $(s_Q, p_Q, o_Q) \mapsto (s_2, p_Q, o_2)$,

and $(o_1, o_2, o_Q) \in P^Q$; similarly for $(s'_Q, p_Q, s_Q) \in Q(x)$. We refer to P^Q as a *pairing relation* of Q at (e_1, e_2).

One can verify that pairing is a necessary condition for (e_1, e_2) to be identified by key $Q(x)$. Hence we include in L only those pairs that are paired by *some* key $Q(x) \in \Sigma$.

Proposition 3.9: *For any pair (e_1, e_2), (a) if e_1 and e_2 cannot be paired by any key in Σ, then $(G, \Sigma) \not\models (e_1, e_2)$; and (b) if (e_1, e_2) can be paired by a key $Q(x)$, then there exists a unique maximum pairing relation P^Q of $Q(x)$ at (e_1, e_2), and P^Q can be computed in $O(|Q||G_1^d||G_2^d|)$ time.* $\square$

Proof:(a) It suffices to prove that if $(G, \Sigma) \models (e_1, e_2)$, then e_1 and e_2 can be paired by some key in Σ. Assume that (e_1, e_2) is included in Eq by applying a key $Q(x)$ in Σ, *i.e.,* there exist matches S_1 and S_2 of $Q(x)$ at e_1 and e_2 in G under ν_1 and ν_2, respectively, such that $S_1(e_1) \cong_Q^{\mathsf{Eq}} S_2(e_2)$. It can be verified that (e_1, e_2) can be paired by $Q(x)$ with the ternary relation P^Q built as follows: (1) $(e_1, e_2, x) \in P^Q$, and (2) for each s_Q in $Q(x)$, $(\nu_1(s_Q), \nu_2(s_Q), s_Q) \in P^Q$, similarly for each o_Q in $Q(x)$. Indeed, all the conditions of the semantics of P^Q are satisfied as ν_1 and ν_2 are valuations of $Q(x)$ in S_1 and S_2, respectively, and there also exists a bijection μ between S_1 and S_2, mapping $\nu_1(s_Q)$ to $\nu_2(s_Q)$ and preserving value equality.

(b) We first show that if (e_1, e_2) can be paired by Q with ternary relations P_1^Q and P_2^Q respectively, then (e_1, e_2) can also be paired with relation $P_1^Q \cup P_2^Q$. Obviously the conditions (1) and (2)(a) of the semantics of P^Q are still satisfied by $P_1^Q \cup P_2^Q$. Moreover, since no triple in P_1^Q and P_2^Q is removed, the existential semantics of (2)(b) and (2)(c) also hold. Therefore there exists a maximum pairing relation P^Q, which is the union of all pairing relation of $Q(x)$ at (e_1, e_2). We then prove the uniqueness by contradiction. Assume that there exist two distinct maximum pairing relation P_1^Q and P_2^Q. Then $P_1^Q \cup P_2^Q$ is a pairing relation larger than both P_1^Q and P_2^Q, which is a contradiction.

Next we show that it is in cubic-time to compute P^Q. Firstly a *product graph* $G_p = (V_p, E_p)$ of G_1^d and G_2^d is built, where each node in V_p is a pair (s_1, s_2) taken from G_1^d and G_2^d respectively and (1) s_1 and s_2 are entities of the same type or $s_1 = s_2 = d$, (2) $(e_1, e_2) \in V_p$, (3) $((s_1, s_2), p, (o_1, o_2)) \in E_p$ if and only if $(s_1, p, o_1) \in G_1^d$ and $(s_2, p, o_2) \in G_2^d$. The computation of G_p can be done in $O(|E_{G_1^d}||E_{G_2^d}|)$ time and $|V_p| = O(|V_{G_1^d}||V_{G_2^d}|)$, $|E_p| = O(|E_{G_1^d}||E_{G_2^d}|)$. After that we compute a maximum match relation M of $Q(x)$ on G_p based on the semantic of *dual simulation*, where x in $Q(x)$ is

mapped to (e_1, e_2), y and y_- are mapped to entity pairs of the same type and $y*$ could be mapped to any value pair. For each s_Q in $Q(x)$ that is mapped to (s_1, s_2) in M, we add a triple (s_1, s_2, s_Q) to P^Q, similarly for o_Q. By leveraging the graph simulation algorithm, this step can be done in $O((|V_Q| + |E_Q|)(|V_p| + |E_p|))$ time. Thus it has a complexity bound of $O((|V_Q| + |E_Q|)(|V_{G_1^d}||V_{G_2^d}| + |E_{G_1^d}||E_{G_2^d}|))$. □

Reducing (G_1^d, G_2^d). For each $(e_1, e_2) \in L$, we reduce (G_1^d, G_2^d) such that they are subgraphs induced by those nodes that are in the maximum pairing relation P^Q at (e_1, e_2) by some key $Q(x)$ of Σ. Extending Proposition 3.9, one can verify that $(G, \Sigma) \models (e_1, e_2)$ if and only if (e_1, e_2) can be identified by keys in (G_1^d, G_2^d) constructed in this way.

Reducing redundant MapReduce **computation**. We develop two optimization strategies by leveraging the dependency imposed by recursively defined keys. We say that a pair (e_1, e_2) *depends on* (e_1', e_2') if (e_1', e_2') is (a) in d-neighbors of (e_1, e_2); and (b) has the same type as y, where y is a variable in a recursive key in Σ defined on (e_1, e_2).

Entity dependency. Driver$_{MR}$ collects a set L_0 with pairs $(e_1, e_2) \in L$, such that only value-based keys in Σ are defined on. Driver$_{MR}$ triggers MapEM with pairs in L_0 *only*, instead of the entire L. In each MapReduce round, a new pair (e_1', e_2') is emitted only when (e_1', e_2') depends on (e_1, e_2), and if (e_1, e_2) has been already proceeded.

Incremental checking. We revise MapEM such that $(G_1^d \cup G_2^d, \text{Eq}, \Sigma) \models (e_1, e_2)$ is checked only in the first round or when some pairs (e_1', e_2') on which (e_1, e_2) depends are *identified in the last round*, to reduce the expensive checking. This is done by adding a flag Changed to the pairs in Eq.

3.4 A Vertex-Centric Algorithm

The performance of algorithm EM$_{MR}$ is hampered by (1) the maintenance of global variable Eq; and (2) stragglers in each round that may hold up the process of a chain of entity pairs on which dependencies are imposed by recursively defined keys. Such costs are inherent to the I/O bound property and the synchronization policy of MapReduce.

To reduce the costs, we develop an algorithm for entity matching in the vertex-centric model of [LGK$^+$12]. As opposed to MapReduce, [LGK$^+$12] is based on a vertex program that is executed in parallel *on each vertex*, and interacts with the neighbors of the vertex via asynchronous message passing. There is no need for a global variable

Eq, or for synchronizing the computation into rounds. We show the following.

Theorem 3.10: *There exist parallel scalable algorithms in the vertex-centric model of [LGK$^+$12] for entity matching.* $\square$

As a proof, we present such an algorithm (Section 3.4.1), and develop optimization strategies (Section 3.4.2).

3.4.1   Algorithm and Parallel Scalability

The algorithm, referred to as $\mathsf{EM}_{\mathsf{VC}}$, computes $\mathsf{Chase}(G,\Sigma)$ when given a graph G and a set Σ of keys. It works as follows. Let L be the set of candidate pairs to be checked. For all $(e_1,e_2) \in L$, it checks whether $(G,\Sigma) \models (e_1,e_2)$. Similar to $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}_{\mathsf{VC}}$ adds a pair (e_1,e_2) to Eq *once* it is identified by any key in Σ, and it checks all pairs in L in *parallel*. In contrast to $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}_{\mathsf{VC}}$ follows *asynchronous message passing* [LGK$^+$12]. To determine whether $(G,\Sigma) \models (e_1,e_2)$, it checks *different instantiations* of nodes in a key *in parallel* with multiple messages, for all keys defined on (e_1,e_2).

When $(G,\Sigma) \models (e_1,e_2)$ is confirmed, $\mathsf{EM}_{\mathsf{VC}}$ notifies those pairs $(s_1,s_2) \in L$ that depend on (e_1,e_2) by sending messages, so that $(G,\Sigma) \models (s_1,s_2)$ is checked "incrementally". The transitive closure (TC) of Eq is computed by message propagation at the same time. The process proceeds until no messages are active and Eq can no longer be changed.

The key ideas behind $\mathsf{EM}_{\mathsf{VC}}$ include *guided search* for verifying $(G,\Sigma) \models (e_1,e_2)$ and *expansion* of TC based on the dependency of entities, both via asynchronous message passing. To facilitate message passing, $\mathsf{EM}_{\mathsf{VC}}$ uses the following.

Product graph. Given G and Σ, $\mathsf{EM}_{\mathsf{VC}}$ constructs a *product graph* $G_p = (V_p, E_p)$, where each node in V_p is either (a) a pair (o_1,o_2) of entities or values that can be *paired* (see Proposition 3.9); or (b) a pair (e,e) of entities *only if e is paired with* another entity in V_p. There is an edge $((s_1,s_2), p, (o_1,o_2))$ in E_p from node (s_1,s_2) to (o_1,o_2) if (a) (s_1,p,o_1) and (s_2,p,o_2) are both in G; (b) (o_1,o_2) depends on (s_1,s_2) (see Section 3.3.2); here p is a special label dep; or (c) $o_1 \Leftrightarrow o_2$, and $o_1 \Leftrightarrow s_1$ or $o_1 \Leftrightarrow s_2$; p is labeled as tc in this case.

Intuitively, G_p encodes the topology of G, the dependency on entities *w.r.t.* Σ via dep edges, and the transitive closure of Eq via tc edges. We do not include (e,e) in G_p if e is not in L, *i.e.,* if e is not to be paired with another entity or if no key in Σ is defined on e. In our experiments, we find that $|G_p| = 2.7 * |G|$ on average, *much smaller than*

$|G|^2$.

For each (e_1, e_2) in G_p, a Boolean $\mathsf{Flag}(e_1, e_2)$ is used to indicate whether $(e_1, e_2) \in$ Eq, initially *false* unless $e_1 \Leftrightarrow e_2$.

Traversal order. For each key $Q(x)$ in Σ, $\mathsf{EM_{VC}}$ defines a sorted list P_Q of triples in $Q(x)$ such that (a) *all nodes* in $Q(x)$ appear in some triples in P_Q, and (b) it encodes a "tour" of nodes in $Q(x)$, starting from x and ending at x.

Intuitively, $\mathsf{EM_{VC}}$ propagates messages *guided by P_Q*. Together with feasibility checking to be seen shortly, a complete tour that starts from (e_1, e_2) guided by P_Q guarantees that (e_1, e_2) can be identified by $Q(x)$. There are multiple orders for a tour of $Q(x)$. However, finding an optimum order with a shortest tour is NP-complete, by reduction from *Chinese Postman Problem* (cf. [GJ79]). In light of this, $\mathsf{EM_{VC}}$ uses a greedy algorithm to decide P_Q.

Algorithm. $\mathsf{EM_{VC}}$ first constructs G_p as above. Then at each node (e_1, e_2) in V_p, if a value-based key in Σ is defined on it, it triggers procedure $\mathsf{Eval_{VC}}$ for subgraph isomorphism checking, propagates messages to activate other nodes in V_p guided by traversal order, and computes the TC of Eq. $\mathsf{EM_{VC}}$ *terminates* when no messages are active, and it returns Eq of all pairs (e_1, e_2) with $\mathsf{Flag}(e_1, e_2) = true$, as $\mathsf{chase}(G, \Sigma)$.

Procedure $\mathsf{Eval_{VC}}$. At each node (s_1, s_2) in G_p, the actions of $\mathsf{EM_{VC}}$ are summarized in $\mathsf{Eval_{VC}}$, shown in Fig. 3.7.

(1) Initial message. When $\mathsf{Eval_{VC}}$ is activated at a node (s_1, s_2) in G_p, for each key $Q(x) \in \Sigma$ defined on (s_1, s_2), an *initial message* $m_Q(s_1, s_2)$ is created (lines 1-2, (1), Fig. 3.7), with $m_Q(s_1, s_2)[x] = (s_1, s_2)$ and $m_Q(s_1, s_2)[s_Q] = \bot$ for all other nodes in $Q(x)$. The message is a vector that encodes a partial injective mapping from nodes in $Q(x)$ to nodes in G_p, similar to those used by procedure $\mathsf{Eval_{MR}}$ (Section 3.3.1).

Then *guided* by the *first* triple (x, p_Q, o_Q) (or (s_Q, p_Q, x)) of P_Q, a *copy* of $m_Q(e_1, e_2)$ is "forked" to propagate to each neighbor (o_1, o_2) of (s_1, s_2), following edge $((s_1, s_2), p_Q, (o_1, o_2))$ in G_p (line 3), for feasibility check (see (4) below).

(2) Early cancellation. Upon receiving a message $m_Q(e_1, e_2)$ at (s_1, s_2), (s_1, s_2) first checks whether $\mathsf{Flag}(e_1, e_2)$ is *true*, by sending a message to (e_1, e_2), whose ID is in $m_Q(e_1, e_2)$. If so, $\mathsf{Eval_{VC}}$ stops the propagation of $m_Q(e_1, e_2)$ (lines 1-2, (2), Fig. 3.7), since (e_1, e_2) is already identified.

(3) Verification. If $\mathsf{Flag}(e_1, e_2)$ is *false*, but $m_Q(e_1, e_2)$ is *fully instantiated*, *i.e.,* it does not contain $\bot$, and moreover, if (e_1, e_2) is (s_1, s_2), *i.e.,* $m_Q(e_1, e_2)$ has completed its

---

Algorithm Eval_{VC} /* Executed at each node (s_1, s_2) */

(1) Initial messages at (s_1, s_2)

1. **for each** key $Q(x) \in \Sigma$ defined on (s_1, s_2) **do**
2.   create an initial message $m_Q(s_1, s_2)$;
3.   propagate $m_Q(s_1, s_2)$ guided by order P_Q;

(2) Upon receiving a message $m_Q(e_1, e_2)$ following (s_Q, p_Q, o_Q)

1. **if** $\text{Flag}(e_1, e_2) = true$**then**
2.   stop propagating $m_Q(e_1, e_2)$; **return**;
3. **if** $m_Q(e_1, e_2)$ is fully instantiated and $(e_1, e_2) = (s_1, s_2)$ **then**
4. $\text{Flag}(e_1, e_2) := true$; compute dependency and TC; **return**;
5. **if** either $m_Q(e_1, e_2)[s_Q]$ or $m_Q(e_1, e_2)[o_Q]$ is $\perp$ **then**
6.   **if** $m_Q(e_1, e_2)$ satisfies all feasibility conditions at (s_1, s_2) **then**
7.   extend $m_Q(e_1, e_2)$ by instantiating a node with (s_1, s_2);
8.   **else** drop $m_Q(e_1, e_2)$; **return**;
9. propagate $m_Q(e_1, e_2)$ guided by order P_Q;

(3) Compute dependency and TC when $\text{Flag}(e_1, e_2)$ becomes true

1. **if** $((e_1, e_2), \text{dep}, (s_1, s_2)) \in G_p$, and $\text{Flag}(s_1, s_2) = false$ **then**
2.   propagate increment message $m_{Q'}(s_1, s_2)$ for each $Q'(x)$ of Σ;
3. **if** $((e_1, e_2), \text{tc}, (s_1, s_2)) \in G_p$ **then**
4.   compute transitive closure of Eq;

Figure 3.7: Algorithm Eval_{VC}

propagation and is sent back to (e_1, e_2), guaranteed by the guided order P_Q, then we can conclude that $(G, \{Q(x)\}) \models (e_1, e_2)$ (see Lemma 3.11 below). Hence $\text{Flag}(e_1, e_2)$ is set *true*, (e_1, e_2) *notifies* nodes that depend on (e_1, e_2) following edges labeled dep, and activates those nodes following edges labeled tc, to compute the TC of Eq (lines 3-4, see (6) and (7) below).

(4) Feasibility checking. Otherwise, assume that $m_Q(e_1, e_2)$ is sent to (s_1, s_2) following triple (s_Q, p_Q, o_Q) in P_Q. If $m_Q(e_1, e_2)[s_Q] = \perp$ (similarly for $m_Q(e_1, e_2)[o_Q] = \perp$), Eval_{VC} checks whether $m_Q(e_1, e_2)[s_Q]$ can be instantiated with (s_1, s_2) (lines 5-6) based on the same feasibility conditions of Eval_{MR} (*injective*, *equality* and *guided expansion*; Section 3.3.1), except that when s_Q is a variable y, it requires $\text{Flag}(s_1, s_2) = true$. If

it does not pass the check, $m_Q(e_1,e_2)$ is *dropped* (line 8), as $m_Q(e_1,e_2)$ cannot be expanded. Otherwise Eval_{VC} sets $m_Q(e_1,e_2)[s_Q] = (s_1,s_2)$ (line 7).

(5) Guided propagation. Now, both $m_Q(e_1,e_2)[s_Q]$ and $m_Q(e_1,e_2)[o_Q]$ are instantiated. Then (s_1,s_2) propagates message $m_Q(e_1,e_2)$ guided by the *next* triple (s'_Q, p'_Q, o'_Q) in P_Q, *i.e.*, the successor of (s_Q, p_Q, o_Q) in P_Q (line 9). Assuming that $m_Q(e_1,e_2)[s'_Q] = (s_1,s_2)$ (the case is similar if $m_Q(e_1,e_2)[o'_Q] = (s_1,s_2)$), Eval_{VC} does the following.

(a) If $m_Q(e_1,e_2)[o'_Q] = (o_1,o_2)$, *i.e.,* message $m_Q(e_1,e_2)$ has already been instantiated with (o_1,o_2), then $m_Q(e_1,e_2)$ is sent "back" to (o_1,o_2) directly.

(b) If $m_Q(e_1,e_2)[o'_Q] = \bot$, a *copy* of $m_Q(e_1,e_2)$ is propagated to each neighbor (o_1,o_2) of (s_1,s_2), following edge $((s_1,s_2), p_Q, (o_1,o_2))$ in G_p. If no such neighbor exists, $m_Q(e_1,e_2)$ is dropped.

One can verify that $m_Q(e_1,e_2)$ is propagated only *within* the d-neighbor of (e_1,e_2) in G_p as it is guided by P_Q.

(6) Dependency. When $\text{Flag}(e_1,e_2)$ is set *true*, Eval_{VC} notifies all nodes (s_1,s_2) that depend on (e_1,e_2), by following edge $((e_1,e_2), \text{dep}, (s_1,s_2))$ (see (3) above). Then Eval_{VC} is activated at (s_1,s_2). It checks whether $\text{Flag}(s_1,s_2)$ is *false*(line 1, (3), Fig. 3.7). If so, Eval_{VC} triggers *increment messages* $m_{Q'}(s_1,s_2)$ for each $Q'(x)$ defined on (s_1,s_2), with $m_{Q'}(s_1,s_2)[x] = (s_1,s_2)$, $m_{Q'}(s_1,s_2)[y] = (e_1,e_2)$ and $m_{Q'}(s_1,s_2)[z_Q] = \bot$ for other nodes in $Q'(x)$, where y is a variable in $Q'(x)$ with the same type of (e_1,e_2) (line 2). These messages are propagated in the same way as above.

(7) Transitive closure. When $\text{Flag}(e_1,e_2)$ is *true*, (s_1,s_2) is notified if $((e_1,e_2), \text{tc}, (s_1,s_2)) \in G_p$. Assume *w.l.o.g.* that $(s_1,s_2) = (e_1,e_1)$. Then at (e_1,e_1), (e_1,e_2) is *joined* with (e_1,e), when either (a) $e \Leftrightarrow e_1$ or (b) $((e_1,e), \text{tc}, (e_1,e_1)) \in G_p$ and $\text{Flag}(e_1,e) = true$; it sets $\text{Flag}(e_2,e) = true$. The newly identified nodes conduct the same process following tc edges, to further compute the TC (lines 3-4, (3), Fig. 3.7).

The correctness of Eval_{VC} is warranted by the following.

Lemma 3.11: $(G, \{Q(x)\}) \models (e_1,e_2)$ *if and only if there exists a message $m_Q(e_1,e_2)$ that can be fully instantiated by algorithm* Eval_{VC}*; and the message is propagated at most $2|Q|$ times.* $\qquad\square$

Proof: The *if and only if* condition follows from Lemma 3.8. By induction on $|Q|$, we show that the message is propagated at most $2|Q|$ times. The case when $|Q| = 1$ is trivial, as P_Q contains a single triple and $m_Q(e_1,e_2)$ is propagated at most twice, once forward and then backward to (e_1,e_2). Assume the proposition holds when $|Q| = k$.
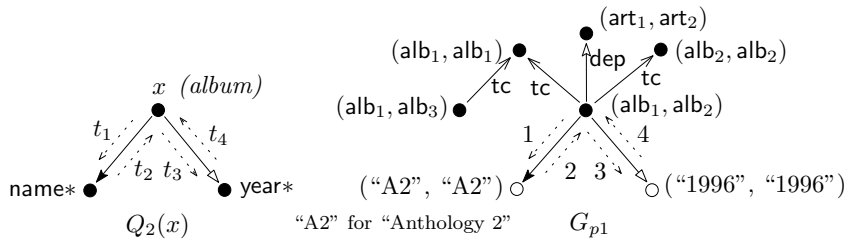
Figure 3.8: Message propagation in $\mathsf{Eval}_{\mathsf{VC}}$

For $|Q'| = k+1$, assume *w.l.o.g.* that $(s_{Q'}, p_{Q'}, o_{Q'})$ is newly added to Q', then it can be verified that compared with P_Q, the length of $P_{Q'}$ is incremented at most by two, *i.e.,* $(s_{Q'}, p_{Q'}, o_{Q'})$ and $(o_{Q'}, p_{Q'}, s_{Q'})$ are included in $P_{Q'}$, and the connectivity of $Q(x)$ and $Q'(x)$ warranties the correctness of $P_{Q'}$. From which we conclude that the propagation of $m_{Q'}(e_1, e_2)$ is also enlarged by at most two steps comparing to that of $m_Q(e_1, e_2)$. Thus the proof of Lemma 3.11.   □

Example 3.10: We show how $\mathsf{EM}_{\mathsf{VC}}$ works on G_1 and Σ_1 of Example 3.7. A (partial) product graph G_{p_1} of G_1 is shown in Fig. 3.8, where $(\mathsf{art}_1, \mathsf{art}_2)$ depends on $(\mathsf{alb}_1, \mathsf{alb}_2)$.

For Q_2, the order P_{Q_2} is $[t_1, t_2, t_3, t_4]$, where t_1 and t_2 are $(x, \mathsf{name\_of}, \mathsf{name}*)^{+/-}$, and t_3 and t_4 are $(x, \mathsf{release\_year}, \mathsf{year}*)^{+/-}$, respectively; here $+$ and $-$ indicate forward and backward traversal, respectively. At $(\mathsf{alb}_1, \mathsf{alb}_2)$, $\mathsf{Eval}_{\mathsf{VC}}$ constructs initial message m_{Q_2} for $Q_2(x)$, where $m_{Q_2}[x] = (\mathsf{alb}_1, \mathsf{alb}_2)$, and $\perp$ for the other nodes. As shown in Fig. 3.8, it propagates m as follows, guided by P_{Q_2}.

Node (m_{Q_2} visits)	Feasibility checking	P_{Q_2}
("A2", "A2")	$m_{Q_2}[\mathsf{name}*] = $ ("A2", "A2")	t_2
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$m_{Q_2}[x]$ is instantiated	t_3
("1996", "1996")	$m_{Q_2}[\mathsf{year}*] = $ ("1996", "1996")	t_4
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$\mathsf{Flag}(\mathsf{alb}_1, \mathsf{alb}_2) = true$	

When m_{Q_2} is sent back to $(\mathsf{alb}_1, \mathsf{alb}_2)$, it is fully instantiated, and $\mathsf{Flag}(\mathsf{alb}_1, \mathsf{alb}_2)$ is set *true*. $\mathsf{Eval}_{\mathsf{VC}}$ then notifies node $(\mathsf{art}_1, \mathsf{art}_2)$ via edge labeled dep, triggers an increment message m_{Q_3} for $Q_3(x)$ there, and identifies $(\mathsf{art}_1, \mathsf{art}_2)$ along the same lines. While some other nodes are notified by following tc edges for computing TC, no new entity pairs are derived. At this point, no message is in transit, and $\mathsf{EM}_{\mathsf{VC}}$ returns all entity pairs with $\mathsf{Flag} = true$.   □

Parallel scalability. We show that algorithm $\mathsf{EM}_{\mathsf{VC}}$ is parallel scalable. The total amount of computation by $\mathsf{EM}_{\mathsf{VC}}$ is at most $O(t(|G_p^d|, |\Sigma|)|L||\mathsf{Eq}|)$, where G_p^d is the

maximum d-neighbor of entity pairs in G_p, Eq is the set of entity pairs identified by $\mathsf{EM_{VC}}$, and $O(t(|G_p^d|,|\Sigma|))$ is the time for checking $(G,\Sigma) \models (e_1,e_2)$ via message passing. Indeed, each pair may be checked $|\mathsf{Eq}|$ times in the worst case, triggered by increment messages, due to the dependency on the nodes of G_p imposed by recursively defined keys in Σ. Assume that the work is distributed evenly across p processors, *i.e.,* the resources of an idle node are re-allocated to process other nodes as conducted in the vertex-centric model [LGK$^+$12], and that $p \ll |G|$. Then $\mathsf{EM_{VC}}$ is in $O(t(|G_p^d|,|\Sigma|)|L||\mathsf{Eq}|/p)$ time.

From this and Lemma 3.11, Theorem 3.10 follows.

3.4.2 Optimization Strategies

Procedure $\mathsf{Eval_{VC}}$ may fork excessive messages and incur redundant computation. To reduce the cost, we adopt prior optimization techniques [LKDL12] to extract common sub-structures of keys in Σ. In addition, we present another two strategies to reduce excessive messages.

Bounded messages. To check $(G, \{Q(x)\}) \models (s_1,s_2)$, $\mathsf{Eval_{VC}}$ generates at most k messages, for a (user-defined) constant k. To do this, we revise $\mathsf{Eval_{VC}}$ as follows.

(1) When $\mathsf{Eval_{VC}}$ is activated at (s_1,s_2), a variable $K_Q(s_1,s_2)$ is defined to keep track of the number of copies of $m_Q(s_1,s_2)$ that are active, initially 1 for the initial message.

(2) Suppose that $m_Q(e_1,e_2)[s_Q]$ is instantiated with (s_1,s_2) (while $m_Q(e_1,e_2)[o_Q] = \bot$). $\mathsf{Eval_{VC}}$ propagates $m_Q(e_1,e_2)$ guided by a triple (s_Q,p_Q,o_Q) in P_Q as follows.

  ○ If $K_Q(e_1,e_2) < k$, for each edge $((s_1,s_2),p_Q,(o_1,o_2))$ in G_p that is yet *unmarked* with (s_Q,p_Q,o_Q) for $m_Q(e_1,e_2)$, a new copy of $m_Q(e_1,e_2)$ is propagated to (o_1,o_2), and $K_Q(e_1,e_2)$ is increased by 1, until $K_Q(e_1,e_2) = k$ or all unmarked edges are covered.

  ○ Otherwise (if there is no budget for new copies), $m_Q(e_1,e_2)$ is propagated following an unmarked edge $((s_1,s_2), p_Q,(o_1,o_2))$, without forking new copies.

Those edges $((s_1,s_2),p_Q,(o_1,o_2))$ that message $m_Q(e_1,e_2)$ follows are *marked* with (s_Q,p_Q,o_Q) for $m_Q(e_1,e_2)$, to avoid repeated checking. The process is similar if $m_Q(e_1,e_2)[o_Q] = (s_1,s_2)$ and $m_Q(e_1,e_2)[s_Q] = \bot$.

(3) When there are no nodes to propagate, or the feasibility conditions are not satisfied, $m_Q(e_1,e_2)$ will *backtrack* to check other instantiation, instead of being dropped.

In this way, to check whether $(G,\Sigma) \models (e_1,e_2)$, at most $O(k\|\Sigma\||\mathsf{Eq}|)$ messages are
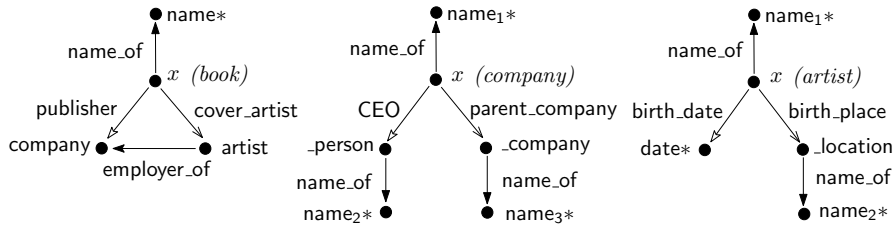
Figure 3.9: Keys defined on DBpedia

generated and propagated.

Prioritized propagation. When $\mathsf{Eval}_{\mathsf{VC}}$ picks an unmarked edge to propagate message $m_Q(e_1, e_2)$ from (s_1, s_2), it selects an edge with the highest *potential* that can make $m_Q(e_1, e_2)$ fully instantiated. This is estimated based on the number of neighbors of (o_1, o_2) that have the same types and values as those variables in $m_Q(e_1, e_2)$ to be instantiated. Such information is collected when constructing G_p.

3.5 Experimental Study

Using real-life and synthetic graphs, we conducted three sets of experiments on $\mathsf{EM}_{\mathsf{MR}}$ and $\mathsf{EM}_{\mathsf{VC}}$ to evaluate the impacts of (1) the number p of processors used; (2) the size of graph G; and (3) the complexity of keys Σ (see below). The results verify that the algorithms are parallel scalable and can efficiently identify entities in reasonably large graphs.

Experimental setting. We used two real-life graphs: (a) *Google+* [GXH^{+}12] (Google in short), a social network with 2.6 million nodes and 17.5 million edges (relationships such as *friend*), where 30 types of entities are determined by its node attributes, *e.g., major, university, place* and *employer*; and (b) DBpedia [DBp], a knowledge base with 4.3 million nodes and 40.3 million links, including 495 types of entities.

We also developed a generator to produce synthetic graphs G, controlled by the number of entities $\mathcal{E}$ and data values $\mathcal{D}$. Predicates $\mathcal{P}$ and entity types Θ were drawn from an alphabet $\mathcal{L}$ of 6000 labels. The size of G is up to 95 million entities (100 million nodes) and 500 million edges.

Key generator. We generated keys Σ controlled by the maximum radius d and the length c of longest *dependency chains* from recursively defined keys in Σ. (1) We constructed 30 and 100 keys for Google and DBpedia, respectively, with attributes and predicates from the data graphs. Some keys for DBpedia are shown in Fig. 3.9. (2) For

synthetic graphs, we randomly generated 500 keys for different types of entities in Θ, with values from $\mathcal{D}$ and predicates from $\mathcal{P}$.

Algorithms. We implemented the following algorithms: (1) MapReduce algorithms on Hadoop 1.2.1: (a) $\mathsf{EM}_{\mathsf{MR}}$ of Section 3.3.1, (b) $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{VF2}}$, which replaces $\mathsf{Eval}_{\mathsf{MR}}$ of $\mathsf{EM}_{\mathsf{MR}}$ with VF2 [CFSV04] by enumerating all matches without early termination; (c) $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{Opt}}$, a revision of $\mathsf{EM}_{\mathsf{MR}}$ by supporting the optimization strategies of Section 3.3.2. (2) Vertex-centric algorithms on GraphLab [LGK$^+$12]: (a) $\mathsf{EM}_{\mathsf{VC}}$ of Section 3.4.1, and (b) $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$, which optimizes $\mathsf{EM}_{\mathsf{VC}}$ by using $k = 4$ messages and prioritized message propagation strategy (Section 3.4.2). Conventional algorithms for subgraph isomorphism algorithms and entity resolution do not work on entity matching and graphs, respectively, and hence, cannot be compared with.

Distributed sites. We deployed the graphs, keys and algorithms on $p \in [4, 20]$ machines of Amazon EC2 Compute-Optimized Instance c4.4xlarge. Each experiment was run 3 times and the average is reported here.

Experimental results. We next report our findings. In all the experiments, we used 30, 100 and 500 keys for Google, DBpedia and Synthetic respectively.

Exp-1: Varying p. Fixing $c = 2$ and $d = 2$, we first evaluated the parallel scalability of these algorithms by varying p from 4 to 20. The results are reported in Figures 3.10(a), 3.10(b) and 3.10(c), for Google, DBpedia and Synthetic (fixing $G = (100M, 500M)$), respectively, in which we use *logarithmic scale* for the *y*-axis. We find the following.

Parallel scalability. On a given graph, these algorithms took less time proportional to the increase of processors. For instance, $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$ (resp. $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{Opt}}$) are 4.8, 4.7 and 5 times faster (resp. 4.6, 4.7 and 4.8) when p increases from 4 to 20 on Google, DBpedia and Synthetic, respectively. We find that $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$ scales the best among all the algorithms: it takes 2.4 seconds to identify all entities in Google with 20 processors.

We also experimented with p up to 32. The results are consistent with Figures 3.10(a), 3.10(b) and 3.10(c): the algorithms are 1.5 times faster than the setting with $p = 20$ on average. These experimentally verify Theorems 3.6 and 3.10, *i.e.,* our algorithms are parallel scalable, despite Theorems 3.2 and 3.4.

MapReduce vs. vertex-centric. Algorithm $\mathsf{EM}_{\mathsf{VC}}$ outperforms all the MapReduce algorithms, even $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{Opt}}$. It is at least 12.1, 10.9 and 13.5 times faster on Google, DBpedia and Synthetic, respectively. For instance, it takes 5.8 seconds on Google when $p = 12$,
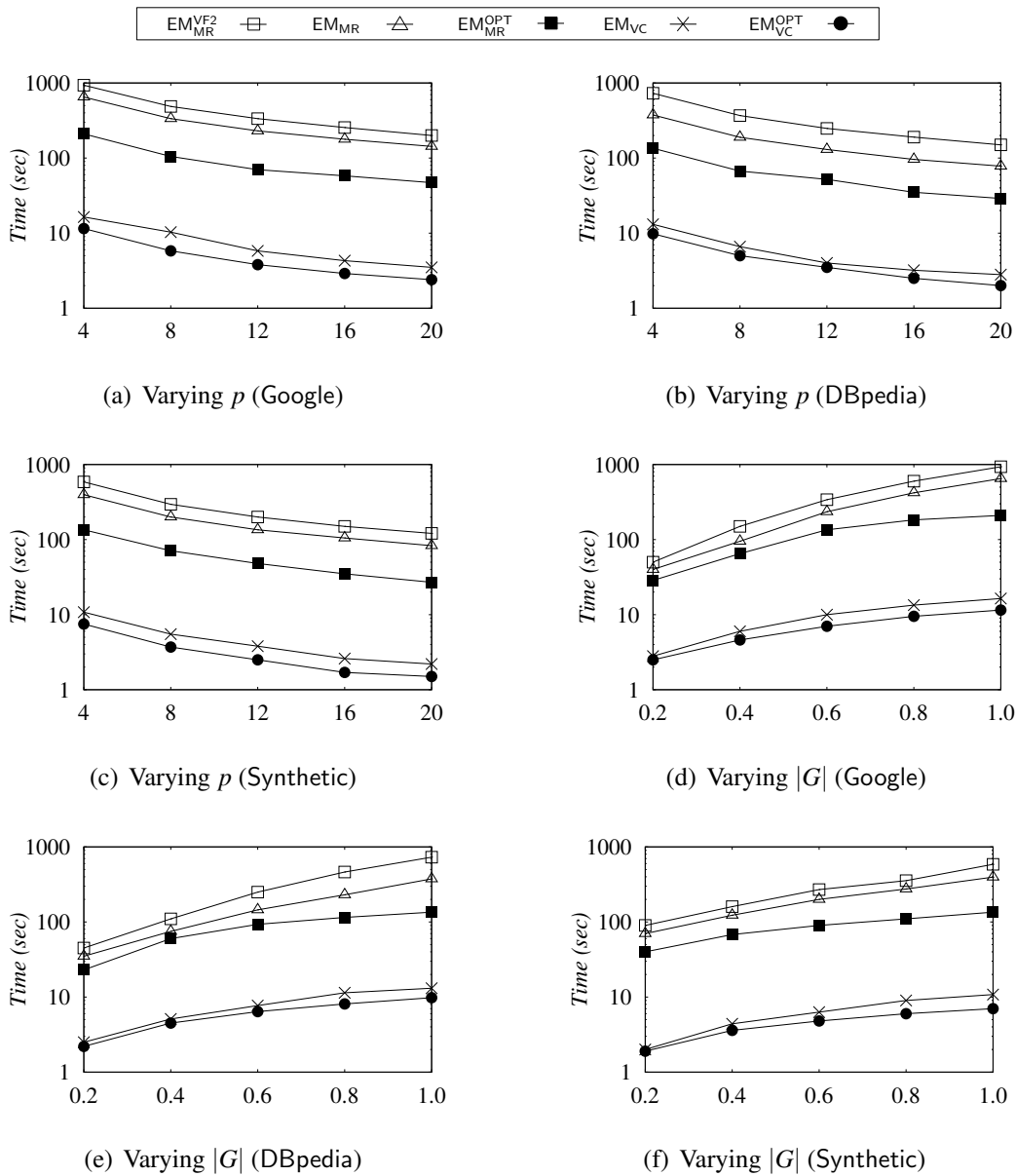
Figure 3.10: Scalability of entity matching

while EM_{MR}^{Opt} takes 70 seconds. This verifies that EM_{VC} reduces the inherent costs of the I/O bound and the synchronization policy of MapReduce.

We developed and evaluated EM_{MR}^{Opt} and EM_{MR} because of the prevalent use of the MapReduce framework. Moreover, EM_{MR}^{Opt} may be advantageous to EM_{VC} when EM_{VC} requires a product graph much larger than G (see Section 3.4 and below).

Effectiveness of optimization. (1) EM_{MR} is 1.5, 1.9 and 1.4 times faster than EM_{MR}^{VF2} on average on Google, DBpedia and Synthetic, respectively. This verifies the effectiveness of procedure $Eval_{MR}$ (Section 3.3.1) that employs guided expansion and early

Datasets	Candidate Matches		Confirmed Matches
	EM_{VC}^{Opt}	EM_{MR}^{Opt}	
Google	24500	11760	1620
DBpedia	22615	15380	1357
Synthetic	20000	11000	1000

Table 3.2: Candidate matches vs. confirmed matches

termination for subgraph isomorphism checking.

(2) Compared with EM_{MR}, EM_{MR}^{Opt} is at least 3.2, 2.9 and 3 times faster on Google, DBpedia and Synthetic, respectively. These verify the effectiveness of our optimization strategies: on average, (a) L is reduced 52%, 38% and 45%, (b) G^d is 2.5, 1.7 and 2.1 times smaller; and (c) it reduces 23%, 15% and 20% of redundant subgraph isomorphism checking *in each MapReduce round* by leveraging dependency and incremental checking, on the three datasets, respectively.

(3) Compared with EM_{VC}, EM_{VC}^{Opt} is 1.5 times faster on average when $k = 4$ on Google; similarly for DBpedia and Synthetic. These verify the effectiveness of bounded messages and prioritized message propagation (Section 3.4.2).

Table 3.2 shows the numbers of candidate and confirmed matches checked by EM_{VC}^{Opt} and EM_{MR}^{Opt} in the three datasets.

Exp-2: Varying $|G|$. Fixing $p = 4$, $c = 2$ and $d = 2$, we varied $|G|$ with scale factors from 0.2 to 1 for Google, DBpedia and Synthetic. As shown in Figures 3.10(d), 3.10(e) and 3.10(f), (1) all the algorithms take longer on larger $|G|$, as expected; (2) EM_{VC}^{Opt} performs the best among all of them, and EM_{MR}^{Opt} outperforms the other MapReduce algorithms; these are consistent with the results of Exp-1; (3) for product graphs G_p used by EM_{VC} and EM_{VC}^{Opt}, $|G_p| = 2.7 * |G|$ on average, which is much smaller than $|G|^2$; and (4) EM_{MR}^{Opt} and EM_{VC}^{Opt} are reasonably efficient: when $G = (40M, 200M)$ for Synthetic, they take 68 and 3.6 seconds respectively, with 4 processors; the results are similar on Google and DBpedia.

Exp-3: Varying Σ. Finally, we evaluated the impact of Σ, by varying the longest chain c and maximum radius d in Σ.

Varying c. Fixing $p = 4$ and $d = 2$, we varied c from 1 to 5. As shown in

(a) Varying c (Google)

(b) Varying d (Google)

(c) Varying c (DBpedia)

(d) Varying d (DBpedia)

(e) Varying c (Synthetic)

(f) Varying d (Synthetic)
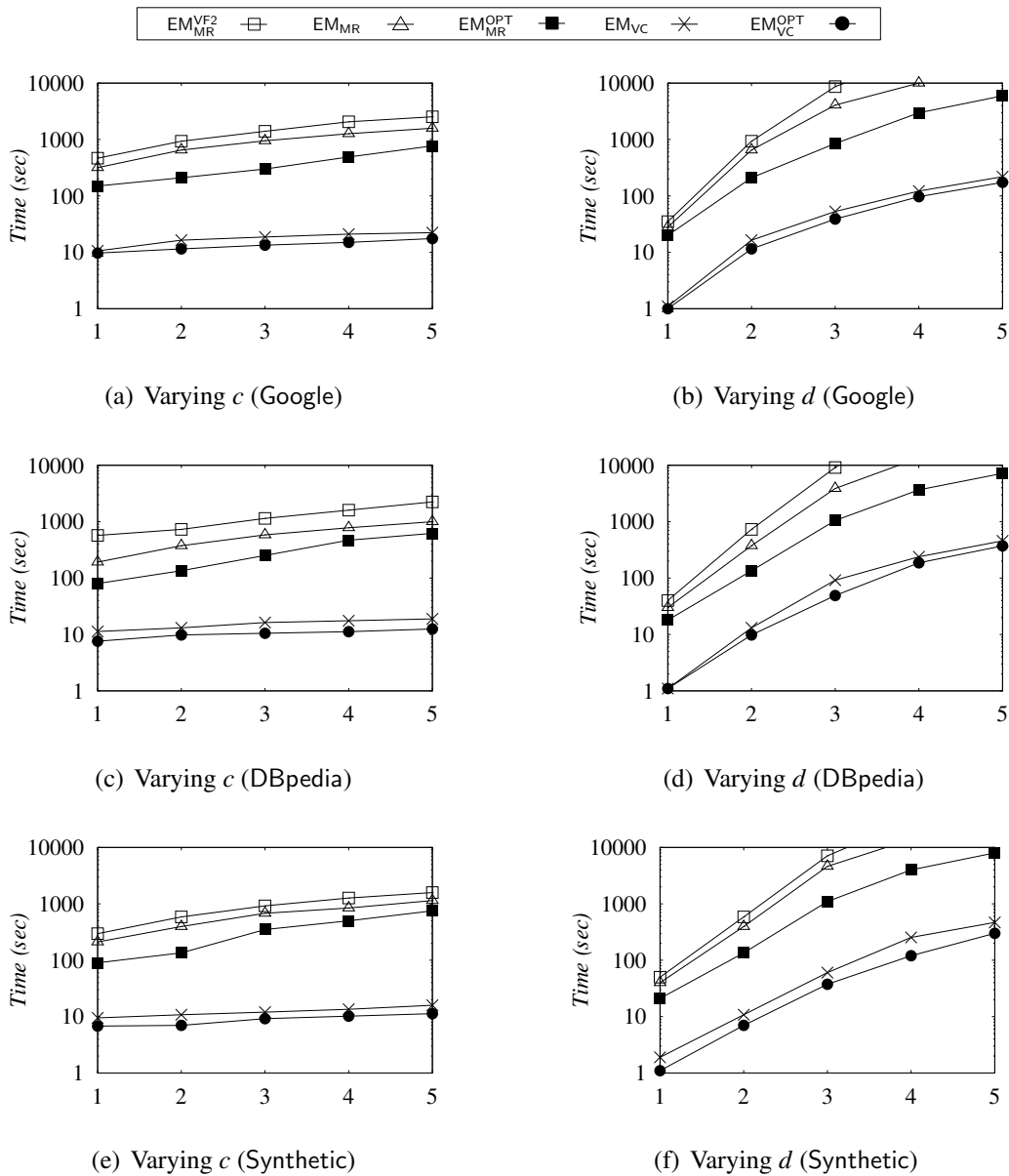
Figure 3.11: Efficiency of entity matching

Figures 3.11(a), 3.11(c) and 3.11(e) for Google, DBpedia and Synthetic ($|G| = (100M, 500M)$), respectively, (1) all the algorithms take longer on larger c, (2) the number of MapReduce rounds increases from 2 to 9, for all MapReduce algorithms; and (3) EM_{VC} and EM_{VC}^{Opt} are less sensitive to c; this is because by asynchronous message passing, these algorithms do not separate computation into "rounds" and avoid the "blocking" of stragglers in each MapReduce round.

Varying d.   Fixing $p = 4$ and $c = 2$, we varied d from 1 to 5. As reported in Figures 3.11(b), 3.11(d) and 3.11(f) for Google, DBpedia and Synthetic

$(|G| = (100M, 500M))$, respectively, (1) d is a major factor for the costs: all the algorithms take longer on larger d; and (2) the pairing strategy is effective as the d-neighbors of EM_{MR}^{Opt} are 60%, 42%, 53% smaller than those of EM_{MR}, and it makes EM_{MR}^{Opt} 4.8, 3.7 and 4.2 times faster than EM_{MR} on average, when $d = 3$, on the three graphs, respectively. We find that keys often have a small radius in real life. This is analogous to real-life SPARQL queries: 98% of them have radius 1, and 1.8% have radius 2 [GFMPdlF11].

Summary. We find the following. (1) Our algorithms scale well with the increase of processors: EM_{MR}, EM_{VC}, EM_{MR}^{Opt} and EM_{VC}^{Opt} are $4.8, 4.8, 4.7$ and 4.9 times faster on average when p increases from 4 to 20. (2) Our algorithms perform well on large graphs and complex Σ: on graphs with $G = (100M, 500M)$, Σ with 500 keys, $c = 2$, $d = 2$, EM_{MR}^{Opt} and EM_{VC}^{Opt} take 27 and 1.5 seconds on average with 20 processors, respectively. (3) Our optimization techniques are effective: EM_{MR}^{Opt} and EM_{VC}^{Opt} are 3 and 1.5 times faster than EM_{MR} and EM_{VC} on average, and EM_{MR}^{Opt} is 4.8 times faster than EM_{MR}^{VF2}. (4) EM_{VC} and EM_{VC}^{Opt} perform better than EM_{MR} and EM_{MR}^{Opt} by reducing unnecessary costs inherent to MapReduce.

Chapter 4

Catching Numeric Inconsistencies in Graphs

Numeric inconsistencies are common in real-life knowledge bases and social networks. To catch such errors, in this chapter, we propose to extend graph functional dependencies with linear arithmetic expressions and comparison predicates, referred to as NGDs.

Example 4.1: Consider the following inconsistencies taken from real-life knowledge bases and social graphs.

(1) Yago. It is recorded that an institute BBC Trust was created in 2007 but destroyed in 1946, as shown in graph G_1 of Fig. 4.1. To detect this, we need to check whether wasDestroyedOnDate - wasCreatedOndate $\geq c$ for a constant c. However, neither arithmetic operator $-$ nor comparison predicate $\geq$ is supported by existing proposals for graph dependencies.

(2) Yago. A village Bhonpur in India is claimed to have 600 females and 722 males, but its total population is 1572 (see G_2 of Fig. 4.1). To catch this, we need an arithmetic equation femalePopulation $+$ malePopulation = populationTotal.

(3) DBpedia. There are two cities, Corona and Downey, in California. Based on the 2014 population census, it is known that Corona has a larger population than Downey. However, Downey is ranked ahead of Corona in population (11th vs. 33rd; see G_3 of Fig. 4.1). The inconsistency should be checked by using a condition that x.population $<$ y.population implies x.populationRank $>$ y.populationRank, where x and y denote places.

(4) Twitter. Suppose that two accounts refer to the same company. If the two substan-
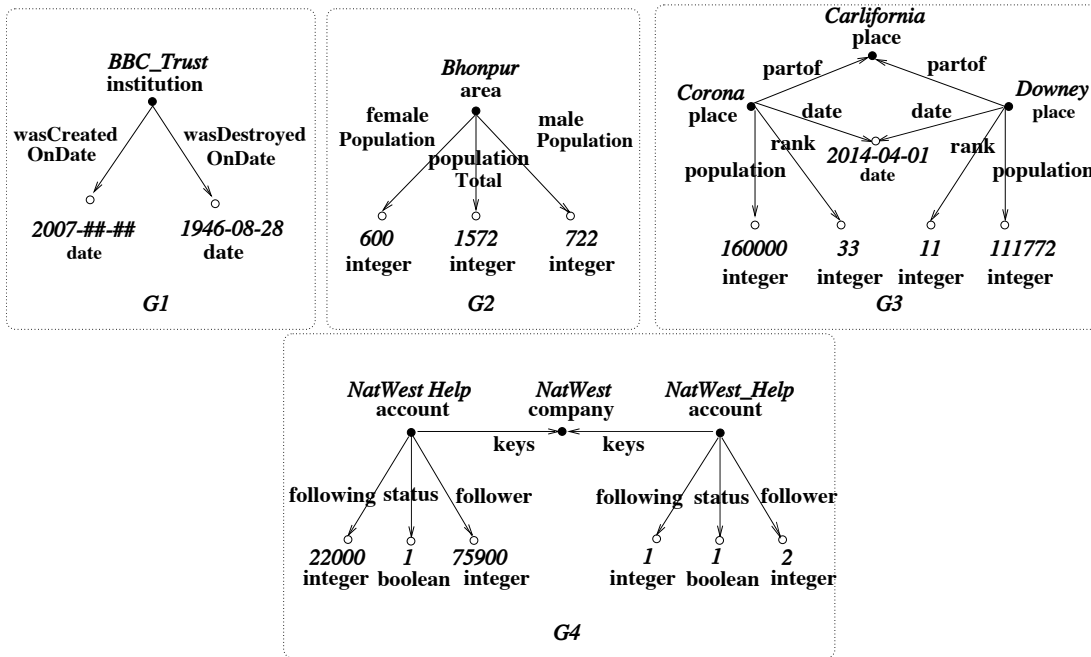
Figure 4.1: Numeric inconsistencies in real-life graphs

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

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

We contend that NGDs and the algorithms developed in this chapter yield a promising tool for catching semantic inconsistencies in graphs, *numeric or not.*

4.1   Preliminaries

We first review basic notations that are needed for defining NGDs. Assume an alphabet Γ of (node and edge) labels.

Graphs. We consider directed *graphs* $G = (V, E, L, F_A)$, where (1) V is a finite set of
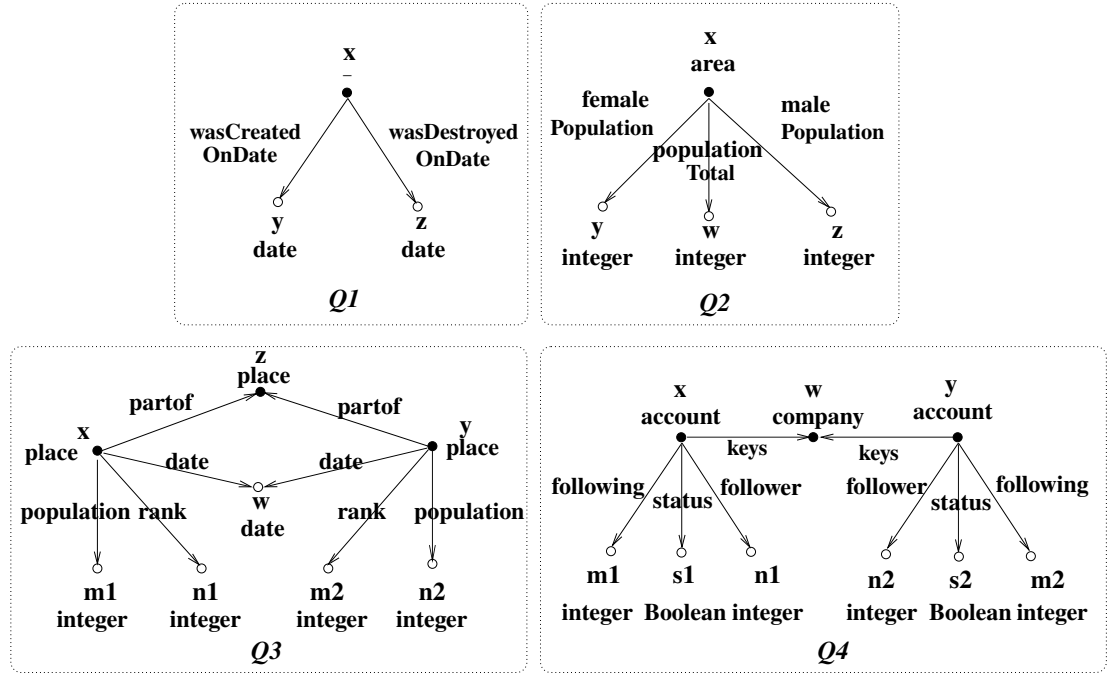
Figure 4.2: Graph patterns

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

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

A subgraph G' is *induced by* a set V' of nodes if $V' \subseteq V$ and E' consists of all the edges in E whose endpoints are both in V'.

Graph patterns. A *graph pattern* is a directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (resp. E_Q) is a set of pattern nodes (resp. edges), (2) L_Q is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) in Γ to each pattern node $u \in V_Q$ (resp. edge $e \in E_Q$), (3) $\bar{x}$ is a list of distinct variables; and (4) μ is a bijective mapping from $\bar{x}$ to V_Q, *i.e.,* it assigns a distinct variable to each node v in V_Q.

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

Example 4.2: Four graph patterns are shown in Fig. 4.2. Here Q_1 depicts an

entity x connected to date y and z with edges labeled wasCreatedOndate and wasDestroyedOndate, respectively. Node x is labeled '_', denoting arbitrary entities regardless of their labels. In G_1 of Fig.4.1, x is mapped to BBC_Trust. Similarly, Q_2–Q_4 can be interpreted by referencing their counterparts in Fig. 4.1.   □

Pattern matching. We adopt the homomorphism semantics of matching following [FL17, ACP10, CP12]. A *match* of pattern $Q[\bar{x}]$ in graph G is a mapping h from Q to G such that (a) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (b) for each $e = (u, u')$ in Q, $e' = (h(u), h(u'))$ is an edge in G and $L_Q(e) = L(e')$. Here $L_Q(u) = L(h(u))$ if $L_Q(u)$ is '_', *i.e.,* wildcard matches any label to indicate generic entities.

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

4.2   Numeric Graph Dependencies

We extend the GFDs of [FWX16, FL17] to incorporate arithmetic expressions and built-in predicates. We start with basic notations.

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

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

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

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

For instance, all the arithmetic expressions given in Example 4.1 are linear. As will be seen in Section 4.3, we adopt linear e to strike a balance between the expressive power and complexity.

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

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

    ○ $Q[\bar{x}]$ is a graph pattern, called the *pattern* of φ; and

    ○ X and Y are (possibly empty) sets of of literals of $Q[\bar{x}]$.

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

NGDs extend GFDs of [FWX16, FL17] by supporting

(a) linear arithmetic expressions with $+, -, \times, \div$ and $|\cdot|$, and

(b) comparisons with built-in predicates $=, \neq, <, \leq, >, \geq$.

In other words, GFDs of [FWX16, FL17] are a special case of NGDs when literals are restricted to terms connected with equality '=' only, *i.e.,* literals of the form $x.A = c$ or $x.A = x.B$.

Example 4.3: To catch those errors spotted in Example 4.1, we define the following NGDs, in terms of the patterns depicted in Fig. 4.2.

(1) Yago. NGD $\varphi_1 = Q_1[x, y, z](\emptyset \rightarrow z.\text{val} - y.\text{val} \geq c)$. Here X is empty set $\emptyset$ and Y includes a single literal. From Q_1 of Fig. 4.2, we can see that x, y and z denote an entity, the date when it was created and the date when it was destroyed, respectively; val is an attribute for the integer values of y and z in days (not shown in Q_1); and c is a constant integer. It states that an entity cannot be destroyed within c days of its creation. It catches the error in G_1 of Fig. 4.1.

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

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

(4) Twitter. NGD $\varphi_4 = Q_4[\bar{x}](\{s_1.\text{val} = 1, a * (m_1.\text{val} - m_2.\text{val}) + b * (n_1.\text{val} - n_2.\text{val}) > c\} \rightarrow s_2.\text{val} = 0)$. Here $\bar{x}$ includes variables x and y referring to two accounts about the same company w, where x (resp. y) has $n_1.\text{val}$ (resp. $n_2.\text{val}$) followers and is following $m_1.\text{val}$ (resp. $m_2.\text{val}$) people; and x (resp. y) has status $s_1.\text{val}$ (resp. $s_2.\text{val}$) indicating whether account x (resp. y) is real. Integers a and b specify the weights of following

and followers, respectively; and c is the threshold for their difference (see Example 4.1). This NGD states that if the gap between the followers and followings of a real account x and account y exceeds c, then the chances are that y is fake. It catches NatWest_Help in G_4 as a fake account.   □

As shown in [FL17], GFDs can express (a) conditional functional dependencies (CFDs [FGJK08]) and (b) equality generating dependencies (EGDs) [AHV95] when relational tuples are represented as vertices in a graph. Since NGDs subsume GFDs, NGDs can also express CFD and EGDs. In particular, NGDs support constant bindings of CFDs [FGJK08], which have proven useful in detecting errors in relations [FG12]. Hence, NGDs can catch non-numeric inconsistencies that GFDs and CFDs can detect, in addition to numeric errors.

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

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

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

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

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

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

Example 4.4: Consider G_1 of Fig. 4.1 and NGD φ_1 of Example 4.3. Then $G_1 \not\models \varphi_1$, since there exists a match $h(x,y,z)$: $x \mapsto$ BBC_Trust, $y \mapsto$ 2007-#-# and $z \mapsto$ 1946-08-28, such that $h(y).\mathsf{val} > h(z).\mathsf{val}$, *i.e.,* $h(x,y,z) \not\models Y$. That is, $h(x,y,z)$ denotes entities that make a *violation* of φ_1 in G_1. Similarly, $G_2 \not\models \varphi_2$, $G_3 \not\models \varphi_3$ and $G_4 \not\models \varphi_4$.   □

symbols	notations
G	graph (V, E, L, F_A)
$Q[\bar{x}]$	graph pattern (V_Q, E_Q, L_Q, μ)
φ, Σ	NGD $\varphi = Q[\bar{x}](X \to Y)$, Σ is a set of NGDs
$h(\bar{x}) \models X \to Y$	a match $h(\bar{x})$ of Q satisfies $X \to Y$
$G \models \Sigma$	graph G satisfies a set Σ of NGDs
$\Sigma \models \varphi$	Σ implies another NGD φ

Table 4.1: Notations in Chapter 4

The notations adopted in this chapter are summarized in Table 4.1.

4.3 Fundamental Problems for NGDs

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

Problems. We first state the two problems.

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

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

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

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

There exist NGDs that are satisfiable when they are taken separately; however, when put together, they are not, *i.e.,* there exist no nonempty graph G that satisfies all

of them.

Example 4.5: Consider a set Σ consisting of two NGDs: $\varphi_5 = Q[x](\emptyset \rightarrow x.A = 6 \wedge x.B = 6)$ and $\varphi_6 = Q[x](\emptyset \rightarrow x.A + x.B = 11)$, where Q has a single node x labeled '_'. One can verify that there exist graphs that satisfy φ_5 and φ_6 when taken separately. However, they are not satisfiable when put together as Σ. Indeed, the values of attributes A and B on each node must be 6 as required by φ_5, while their sum is required to be 11 by φ_6, which is impossible.

As another example, consider NGDs $\varphi_7 = Q[x](x.A \leq 3 \rightarrow x.B > 6)$, $\varphi_8 = Q[x](x.A > 3 \rightarrow x.B > 6)$, and $\varphi_9 = Q[x](\emptyset \rightarrow x.B < 6 \wedge x.A \neq 0)$. Then there exists no graph that satisfies all these three NGDs simultaneously. For if such a graph exists, then each node has attribute B with value less than 6 by φ_9, while by φ_7 and φ_8, it must take a B-attribute of value larger than 6. $\square$

These show that the presence of either linear arithmetic expressions or built-in comparison predicates beyond equality makes the satisfiability analysis more intriguing than that of GFDs [FWX16, FL17].

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

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

As remarked in Section 1.2, the practical need for studying these problems is evident, besides theoretical interest, for determining whether data quality rules discovered from possibly dirty data are sensible, and for optimizing data quality rules, among other things.

Complexity. We next settle the complexity of these problems. The proofs of the results below are quite involved.

Recall that the satisfiability problem for relational functional dependencies (FDs) is trivial, *i.e.,* for any set Σ of FDs over a relation schema R, there always exists a nonempty database instance of R that satisfies Σ [FG12]. Moreover, the implication problem for FDs is in linear-time (cf. [AHV95]). It is known that the satisfiability and implication problems for GFDs are coNP-complete and NP-complete [FWX16], respectively. These are comparable to their counterparts for relational CFDs, which are NP-complete and coNP-complete, respectively [FGJK08]. In contrast, NGDs make our

lives harder.

Theorem 4.1: *For* NGDs*, (a) the satisfiability problem and strong satisfiability problems are both Σ_2^p-complete, and (b) the implication problem is Π_2^p-complete.* ☐

Here Σ_2^p is the class of decision problems that are solvable in NP by calling an NP oracle, *i.e.,* $\Sigma_2^p = \mathsf{NP}^{\mathsf{NP}}$. It is considered more intriguing than NP unless P = NP. Similarly, $\Pi_2^p = \mathsf{coNP}^{\mathsf{NP}}$, which is also above NP in the polynomial hierarchy (see [Pap94] for details).

Proof: We show that the satisfiability, strong satisfiability and implication problems are Σ_2^p-complete, Σ_2^p-complete and Π_2^p-complete for NGDs, respectively, one by one.

The satisfiability problem for NGDs. The proof is involved. We first show that the satisfiability problem for NGDs has a small model property, based on which we then give an Σ_2^p algorithm to check whether a set Σ of NGDs is satisfiable. We prove that the problem is Σ_2^p-hard at last.

The small model property. We show that if a set Σ of NGDs is satisfiable, then Σ has a model G_Σ of size at most $3(|\Sigma|+1)^5$, *i.e.,* a graph G_Σ such that $G_\Sigma \models \Sigma$, $|G_\Sigma| \leq 3(|\Sigma|+1)^5$ and Q has a match in G_Σ for some $\varphi = Q[\bar{x}](X \to Y)$ in Σ.

By the definition of satisfiability, for any satisfiable set Σ, there exists a graph $G = (V, E, L, F_A)$ such that $G \models \Sigma$ and Q has a match h_φ in G for some NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ. Based on match h_φ, we construct graph G_Σ in two steps. (a) We first deduce a subgraph G_φ of G such that G_φ has at most $|\Sigma|$ nodes, by using the topological structure derived from h_φ. (b) We then revise the attribute values in G_φ to obtain the model G_Σ of Σ such that $|G_\Sigma| \leq 3(|\Sigma|+1)^5$, in which each attribute value is of bounded length.

(a) We deduce G_φ as the subgraph of G "induced" by match h_φ, which includes those nodes and edges that are mapped from Q. That is, $G_\varphi = (V_\varphi, E_\varphi, L_\varphi, F_A^\varphi)$, where

- $V_\varphi = \{h_\varphi(x) \mid x \in \bar{x}\}$, where $\bar{x}$ refers to the list of distinct variables in NGD $\varphi = Q[\bar{x}](X \to Y)$;
- $E_\varphi = \{(h_\varphi(x_1), h_\varphi(x_2)) \mid (\mu(x_1), \mu(x_2)) \in E_Q\}$, where E_Q is the set of edges in pattern $Q[\bar{x}]$;
- L_φ is such defined that $L_\varphi(v) = L(v)$ for $v \in V_\varphi$, and $L_\varphi(e) = L(e)$ for $e \in E_\varphi$; and
- we define F_A^φ by taking attributes that only appear in Σ; more specifically, for

each NGD $\varphi' = Q'[\bar{x}'](X' \to Y')$ in Σ, match $h_{\varphi'}$ of Q' in G_φ, and integer variable $x'.A$ that appears in the literals of X', $F_A^\varphi(v').A = F_A(v').A$, where $v' = h_{\varphi'}(x')$; moreover, if $h_{\varphi'}(\bar{x}') \models X'$, then $F_A^\varphi(v').A = F_A(v').A$ for each integer variable $x'.A$ that appears in Y', where $v' = h_{\varphi'}(x')$.

This is well-defined since $G \models \Sigma$. From this we have that $G_\varphi \models \Sigma$, $|V_\varphi| \leq |\Sigma|$, and each node in G_φ has at most $|\Sigma|$ attributes. Note that the labels and attribute values in G_φ may be of size exponential in $|\Sigma|$ as they are copied from G.

(b) We revise G_φ to get G_Σ. We revise the labels and values of attributes in G_φ to eliminate those unboundedly large ones only. We first replace all labels in G_φ that are not in Σ with a single label l_Σ of size at most $|\Sigma|$ that does not occur in Σ. Indeed, since we only check whether two labels are equal, we can replace all the labels $l_1, \ldots, l_m$ that do not appear in Σ with label l_Σ. Since $l_1, \ldots, l_m$ can only match the wildcard '_' , we can verify that for any pattern Q in Σ, h is a match of Q in G_φ if and only if h is a match of Q in G_Σ. That is, all the matches of pattens from Σ in G_φ remain unchanged after the label replacement by the definition of graph pattern matching. Hence the graph still satisfies Σ.

It remains to revise the values of attributes. The main challenge is to ensure that $G_\Sigma \models \Sigma$ after the values are changed. We "normalize" the attributes by solving an integer linear programming problem $L_\Sigma : D\bar{y} \leq \bar{b}$ constructed from graph G_φ, where D is an integral $m \times n$ coefficient matrix and $\bar{b}$ an integral m-component vector. Denote by $A_1, \ldots, A_n$ the attributes that appear in G_φ. We show that the size of graph G_Σ derived from G_φ by replacing the value of each A_i with a corresponding c_i for $i \in [1,n]$ is at most $3(|\Sigma|+1)^5$ and $G_\Sigma \models \Sigma$, where $(c_1, \ldots, c_n)$ is a feasible solution to L_Σ of length polynomial in $|\Sigma|$. The linear programming instance L_Σ is constructed in three steps: (i) identify the set S of instantiated literals "enforced" on G_φ by Σ; (ii) eliminate the absolute value operator $|\cdot|$ in S; (iii) transform the instantiated literals in S into inequality constraints of the form $e_i \leq b_i$ (see below).

Firstly, a set S of instantiated literals enforced by Σ on G_φ is identified, which includes all instantiated literals that are needed to verify whether $G_\varphi \models \Sigma$. More specifically, for each NGD $\varphi' = Q'[\bar{x}'](X' \to Y')$ in Σ, match $h_{\varphi'}$ of Q' in G_φ, and literal l in X', we add $h_{\varphi'}(l)$ to S, where $h_{\varphi'}(l)$ refers to the instantiated literal of l by substituting $h_{\varphi'}(x)$ for every variable x in l. The instantiated literal $h_{\varphi'}(l)$ is also included in S for each l in Y' when $h_\varphi(\bar{x}') \models X$.

Secondly, to comply with linear programming, we remove absolute value operator

$|\cdot|$ from the instantiated literals in S. For each expression $|e|$ that occurs in S, if e is evaluated to be non-negative, then we replace $|e|$ by e, and add a literal $-e \leq 0$ to S; otherwise, $|e|$ is substituted by $-e$, and a literal $e + 1 \leq 0$ is added to S. Note that the value of expression e can be directly evaluated after identifying S.

Finally, we transform each instantiated literal in S to the required form $e_i \leq b_i$ of inequality constraints. To this end, we aim to eliminate built-in operators $=, \neq, <, >, \geq$ which are not allowed, while preserving the satisfiability of the comparison conditions enforced by instantiated literals on G_φ. That is, the corresponding inequality constraint of each instantiated literal $h(l)$ in S is decided by whether $G_\varphi \models h(l)$. For instance, consider an instantiated literal $h(l) = (v.A \leq 3)$. Then it is transformed to $-v.A + 1 \leq -3$ if $G_\varphi \not\models h(l)$, i.e., $v.A > 3$ in G_φ, and remains unchanged otherwise. Note that G_φ satisfies all inequality constraints after the transformation, while this is consistent with the satisfiability of the original instantiated literals on it.

To simply the discussion, depending on the satisfiability of each instantiated literal $h(l) = e_1 \otimes e_2$ on G_φ, i.e., whether $G_\varphi \models h(l)$, we first transform $h(l)$ into the form of $e_1' \leq e_2'$, where e_2' is not enforced to be an integer, as follows.

$h(l)$	$G_\varphi \models h(l)$	$G_\varphi \not\models h(l)$
$e_1 = e_2$	$e_1 \leq e_2$ and $e_2 \leq e_1$	$e_1 + 1 \leq e_2$ if $e_1 < e_2$, otherwise $e_2 + 1 \leq e_1$
$e_1 \neq e_2$	$e_1 + 1 \leq e_2$ if $e_1 < e_2$, otherwise $e_2 + 1 \leq e_1$	$e_1 \leq e_2$ and $e_2 \leq e_1$
$e_1 < e_2$	$e_1 + 1 \leq e_2$	$e_2 \leq e_1$
$e_1 \leq e_2$	$e_1 \leq e_2$	$e_2 + 1 \leq e_1$
$e_1 > e_2$	$e_2 + 1 \leq e_1$	$e_1 \leq e_2$
$e_1 \geq e_2$	$e_2 \leq e_1$	$e_1 + 1 \leq e_2$

Now all instantiated literals in S are in the form of $e_1 \leq e_2$. We then transform them into the required form of $e_i \leq b_i$ by using standard arithmetic transformations to move constant to the right-hand-side and remove division operator $\div$. This completes the construction of S.

The set of instantiated literals in S can be regarded as a linear integer programming problem instance $L_\Sigma : D\bar{y} \leq \bar{b}$ after the transformation above. Here variables $y_1, \ldots, y_n$ in $\bar{y}$ correspond to the attributes in G_φ, i.e., A_i, for each $i \in [1, n]$. One can verify that the satisfiability of the comparison conditions enforced by the original literals in

S before transformation are preserved in L_Σ on G_φ. We construct graph G_Σ from G_φ by using some feasible solution $(c_1,\ldots,c_n)$ to L_Σ of bounded length polynomial in $|\Sigma|$. More specifically, the value of each A_i in G_φ is normalized with the answer to the corresponding variable expression of A_i in L_Σ, *i.e.*, the instantiation of A_i for $i \in [1,n]$ with $(c_1,\ldots,c_n)$.

We now prove the existence of such solutions to L_Σ that are of size polynomial in $|\Sigma|$. It is known that if linear programming $\mathsf{L}_\Sigma : D\bar{y} \le \bar{b}$ has an n-component integral solution, then it has one solution $(c_1,\ldots,c_n)$ with $c_i \le (n+1)M$ for each $i \in [1,n]$, where M refers to the maximal absolute value of the determinants of the square sub-matrices of $[D,\bar{b}]$, and $[D,\bar{b}]$ denotes the augmentation matrix of $D\bar{y} \le \bar{b}$ [CGST86]. One can verify that $M \le (2^{|\Sigma|}(|\Sigma|^2 + 1))^{|\Sigma|^2+1}$ since the number of variables in L_Σ is at most $|\Sigma|^2$, *i.e.*, $n \le |\Sigma|^2$, and each value in $[D,\bar{b}]$ is no larger than $2^{|\Sigma|}$. Moreover, the attribute values in G_φ constitute a feasible solution to L_Σ by the definition of L_Σ. Hence such solution $(c_1,\ldots,c_n)$ of bounded size always exists, in which each $\|c_i\| \le \log_2(2^{|\Sigma|^3+|\Sigma|}(|\Sigma|^2 + 1)^{|\Sigma|^2+2}) \le 3(|\Sigma| + 1)^3$ for $i \in [1,n]$, and $\|c_i\|$ refers to the size of integer c_i.

We next show that G_Σ is a model of bounded size.

We prove that G_Σ is a model of Σ by contradiction. Suppose that $G_\Sigma \not\models \Sigma$. Then there exist some NGD $\varphi' = Q'[\bar{x}'](X' \to Y')$ and match $h_{\varphi'}$ of Q' in G_Σ such that $h_{\varphi'} \models X'$ and $h_{\varphi'} \not\models Y'$. By the definition of G_Σ, $h_{\varphi'}$ is also a match of Q' in G_φ. Suppose that $h_{\varphi'} \models X'$ in G_φ. Then $h_{\varphi'} \models Y'$ and the instantiated $h_{\varphi'}(l)$ also exists in set S for any literal l in Y'. Since $(c_1,\ldots,c_n)$ is a feasible solution to L_Σ and the satisfiability of comparison conditions enforced by original literals in S are preserved in L_Σ, it also satisfies all the corresponding instantiated literals in Y'. Hence $h_{\varphi'} \models Y'$ in G_Σ, a contradiction. One might think that it is possible $h_{\varphi'} \not\models X'$ in G_φ, since G_Σ and G_φ carry different attribute values. In this case, there exist some literal $l = e_1 \otimes e_2$ in X' such that $h_{\varphi'} \not\models l$ in G_φ. We assume *w.l.o.g.* that l is in the form of $e_1 < e_2$; the other cases can be proved similarly. By the definition of L_Σ, there exists a corresponding expression $e_2 \le e_1$ in L_Σ. As $(c_1,\ldots,c_n)$ is a feasible solution to L_Σ, $e_2 \le e_1$ is satisfied. Then $h_{\varphi'} \models e_2 \le e_1$ in G_Σ follows, which contradicts to assumption that $h_{\varphi'} \models X'$ in G_Σ.

Finally, we show that $|G_\Sigma| \le 3(|\Sigma| + 1)^5$. To see this, observe the following. (i) G_Σ has at most $|\Sigma|$ many nodes and edges, since they are inherited from their counterparts in G_φ. (ii) There are at most $|\Sigma|$ many labels whose size is bounded by $|\Sigma|^2$ in G_Σ, as

unboundedly large ones are replaced by l_Σ of length bounded by $|\Sigma|$. (iii) The total size of attribute values in G_Σ is at most $3|\Sigma|^2(|\Sigma|+1)^3$. Putting these together, the size $|G_\Sigma|$ of G_Σ is at most $3(|\Sigma|+1)^5$.

Upper bound. Based on the small model property, we give an Σ_2^p algorithm to check whether a given set Σ of NGDs is satisfiable. The algorithm works as follows.

(1) Guess a graph $G = (V, E, L, F_A)$ such that $|G| \leq 3(|\Sigma|+1)^5$, guess an NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ, and guess a mapping h_φ from V_Q to V, where V_Q is the set of nodes in Q.

(2) Check whether h_φ is a match of Q in G; if so, continue; otherwise, reject the current guess.

(3) Check whether $G \models \Sigma$; if so, return true; otherwise, reject current guess.

The correctness of the algorithm follows from the small model property. For its complexity, step (2) is in PTIME, step (3) is in coNP (see the proof of Corollary 4.4 to be given shortly). Therefore, the algorithm is in Σ_2^p and so is the satisfiability problem for NGDs.

Lower bound. We show that the satisfiability problem for NGDs is Σ_2^p-hard by reduction from the generalized subset sum problem, denoted by GSSP, which is Σ_2^p-complete [SU02]. GSSP is to decide, given an m-component vector $\bar{u}_1$, an n-component vector $\bar{u}_2$ of integers, and an integer w, whether $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Here $\bar{v}_1$ (resp. $\bar{v}_2$) denotes an m-component (resp. n-component) vector of Boolean values, *i.e.,* 0 or 1, $\bar{u}_1^T$ (resp. $\bar{u}_2^T$) is the transpose of $\bar{u}_1$ (resp. $\bar{u}_2$), and $\bar{u}_1^T \cdot \bar{v}_1$ (resp. $\bar{u}_2^T \cdot \bar{v}_2$) refers to the inner product of $\bar{u}_1$ and $\bar{v}_1$ (resp. $\bar{u}_2$ and $\bar{v}_2$).

Given $\bar{u}_1 = (u_1, \ldots, u_m)$, $\bar{u}_2 = (u_1', \ldots, u_n')$ and w, we construct a set Σ of NGDs such that Σ is satisfiable if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds. We use three NGDs that share the same graph pattern to encode GSSP. The first two are to encode the possible vectors $\bar{v}_1$ and $\bar{v}_2$, respectively, while the third one is to encode the given vectors $\bar{u}_1$ and $\bar{u}_2$, and to check whether $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds. To encode the existential semantic of vector $\bar{v}_1$, we use an NGD to ensure that there are m nodes carrying A-attributes with Boolean values. The encoding of the universal semantic of vector $\bar{v}_2$ is accomplished by using wildcards in the graph pattern to arbitrarily match two nodes with value 0 and 1, respectively, of another attribute B.

Based on these considerations, we define the common graph pattern and the set Σ of three NGDs as follows.

(1) The pattern $Q[x_1, \ldots, x_m, y_0, y_1, z_1, \ldots, z_n, z] = (V_Q, E_Q, L_Q, \mu)$ shared by all NGDs in

Σ is defined as:

- $V_Q = \{v_i \mid i \in [1,m]\} \cup \{v'_0, v'_1, v'_2\} \cup \{v''_i \mid i \in [1,n]\}$;

- $E_Q = \emptyset$;

- $L_Q(v_i) = \tau_i$ for $i \in [1,m]$, $L_Q(v'_0) = \gamma_0$, $L_Q(v'_1) = \gamma_1$, $L_Q(v''_i) = $ '_' for $i \in [1,n]$, $L_Q(v'_2) = \chi$; and

- $\mu(x_i) = v_i$ for $i \in [1,m]$, $\mu(y_0) = v'_0$, $\mu(y_1) = v'_1$, $\mu(z_i) = v''_i$ for $i \in [1,n]$, $\mu(z) = v'_2$.

That is, Q consists of $m+n+3$ isolated nodes, in which n nodes are labeled wildcard '_' that can match any label, and the other $m+3$ nodes carry distinct labels.

(2) The first NGD of Σ encodes the Boolean values of $\bar{v}_1$, and is defined as $\varphi_1 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z](\emptyset \to (|2 \times x_1.A - 1| = 1) \wedge \ldots \wedge (|2 \times x_m.A - 1| = 1))$.

Intuitively, it assures that there are m nodes having A-attributes with Boolean values.

(3) The second NGD of Σ is defined as $\varphi_2 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z](\emptyset \to (y_0.B = 0) \wedge (y_1.B = 1) \wedge (z.C = 1))$, which assures that two distinct nodes carrying distinct Boolean values 0 and 1 for their B-attributes, respectively. It also enforces the value of C-attribute to be 1.

(4) The third NGD φ_3 encodes vectors $\bar{u}_1$ and $\bar{u}_2$; it is defined as $\varphi_3 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z]((|2 \times z_1.B - 1| = 1) \wedge \ldots \wedge (|2 \times z_n.B - 1| = 1) \wedge (A' + B' = w) \to (z.C = 2))$, where $A' = x_1.A \cdot u_1 + \ldots + x_m.A \cdot u_m$, and $B' = z_1.B \cdot u'_1 + \ldots + z_n.B \cdot u'_n$.

Intuitively, φ_3 is also used for checking whether $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. To see this, observe that (a) the Boolean values of instantiated $x_1.A, \ldots, x_m.A$ can be assigned to $\bar{v}_1$ when Σ has a model G, since Q must have a match in G; (b) variables $z_1,\ldots,z_n$ can be mapped to nodes labeled γ_1 or γ_2 in an arbitrary way by the definition of wildcard '_'; moreover, there exist two such nodes having B-attributes with distinct values 0 and 1, respectively, as assured by φ_2; therefore, the instantiated $z_1.B,\ldots,z_n.B$ for all the matches of Q can be regarded as the set of all n-component vectors of Boolean values that encode the universal semantic of $\bar{v}_2$; (c) the instantiated $z.C$ is enforced to be 1 by φ_2 and contradicts to $z.C = 2$ when the condition $A' + B' = w$ in φ_3 holds, which indeed encodes the negation of $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$.

We next prove that Σ is satisfiable if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds for given $\bar{u}_1$ and $\bar{u}_2$.

($\Rightarrow$) First assume that Σ is satisfiable. Then there exists a graph G such that $G \models \Sigma$, and Q has a match h_Q in G. Based on the match h_Q, we define the Boolean vector $\bar{v}_1 = (h_Q(x_1).A, \ldots, h_Q(x_m).A)$. This is well-defined since $G \models \varphi_1$, which ensures that $h_Q(x_i)$ carries A-attribute of Boolean value for each $i \in [1, m]$.

It remains to prove that $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ for the vector $\bar{v}_1$ defined above. Suppose by contradiction that there exists a Boolean vector $\bar{v}_2 = (t'_1, \ldots, t'_n)$ such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$. We show that there must also exists another match h'_Q of Q in G such that $h'_Q \not\models \Sigma$, and hence yields a contradiction to $G \models \Sigma$ above. Here the match h'_Q is constructed from the match h_Q and the Boolean vector $\bar{v}_2$ by mapping each variable z_i to the two distinct nodes $h_Q(y_0)$ and $h_Q(y_1)$ according to the value of t'_i in $\bar{v}_2$ for $i \in [1, n]$. More specifically, h'_Q is such defined that (a) $h'_Q(x_i) = h_Q(x_i)$ for $i \in [1, m]$, (b) $h'_Q(y_0) = h_Q(y_0)$, $h'_Q(y_1) = h_Q(y_1)$, $h'_Q(z) = h_Q(z)$, and (c) $h'_Q(z_i) = h_Q(y_0)$ if t'_i of $\bar{v}_2$ is 0, and otherwise $h'_Q(z_i) = h_Q(y_1)$. One can verify that h'_Q is also a match of Q in G since variable z_i (for $i \in [1, n]$) can be mapped to any node, including $h_Q(y_0)$ and $h_Q(y_1)$, by its associated label of wildcard '_'.

It is easy to see that $h'_Q(z).C = 1$ since $G \models \varphi_2$. Moreover, $h'_Q(z).C = 2$, a contradiction, because (a) $h'_Q(z_i).B (i \in [1, n])$, i.e., $h_Q(y_0).B$ or $h_Q(y_1).B$, is a Boolean value that equals t'_i of $\bar{v}_2$, which is guaranteed by φ_2 and the construction of h'_Q, (b) $h'_Q(A' + B') = w$ holds, i.e., $h'_Q(x_1).A \cdot u_1 + \ldots + h'_Q(x_m).A \cdot u_m + h'_Q(z_1).B \cdot u'_1 + \ldots + h'_Q(z_n).B \cdot u'_n = w$, by the assumption of $\bar{v}_2$, and (c) $G \models \varphi_3$. Therefore, $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds.

($\Leftarrow$) Conversely, assume that $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds. Based on such a vector $\bar{v}_1 = (s'_1, \ldots, s'_m)$, we construct a model G of Σ, and hence show that Σ is satisfiable. We define $G = (V, E, L, F_A)$ as follows:

- $V = \{v_1^M, \ldots, v_m^M, v_0^N, v_1^N, v^T\}$;
- $E = \emptyset$;
- $L(v_i^M) = \tau_i$ for $i \in [1, m]$, $L(v_0^N) = \gamma_0$, $L(v_1^N) = \gamma_1$, $L(v^T) = \chi$; and
- $F_A(v_i^M).A = s'_i$ for $i \in [1, m]$, $F_A(v_0^N).B = 0$, $F_A(v_1^N).B = 1$, and $F_A(v^T).C = 1$.

We next show that G is a model of Σ. Observe the following. (a) By the definition of G, it is easy to see that Q has a match h in G and $h(x_i).A = s'_i$ for $i \in [1, m]$. (b) Since for each node v_i^M ($i \in [1, m]$) labeled τ_i in G, $F_A(v_i^M).A$ is a Boolean value, $F_A(v_0^N).B = 0$, $F_A(v_1^N).B = 1$, and $F_A(v^T).C = 1$ for the nodes labeled γ_0, γ_1 and τ, respectively, we have that $G \models \varphi_1$ and $G \models \varphi_2$. It remains to show that $G \models \varphi_3$. Suppose by contradiction that $G \not\models \varphi_3$. Then there exists a match h_Q of Q in G such that $h_Q \models (|2 \times z_1.B - 1| = 1) \wedge \ldots \wedge (|2 \times z_n.B - 1| = 1) \wedge (A' + B' = w)$. Based on h_Q, we show

that there exists an n-component Boolean vector $\bar{v}_2$ such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$, which contradicts to the assumption that $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. More specifically, $\bar{v}_2 = (t'_1, \ldots, t'_n)$ is such defined that $t'_i = h_Q(z_i).B$ for $i \in [1,n]$. Since $h_Q \models |2 \times z_i.B - 1| = 1$, we know that $h_Q(z_i).B$ is a Boolean value and thus can be assigned to t'_i for $i \in [1,n]$. Moreover, it can be verified that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$ for the $\bar{v}_2$ such defined since $h_Q \models (A' + B' = w)$, hence a contradiction.

The strong satisfiability problem for NGDs. We show that the strong satisfiability problem is also Σ_2^p-complete.

Similar to the proof for the satisfiability problem given above, we first show that the strong satisfiability problem for NGDs has a small model property, based on which we then give an Σ_2^p algorithm to check whether a set Σ of NGDs is strongly satisfiable. After this, we finally prove that the strong satisfiability problem is Σ_2^p-hard.

The small model property. We show that if a set Σ of NGDs is strongly satisfiable, then Σ has a model G_Σ of size no larger than $3(|\Sigma|+1)^5$. That is, there exists a graph G_Σ such that $G_\Sigma \models \Sigma$, $|G_\Sigma| \leq 3(|\Sigma|+1)^5$ and every pattern in Σ has a match in G_Σ. By the definition of strong satisfiability, if Σ is strongly satisfiable, then there exists a graph $G = (V, E, L, F_A)$ such that $G \models \Sigma$ and for each pattern Q in Σ, there exists a match h_Q in G. Based on these matches, we construct the graph G_Σ as follows. (a) We first deduce a subgraph G' of G such that G' has at most $|\Sigma|$ nodes. (b) We then revise the attribute values and the labels in G' to obtain the model G_Σ of Σ such that $|G_\Sigma| \leq 3(|\Sigma|+1)^5$, in which each attribute value is of bounded length.

(a) We deduce G' as the subgraph of G "induced" by the matches of patterns in Σ, *i.e.,* $G' = (V', E', L', F'_A)$, where

- $V' = \{h_Q(\bar{x}) \mid Q[\bar{x}](X \to Y) \in \Sigma\}$, where h_Q is the match of Q in G; note that this step is different from its counterpart for the satisfiability problem above;
- $E' = \{(h_Q(v_1), h_Q(v_2)) \mid (v_1, v_2) \in E_Q, Q[\bar{x}](X \to Y) \in \Sigma\}$, where E_Q is the set of edges in pattern $Q[\bar{x}]$, and h_Q is the match of the Q in G;
- L' is such defined that for $v \in V'$ (resp. $e = (v_1, v_2) \in E'$), $L'(v) = L(v)$ (resp. $L'(e) = L(e)$); and
- we define F'_A by taking attributes that only appear in Σ; more specifically, for each NGD $\varphi' = Q'[\bar{x}'](X' \to Y')$ in Σ, match $h_{\varphi'}$ of Q' in G_φ, and integer variable $x'.A$ that appears in the literals of X', $F_A^\varphi(v').A = F_A(v').A$, where $v' = h_{\varphi'}(x')$; moreover, if $h_{\varphi'}(\bar{x}') \models X'$, then $F_A^\varphi(v').A = F_A(v').A$ for each integer variable $x'.A$ that appears in Y', where $v' = h_{\varphi'}(x')$.

This is well-defined since $G \models \Sigma$. From this we have that $G' \models \Sigma$, $|V'| \leq |\Sigma|$, and each node in G' has at most $|\Sigma|$ attributes. Note that the labels and attribute values in G' may be of size exponential in $|\Sigma|$ as they are copied from G.

(b) Similar to the counterpart in the proof for the satisfiability problem above, we can normalize G' to get G_Σ such that $G_\Sigma \models \Sigma$ and $|G_\Sigma| \leq 3(|\Sigma| + 1)^5$.

Upper bound. Based on the small model property, we give an Σ_2^p algorithm to check whether a given set Σ of NGDs is strongly satisfiable, which works as follows.

(1) Guess a graph $G = (V, E, L, F_A)$ such that $|G| \leq 3(|\Sigma| + 1)^5$, and for each NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ, guess a mapping h_φ from V_Q to V, where V_Q is the set of nodes in Q.

(2) Check whether h_φ is a match of Q in G for each NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ; if so, continue; otherwise, reject the current guess.

(3) Check whether $G \models \Sigma$; if so, return true; otherwise, reject current guess.

The correctness of the algorithm is assured by the small model property. For its complexity, step (2) is in PTIME, step (3) is in coNP (see the proof of Corollary 4.4 to be given shortly). Therefore, the algorithm is in Σ_2^p and so is the strong satisfiability problem for NGDs.

Lower bound. We show that the strong satisfiability problem for NGDs is Σ_2^p-hard also by reduction from GSSP. Given $\bar{u}_1 = (u_1, \ldots, u_m)$, $\bar{u}_2 = (u'_1, \ldots, u'_n)$ and w, we construct a set Σ of NGDs such that Σ is strongly satisfiable if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds. The set Σ of NGDs is the same as its counterpart defined in the proof of the satisfiability problem. Since all the NGDs in Σ have the same pattern Q, one can verify that Σ is satisfiable if and only if it is strongly satisfiable. Thus, the lower bound proof for NGD satisfiability can be directly used to prove the same lower bound for strong satisfiability.

The implication problem for NGDs. We now study the implication problem. Similar to the satisfiability problem, we first establish a small model property, and then use it to prove the upper bound. After these, we show that the implication problem is Π_2^p-hard for NGDs.

The small model property. We prove the following: given a set Σ of NGDs and an NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$, if $\Sigma \not\models \varphi$, then there exists a graph $G_{(\Sigma, \varphi)}$ such that $|G_{(\Sigma, \varphi)}| \leq 3(|\Sigma| + |\varphi| + 1)^5$, $G_{(\Sigma, \varphi)} \models \Sigma$ and $G_{(\Sigma, \varphi)} \not\models \varphi$.

If $\Sigma \not\models \varphi$, then there exists a graph $G = (V, E, L, F_A)$ such that $G \models \Sigma$, but $G \not\models \varphi$.

By $G \not\models \varphi$, there exists a match h of Q in G such that $h(\bar{x}) \models X$, but $h(\bar{x}) \not\models Y$. Based on h, we build $G_{(\Sigma,\varphi)}$ as follows. (1) We first deduce a subgraph G_φ of G from h, such that G_φ has at most $|\varphi|$ nodes. (2) We then normalize the labels and attribute values in G_φ to deduce $G_{(\Sigma,\varphi)}$ such that $|G_{(\Sigma,\varphi)}| \leq 3(|\Sigma|+|\varphi|+1)^5$, *i.e.,* $G_{(\Sigma,\varphi)}$ has bounded size. Moreover, we show that $G_{(\Sigma,\varphi)} \models \Sigma$ and $G_{(\Sigma,\varphi)} \not\models \varphi$. The construction of $G_{(\Sigma,\varphi)}$ is similar to its counterpart given above for the satisfiability problem.

(1) We define G_φ as the subgraph of G "induced" by match h. Denote by $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$ the graph pattern of φ. The graph $G_\varphi = (V_\varphi, E_\varphi, L_\varphi, F_A^\varphi)$ is defined as follows:

- $V_\varphi = \{h(v) \mid v \in V_Q\}$, *i.e.,* it includes those nodes mapped from Q via h;
- $E_\varphi = \{(h(v), h(v')) \mid (v, v') \in E_Q\}$, *i.e.,* it also includes those edges mapped from Q;
- the function L_φ is such defined that $L_\varphi(v) = L(v)$ for each $v \in V_\varphi$, and $L(e) = L(e)$ for each $e \in E_\varphi$;
- F_A^φ is defined in the same way as its counterpart for satisfiability by including attributes that appears in Σ and φ; more specifically, for each NGD $\varphi_1 = Q_1[\bar{x}_1](X_1 \to Y_1)$ in $\Sigma \cup \{\varphi\}$, and match h_{Q_1} of Q_1 in G_φ (if exists), if $h_{Q_1}(\bar{x}_1) \models X_1$, then $F_A^\varphi(v_1).A$ is defined and takes the value of $F_A(v_1).A$ for each $x.A$ that appears in the literals of X_1 and Y_1, where $v_1 = h_{Q_1}(x)$; if $h_{Q_1}(\bar{x}_1) \not\models X_1$, then only attributes that appear in literals of X_1 are defined and take the corresponding values from G.

The graph G_φ is well-defined since $F_A^\varphi(\cdot)$ inherits values from $F_A(\cdot)$. Moreover, one can verify that $|V_\varphi| \leq |\varphi|$, each node in G_φ has at most $|\Sigma| + |\varphi|$ many attributes, and $G_\varphi \models \Sigma$ but $G_\varphi \not\models \varphi$ by the definition of G_φ.

(2) We normalize those unboundedly large labels and attribute values in G_φ to construct $G_{(\Sigma,\varphi)}$ along the same lines as its counterpart in the satisfiability proof. Labels not in Σ are replaced by a single label of size bounded by $|\Sigma| + |\varphi|$, and the value of each attribute in G_φ is replaced by a bounded-length solution to its corresponding variable in the linear programming problem $\mathsf{L}_{(\Sigma,\varphi)}$ constructed from G_φ. The only difference is that besides Σ, we also use instantiated literals enforced by φ on G_φ in constructing $\mathsf{L}_{(\Sigma,\varphi)}$.

We next show that $G_{(\Sigma,\varphi)}$ witnesses $\Sigma \not\models \varphi$, *i.e.,* $G_{(\Sigma,\varphi)} \models \Sigma$ but $G_{(\Sigma,\varphi)} \not\models \varphi$, of size at most $3(|\Sigma| + |\varphi| + 1)^5$.

Indeed, $G_{(\Sigma,\varphi)} \models \Sigma$ can be verified along the same lines as its counterpart in the proof for the satisfiability problem. Thus we just prove that $G_{(\Sigma,\varphi)} \not\models \varphi$ by contradiction. Assume that $G_{(\Sigma,\varphi)} \models \varphi$. Then by the construction of $G_{(\Sigma,\varphi)}$, we have that $h \models X$ and $h \models Y$ in $G_{(\Sigma,\varphi)}$ for the match h that was used in constructing G_φ. Since $h \models X$ while $h \not\models Y$ in G_φ, there exists some literal $l = e_1 \otimes e_2$ in Y such that $h \not\models l$ in G_φ. Moreover, the instantiated literal of l is involved in constructing the linear programming $\mathsf{L}_{(\Sigma,\varphi)}$. We assume *w.l.o.g.* that l is in the form of $e_1 > e_2$; the other cases can be proved similarly. By the definition of $\mathsf{L}_{(\Sigma,\varphi)}$, there exists a corresponding expression $e_1 \leq e_2$ in $\mathsf{L}_{(\Sigma,\varphi)}$, which is also satisfied by any feasible solution to $\mathsf{L}_{\Sigma,\varphi}$. It follows that $h \not\models e_2 > e_1$ in $G_{(\Sigma,\varphi)}$ as the value normalization based on solving $\mathsf{L}_{(\Sigma,\varphi)}$ preserves the comparison conditions enforced by any original instantiated literal in G_φ. It contradicts to the assumption that $h \models Y$ in $G_{(\Sigma,\varphi)}$.

We now show that $|G_{(\Sigma,\varphi)}| \leq 3(|\Sigma|+|\varphi|+1)^5$. Observe the following. (i) $G_{(\Sigma,\varphi)}$ has at most $|\varphi|$ many nodes and edges, since $G_{(\Sigma,\varphi)}$ uses the same sets of nodes and edges of G_φ. (ii) There are at most $|\varphi|$ many labels in $G_{(\Sigma,\varphi)}$; their total size is bounded by $|\varphi|(|\Sigma|+|\varphi|)$ as the ones not in Σ are normalized with a unified label of length bounded by $|\Sigma|+|\varphi|$. (iii) The size of each attribute value in $G_{(\Sigma,\varphi)}$ is at most $3(|\Sigma|+|\varphi|+1)^3$; this can be verified along the same lines as that in the proof of satisfiability, leveraging the property of the bounded-length solution to linear programming. (iv) The total size of attribute values in $G_{(\Sigma,\varphi)}$ is bounded by $3|\varphi|(|\Sigma|+|\varphi|)(|\Sigma|+|\varphi|+1)^3$ since each node carries at most $|\Sigma|+|\varphi|$ attributes. Putting these together, the size $|G_{(\Sigma,\varphi)}|$ of $G_{(\Sigma,\varphi)}$ is at most $3(|\Sigma|+|\varphi|+1)^5$.

Upper bound. Based on the small model property, we develop an Σ_2^p algorithm that given a set Σ of NGDs and an NGD $\varphi = Q[\bar{x}](X \to Y)$, checks whether $\Sigma \not\models \varphi$, as follows.

(1) Guess a graph $G = (V, E, L, F_A)$ such that $|G| \leq 3(|\Sigma|+|\varphi|+1)^5$, and a mapping h_φ from V_Q to V, where V_Q denotes the set of nodes in Q.

(2) Check whether h_φ is a match of Q in G; if so, continue; otherwise, reject current guess.

(3) Check whether $h_\varphi(\bar{x}) \models X$ and $h_\varphi(\bar{x}) \not\models Y$; if so, continue; otherwise, reject current guess.

(4) Check whether $G \models \Sigma$; if so, return true; otherwise, reject current guess.

The correctness of the algorithm is assured by the small model property. For its complexity, step (2) is in PTIME, by the definition of matches. Step (3) is also in

PTIME, since $|X|+|Y| \leq |\varphi|$. Step (4) is in coNP (see the proof of Corollary 4.4 for the validation problem). Thus, the algorithm is in Σ_2^p, and the implication problem for NGDs is in Π_2^p.

<u>*Lower bound*</u>. We show that the implication problem is Π_2^p-hard by reduction from the complement of the GSSP (see GSSP in the proof of the satisfiability problem). Given two integer vectors $\bar{u}_1 = (u_1,\ldots,u_m)$ and $\bar{u}_2 = (u'_1,\ldots,u'_n)$, and another integer w, we construct a set Σ of NGDs and another NGD φ such that $\Sigma \not\models \varphi$ if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. That is, we find a graph G "witnessing" $\Sigma \not\models \varphi$ when the condition in GSSP does not hold.

We borrow some constructions from the lower bound proof for NGD satisfiability. Recall the graph pattern $Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z]$ and the third NGD φ_3 given there. We define $\Sigma = \{\varphi_3\}$, which encodes the two given vectors $\bar{u}_1$ and $\bar{u}_2$ and checks whether $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. The other NGD φ is defined as $\varphi = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z]((|2 \times x_1.A - 1| = 1) \wedge \ldots \wedge (|2 \times x_m.A - 1| = 1) \wedge (y_0.B = 0) \wedge (y_1.B = 1) \rightarrow (z.C = 2))$, to encode the possible vectors $\bar{v}_1$ and $\bar{v}_2$, where the pattern Q of φ is the same as that of φ_3.

Observe that φ ensures that for any match h of Q in some graph G, if $h(x_i).A$ is a Boolean value for $i \in [1,m]$, $h(y_0).B = 0$, and $h(y_1).B = 1$, then $h(z).C$ must be 2. In addition, we can deduce 2^n many matches of Q in G for a given h by changing $h(z_i)(i \in [1,n])$ to $h(y_0)$ or $h(y_1)$ arbitrarily. Moreover, for each such deduced match h', if $h'(A' + B') = w$ holds, i.e., $h'(x_1).A \cdot u_1 + \ldots + h'(x_m).A \cdot u_m + h'(z_1).B \cdot u'_1 + \ldots + h'(z_n).B \cdot u'_n = w$, then $h'(z).C = 2$, i.e., $h(z).C = 2$, as assured by φ_3 of Σ.

Based on these observations, we establish the relationship between the implication problem for the NGDs Σ and φ constructed above and the GSSP. We next show that $\Sigma \not\models \varphi$ if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds.

($\Rightarrow$) First assume that $\Sigma \not\models \varphi$. We show that there exists an m-component vector $\bar{v}_1 = (s'_1,\ldots,s'_m)$ such that $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. By $\Sigma \not\models \varphi$, there exists a graph G such that $G \models \Sigma$ but $G \not\models \varphi$. Since $G \not\models \varphi$, there exists a match h of Q in G such that $h \models (|2 \times x_1.A - 1| = 1) \wedge \ldots \wedge (|2 \times x_m.A - 1| = 1) \wedge (y_0.B = 0) \wedge (y_1.B = 1)$ and $h \not\models z.C = 2$. Based on h, we define the Boolean vector $\bar{v}_1$ such that $s'_i = h(x_i).A$ for $i \in [1,m]$. This is well defined as $h(x_i).A$ has value 0 or 1.

It remains to show that $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Assume by contradiction that there exists a Boolean vector $\bar{v}_2 = (t'_1,\ldots t'_n)$ such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$. Then we

show that $h(z).C$ must be 2, which contradicts to $h \not\models z.C = 2$ as argued above. To see this, it suffices to apply φ_3 of Σ. That is, we construct a match h' of Q in G such that $h'(z) = h(z)$ and $h' \models (|2 \times z_1.B - 1| = 1) \wedge \ldots \wedge (|2 \times z_n.B - 1| = 1) \wedge (A' + B' = w)$. For if it holds, then $h'(z).C = h(z).C = 2$ by $G \models \Sigma$. The match h' is constructed as follows, (a) $h'(x_i) = h(x_i)$ for $i \in [1,m]$; (b) $h'(y_0) = h(y_0)$, $h'(y_1) = h(y_1)$, $h'(z) = h(z)$; and (c) $h'(z_i) = h(y_0)$ when $t_i' = 0$, or $h'(z_i) = h(y_1)$ when $t_i' = 1$ for $i \in [1,n]$. Since z_i's are labeled wildcards that match any label, h' is also a match of Q in G. It is easy to see that $|2 \times h'(z_i).B - 1| = 1$ for $i \in [1,n]$ as $h(y_0).B = 0$ and $h(y_1).B = 1$, and $h'(z_i).B$ takes the value from them. Moreover, one can verify that $h' \models A' + B' = w$ by the construction of h' and the assumption for $\bar{v}_1$ and $\bar{v}_2$ defined above. This leads to both $h(z).C = 2$ and $h(z).C \neq 2$, a contradiction.

($\Leftarrow$) Conversely, assume that $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Let $\bar{v}_1 = (s_1', \ldots, s_m')$ be such a vector. Based on $\bar{v}_1$, we construct a graph G such that $G \models \Sigma$ but $G \not\models \varphi$, i.e., G "witness" $\Sigma \not\models \varphi$. Graph G is almost the same as the pattern $Q = (V_Q, E_Q, L_Q, \mu)$, except that it carries attributes. More specifically, $G = (V, E, L, F_A)$ is defined as follows:

- $V = \{v \mid v \in V_Q, L_Q(v) \neq \text{`\_'}\}$, consisting of those nodes in V_Q that are not labeled wildcard;
- $E = E_Q$, the same set of edges as in Q;
- $L(v) = L_Q(v)$ for each $v \in V$; and
- F_A is such defined that $F_A(v).A = s_i'$ if $L(v) = \tau_i$ for $i \in [1,m]$, $F_A(v).B = 0$ if $L(v) = \gamma_0$, $F_A(v).B = 1$ if $L(v) = \gamma_1$, and $F_A(v).C = 1$ if $L(v) = \chi$.

One can verify that $G \not\models \varphi$ as Q has a match in G, node labeled τ_i in G carries Boolean attribute A, and the only node labeled χ in G is carries B-attribute 1 instead of 2.

It remains to show that $G \models \Sigma$. Assume by contradiction that $G \not\models \Sigma$. Then there exists a match h of Q in G such that $h \models A' + B' = w$ and $h \models |2 \times z_i.B - 1| = 1$ for $i \in [1,n]$. That is, $h(A' + B') = h(x_1).A \cdot u_1 + \ldots + h(x_m).A \cdot u_m + h(z_1).B \cdot u_1' + \ldots + h(z_n).B \cdot u_n' = w$. We now define an n-component Boolean vector $\bar{v}_2 = (h(z_1).B, \ldots, h(z_n).B)$; this is well-defined since $h(z_i).B$'s are Boolean values. By the definition of graph G and match h, we have that $h(x_i).A = s_i'$ for $i \in [1,m]$. Hence one can verify that $h(A' + B') = \bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2$. Thus, $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$, which contradicts to the assumption that $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. $\square$

One might think that the complexity comes from interactions between arithmetic operations and comparison predicates. This is not the case: the lower bounds still hold

when either arithmetic expressions or built-in predicates are present, not necessarily both.

Corollary 4.2: *For* NGDs*, the satisfiability, strong satisfiability and implication problems remain* Σ_2^p*-complete,* Σ_2^p*-complete and* Π_2^p*-complete, respectively, even in the absence of either (a) arithmetic operations, or (b) predicates* $\neq, <, \leq, >, \geq$*.*   □

Proof:(a) We first show that the satisfiability, strong satisfiability and implication problems remain Σ_2^p-complete, Σ_2^p-complete and Π_2^p-complete, respectively, for NGDs without arithmetic operations. For the upper bound, the three algorithms given in the proof of Theorem 4.1 are still in Σ_2^p as it is still in PTIME to check whether (i) a given mapping h is a match of Q in G; and (ii) $h \models X$ and $h \not\models Y$.

The lower bounds for the strong satisfiability and implication problems follow immediately from their counterparts for graph denial constraints (GDCs) without id literals [FL17], which are essentially the same as NGDs in the absence of arithmetic operations. Indeed, the strong satisfiability and implication problems for such GDCs are already shown Σ_2^p-complete and Π_2^p-complete, respectively [FL17].

It remains to show the lower bound for the satisfiability problem for NGDs without arithmetic operations. We prove the Σ_2^p-hardness by reduction from the strong satisfiability problem of GDCs without the id literals, which remains Σ_2^p-complete (see the proof of [FL17]).

Given a set Σ_1 of GDCs without the id literals, we construct a set Σ of NGDs such that Σ_1 is strongly satisfiable if and only if Σ is satisfiable. Suppose that Σ_1 consists of the following n GDCs (without the id literals): $\varphi_1' = Q_1[\bar{x}_1](X_1 \to Y_1)$, ..., $\varphi_n' = Q_n[\bar{x}_n](X_n \to Y_n)$. We construct n NGDs in the absence of arithmetic operations such that these NGDs share the same pattern Q, which is a combination of Q_1, ..., Q_n. Meanwhile, for each $\varphi_i' = Q_i[\bar{x}_i](X_i \to Y_i)$ in Σ_1, Σ includes one NGD such that the constraint $X_i \to Y_i$ is also enforced only on Q_i in Q. More specifically, Σ is constructed as follows.

(1) The graph pattern $Q[\bar{x}_1, \ldots, \bar{x}_n] = (V_Q, E_Q, L_Q, \mu)$ that are shared by all NGDs in Σ is such defined that

  • $V_Q = V_{Q_1} \cup \ldots \cup V_{Q_n}$, where V_{Q_i} is the set of nodes in φ_i' ($i \in [1, n]$);

  • $E_Q = E_{Q_1} \cup \ldots \cup E_{Q_n}$, where E_{Q_i} is the set of edges in φ_i' ($i \in [1, n]$);

  • $L_Q(v) = L_{Q_i}(v)$ if $v \in V_{Q_i}$, where L_{Q_i} is the labeling function in φ_i' ($i \in [1, n]$); and

- $\mu(x_i) = \mu_{Q_i}(v)$ if $v \in V_{Q_i}$, where μ_{Q_i} is the mapping function in φ_i' ($i \in [1,n]$).

  Intuitively, Q is the disjoint union of $Q_1, \ldots, Q_n$, where Q_i and Q_j are disjoint for all $i, j \in [1,n]$ and $i \neq j$.

(2) For each $\varphi_i' = Q_i[\bar{x}_i](X_i \to Y_i)$ in Σ_1, Σ includes NGD $\varphi_i = Q[\bar{x}](X_i \to Y_i)$. Note that since we require that Q_i and Q_j are disjoint for all $i, j \in [1,n]$ and $i \neq j$, $X_i \to Y_i$ is also the constraint on the nodes in Q_i.

We next show that these make a reduction, *i.e.,* Σ_1 is strongly satisfiable if and only if Σ is satisfiable.

($\Rightarrow$) Suppose that Σ_1 is strongly satisfiable, and that G is such a model of Σ_1. Then $G \models \Sigma_1$, and there exists a match of Q_i in G for all $i \in [1,n]$. We prove that G also witnesses the satisfiability of Σ. First observe that because Q contains $Q_1, \ldots, Q_n$, and there exists a match of Q_i in G for any $i \in [1,n]$, we can verify that Q has a match in G.

We next prove that $G \models \Sigma$ by contradiction. If $G \not\models \Sigma$, then there exist an NGD $\varphi_i = Q[\bar{x}](X_i \to Y_i)$ in Σ and a match h of Q in G such that $h(\bar{x}) \models X_i$, but $h(\bar{x}) \not\models Y_i$. By the definition of Σ, there exists a corresponding GDC $\varphi_i' = Q_i[\bar{x}_i](X_i \to Y_i)$ in Σ_1. We will show that $G \not\models \varphi_i'$, which contradicts to $G \models \Sigma_1$. Hence $G \models \Sigma$.

We now prove that $G \not\models \varphi_i'$. By the definition of Q, we can deduce a match h_i of Q_i in G as follows: for each $x \in V_{Q_i}$, $h_i(x) = h(x)$. Here V_{Q_i} is the set of nodes in Q_i. Because X_i and Y_i are literals defined on vertexes in $\bar{x}_i$, from $h(\bar{x}) \models X_i$ and $h(\bar{x}) \not\models Y_i$, we know that $h_i(\bar{x}) \models X_i$ and $h_i(\bar{x}) \not\models Y_i$. That is, $h_i \not\models (X_i \to Y_i)$. Therefore, $G \not\models \varphi_i'$.

($\Leftarrow$) Suppose that Σ is satisfiable, and that G is such a model of Σ. Then $G \models \Sigma$, and there exists a match h of Q in G. We show that G witnesses the strong satisfiability of Σ_1. As argued above, there exists a match of Q_i in G for any $i \in [1,n]$. It remains to show that $G \models \Sigma_1$.

We prove that $G \models \Sigma_1$ by contradiction. If $G \not\models \Sigma_1$, then there exist a GDC $\varphi_i' = Q_i[\bar{x}_i](X_i \to Y_i)$ and a match h_1 of Q_i in G such that $h_1(\bar{x}_i) \models X_i$, but $h_1(\bar{x}_i) \not\models Y_i$. By the definition of Σ, $\varphi_i = Q[\bar{x}](X_i \to Y_i)$ is an NGD in Σ. It suffices to show that $G \not\models \varphi_i$. For if it holds, then it contradicts to the assumption that $G \models \Sigma$.

We now prove that $G \not\models \varphi_i$. By the definition of Q, we can deduce a match h_2 of Q in G as follows: when $x \in V_{Q_i}$, $h_2(x) = h_1(x)$; otherwise, $h_2(x) = h(x)$. Here V_{Q_i} is the set of nodes in Q_i, and h is the match deduced from the assumption that $G \models \Sigma$. Because X_i and Y_i are literals defined on the vertexes in $\bar{x}_i$, from $h_1(\bar{x}_i) \models X_i$ and $h_1(\bar{x}_i) \not\models Y_i$, we know that $h_2(\bar{x}) \models X_i$ and $h_2(\bar{x}) \not\models Y_i$. That is, $h_2 \not\models (X_i \to Y_i)$.

Therefore, $G \not\models \varphi_i$.

(b) We next show that the satisfiability, strong satisfiability, and implication problems are Σ_2^p-complete, Σ_2^p-complete and Π_2^p-complete, respectively, for NGDs in the absence of comparison predicates. To see the lower bound, observe that the encoding used in the lower bound proof of Theorem 4.1 does not use any comparison predicates for NGDs, *i.e.,* the only built-in predicate involved is $=$. The upper bounds can be verified along the same lines as in (a) above. $\square$

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

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

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

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

Theorem 4.3: *The satisfiability, strong satisfiability and implication problems become undecidable for* NGDs *extended with non-linear arithmetic expressions, even when*

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

Proof: We show that the satisfiability, strong satisfiability and implication problems become undecidable for NGDs extended with non-linear arithmetic expressions, respectively, one by one.

Satisfiability. We show that the satisfiability problem becomes undecidable for NGDs extended with non-linear expressions of degree at most 2, even when only built-in predicate $=$ is present. It is verified by reduction from the Hilbert's 10th problem, denoted by HTP, which is undecidable [Mat93, Jon80]. HTP is to decide, given a polynomial Diophantine equation in the form of $\sum\limits_{i=1}^{n} a_i y_1^{n_{1,i}} \ldots y_m^{n_{m,i}} = 0$, where $a_1, \ldots, a_n$ are integer coefficients and $n_{1,i}, \ldots, n_{m,i}$ are non-negative integer exponents for each $i \in [1, n]$, whether there exists a feasible solution of integers for $(y_1, \ldots, y_m)$.

Given a Diophantine equation $\sum\limits_{i=1}^{n} a_i y_1^{n_{1,i}} \ldots y_m^{n_{m,i}} = 0$, we construct a set Σ of extended NGDs such that Σ is satisfiable if and only the Diophantine equation has a solu-

tion of integers. We encode the semantics of polynomials in the Diophantine equation in a recursive manner by using literals of a single NGD with built-in predicate $=$ only, and none of the arithmetic expressions has degree above 2.

We start by illustrating the idea of recursive encoding with an example. Consider a polynomial of $3y_1y_2^5$ to be encoded. We first use a literal with integer variable $x_0^1.A$ to encode the coefficient 3, *i.e.,* $x_0^1.A = 3$. We then encode the exponentiation y_1 and y_2^5. The former is simply expressed by another integer variable $x_{1,1}^1.A$, while the latter is encoded recursively leveraging three literals $x_{2,4}^5.A = x_{2,2}^2.A \times x_{2,3}^3.A$, $x_{2,3}^3.A = x_{2,2}^2.A \times x_{2,1}^1.A$, and $x_{2,2}^2.A = x_{2,1}^1.A \times x_{2,1}^1.A$. That is, we encode y_2^5 by decomposing it into two exponentiation y_2^3 and y_2^2 of smaller exponents, which are also encoded as literals with the associated integer variables $x_{2,3}^3.A$ and $x_{2,2}^2.A$, respectively. Indeed, each exponentiation of y_i^j has a corresponding integer variable $x_{i,k}^j.A$ (if exists) in the encoding. Having processed the coefficient and distinct exponentiation, we finally encode the whole polynomial, also following a recursive strategy. Given a polynomial, we decompose it into a sub-expression followed by an suffix of single exponentiation, and encode these two separately. For instance, we encode $3y_1y_2^5$ with two literals $x_{1,2}'.A = x_{1,1}'.A \times x_{2,4}^5.A$ and $x_{1,1}'.A = x_0^1.A \times x_{1,1}^1.A$, *i.e.,* $3y_1y_2^5$ is split into sub-expression $3y_1$ and suffix y_2^5, which are expressed by $x_{1,1}'.A$ and $x_{2,4}^5.A$, respectively. Putting these together, the $3y_1y_2^5$ is expressed as the integer variable $x_{1,2}'.A$ eventually.

Formally, for each polynomial $a_iy_1^{n_{1,i}} \ldots y_m^{n_{m,i}} (i \in [1,n])$ in the given equation, (1) coefficient a_i is encoded by a single literal $x_0^i = a_i$; (2) each exponentiation $y_j^{n_{j,i}} (j \in [1,m])$ is encoded in terms of the encoding of exponentiation $y_j^{\lfloor n_{j,i}/2 \rfloor}$ and $y_j^{\lceil n_{j,i}/2 \rceil}$ recursively with a literal in the form of $x_{j,k}^{n_{j,i}}.A = x_{j,l}^{\lfloor n_{j,i}/2 \rfloor}.A \times x_{j,p}^{\lceil n_{j,i}/2 \rceil}.A$; and (3) the whole polynomial $a_iy_1^{n_{1,i}} \ldots y_m^{n_{m,i}}$ is recursively encoded through the encoding of sub-expression $a_iy_1^{n_{1,i}} \ldots y_{m-1}^{n_{m-1,i}}$ and exponentiation $y_m^{n_{m,i}}$ with a literal $x_{i,m}'.A = x_{i,m-1}'.A \times x_{m,k}^{n_{m,i}}.A$, where $x_{i,m-1}'.A$ denotes the corresponding integer variable of the sub-expression. One can verify that each exponentiation $y_j^{n_{j,i}}$ can be encoded by using at most $2\lceil \log_2 n_{j,i} \rceil$ literals, and the encoding of a whole polynomial needs m literals. Thus the total size of the encoding is polynomial in that of the given equation, yielding a PTIME reduction.

Based on the recursive encoding, we construct an extended NGD $\varphi = Q[\bar{x}](\emptyset \to (Z_1 \wedge Z_2))$, where Z_1 includes those literals for encoding the polynomials of the given equation as described above, and Z_2 checks the existence of integer solutions to the equation. Let Σ consists of φ only. More specifically, φ is defined as follows.

- The graph pattern $Q[x_0^1, \ldots, x_0^n, x_{1,1}^{N_{1,1}}, \ldots, x_{1,K_i}^{N_{K_i,1}}, \ldots, x_{m,1}^{N_{1,m}}, \ldots, x_{m,K_m}^{N_{K_m,m}}, x_{1,1}', \ldots, x_{1,m}',$

$\ldots, x'_{n,1}, \ldots, x'_{n,m}, x''] = (V_Q, E_Q, L_Q, \mu)$ is such defined that

- $V_Q = \{v_0^i \mid i \in [1,n]\} \cup \{v_{i,j}^{N_{j,i}} \mid i \in [1,m], j \in [1,K_i]\} \cup \{v'_{i,j} \mid i \in [1,n], j \in [1,m]\} \cup \{v''\}$, where K_i refers to the number of variables introduced to encode all the exponentiation of base y_i in the equation for each $i \in [1,m]$, $N_{j,i}$'s ($i \in [1,m], j \in [1,K_i]$) indicate their exponents, *i.e.*, $x_{i,j}^{N_{j,i}}$ is the corresponding integer variable of exponentiation $y_i^{N_{j,i}}$ in the recursive encoding; note that $N_{j,i} < N_{j',i}$ when $j < j'$, and that moreover, the nodes in $Q[\bar{x}]$ are split into four groups to encode the coefficients, the exponentiation, the polynomials, and the equation, respectively;
- $E_Q = \emptyset$, *i.e.*, $Q[\bar{x}]$ consists of isolated nodes;
- L_Q is such defined that nodes in V_Q are associated with distinct labels; and
- for each $i \in [1,n]$, $\mu(x_0^i) = v_0^i$; for each $i \in [1,m]$ and $j \in [1,K_i]$, $\mu(x_{i,j}^{N_{j,i}}) = v_{i,j}^{N_{j,i}}$; for each $i \in [1,n]$ and $j \in [1,m]$, $\mu(x'_{i,j}) = v'_{i,j}$ and $\mu(x'') = v''$, which maps variables from $\bar{x}$ to V_Q;

- Z_1 is the conjunction of all literals introduced for recursively encoding the polynomials of the equation, *e.g.*, $x_0^i.A = a_i$ to encode coefficient, $x_{i,j}^{N_{j,i}}.A = x_{i,l}^{\lfloor N_{j,i}/2 \rfloor}.A \times x_{i,p}^{\lceil N_{j,i}/2 \rceil}.A$ to encode exponentiation, and $x'_{i,j}.A = x'_{i,j-1}.A \times x_{j,k}^{n_{j,i}}.A$ to encode the polynomial; and

- Z_2 is given as $(x''.A = x'_{1,m}.A + x'_{2,m}.A + \ldots + x'_{n,m}.A) \wedge (x''.A = 0)$, *i.e.*, the equation is expressed as $x''.A = 0$.

Observe that φ ensures the instantiation of integer variables, *i.e.*, values of attribute A, must satisfy all the literals enforced by the recursive encoding, and it forms a feasible solution to the given Diophantine equation by the definition of Z_2. Based on this, we next show that Σ is satisfiable if and only if $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \ldots y_m^{n_{m,i}} = 0$ has an integer solution.

($\Rightarrow$) First assume that Σ is satisfiable. Then there exists a graph G such that $G \models \Sigma$ and Q has a match h in Q. We next show that $(h(x_{1,1}^1).A, \ldots, h(x_{m,1}^1).A)$ is a feasible integer solution to the given Diophantine equation $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \ldots y_m^{n_{m,i}} = 0$. Assume by contradiction that $(h(x_{1,1}^1).A, \ldots, h(x_{m,1}^1).A)$ is not a feasible solution to the equation. Then $\sum_{i=1}^{n} a_i (h(x_{1,1}^1).A)^{n_{1,i}} \ldots (h(x_{m,1}^1).A)^{n_{m,i}} \neq 0$. Since $h \models (\emptyset \to (Z_1 \wedge Z_2))$, we have that $h(x'').A = h(x'_{1,m}).A + \ldots + h(x'_{n,m}).A = \sum_{i=1}^{n} a_i (h(x_{1,1}^1).A)^{n_{1,i}} \ldots (h(x_{m,1}^1).A)^{n_{m,i}} = 0$, a contradiction to the assumption. This can be verified by repeatedly substituting $h(x'_{i,j}).A$ by $h(x'_{i,j-1}).A \times h(x_{j,k}^{n_{j,i}}).A$, $h(x_{i,j}^{N_{j,i}}).A$ by $h(x_{i,l}^{\lfloor N_{j,i}/2 \rfloor}).A \times h(x_{i,p}^{\lceil N_{j,i}/2 \rceil}).A$, and

$h(x_0^i.A)$ by a_i, respectively, until only a_i's and $h(x_{j,1}^1).A$'s are left. This is well-defined as h satisfies all the literals in Z_1 that are introduced to recursively encode the computation of polynomials. Thus $(h(x_{1,1}^1).A, \ldots, h(x_{m,1}^1).A)$ makes a solution to the Diophantine equation.

($\Leftarrow$) Conversely, assume that $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \ldots y_m^{n_{m,i}} = 0$ has a solution $(b_1, \ldots, b_m)$ of integers. Based on this solution, we build a graph G such that $G \models \Sigma$ and there exists a match of Q in G. We define graph $G = (V, E, L, F_A)$ as follows:

- $V = V_Q$, $E = E_Q$ and $L = L_Q$, *i.e.,* it takes the same nodes, edges and labels as in the graph pattern Q;

- F_A is such defined that

  - for each $i \in [1, n]$, $F_A(v_0^i).A = a_i$;
  - for each $i \in [1, m]$ and each $j \in [1, K_i]$, $F_A(v_{i,j}^{N_{j,i}}).A = b_i^{N_{j,i}}$;
  - for each $i \in [1, n]$ and each $j \in [1, m]$, $F_A(v_{i,j}').A = a_i b_1^{n_{1,i}} \ldots b_j^{n_{j,i}}$; and
  - $F_A(v'').A = \sum_{i=1}^{n} a_i b_1^{n_{1,i}} \ldots b_m^{n_{m,i}}$.

Intuitively, G is the same as the graph pattern Q except the associated attributes, which are assigned values of the coefficients, exponentiation, polynomials and sum of polynomials in the given equation when variables are instantiated by the feasible solution $(b_1, \ldots, b_m)$. Since G has the same topological structure as that in Q, and all nodes carry distinct labels, we know that there only exists a single match h of Q in G. It remains to show that $h \models (\emptyset \rightarrow Z_1 \cap Z_2)$. Suppose by contradiction that there exists a literal l in Z_1 or Z_2 such that $h \not\models l$. Observe the following. (a) By the definition of F_A, we have that $F_A(v_0^i.A) = a_i$, $F_A(v_{i,j}^{N_{j,i}}).A = F_A(v_{i,l}^{\lfloor N_{j,i}/2 \rfloor}).A \times F_A(v_{i,p}^{\lceil N_{j,i}/2 \rceil}).A$ and $F_A(v_{i,j}').A = F_A(v_{i,j-1}').A \times F_A(v_{j,k}^{n_{j,i}}).A$. Hence h satisfies all the literals in Z_1. (b) Since $F_A(v'').A = F_A(v_{i,m}').A + \ldots + F_A(v_{n,m}')$, we have that $h \models (x''.A = x_{1,m}'.A + x_{2,m}'.A + \ldots + x_{n,m}'.A)$. Hence, the only literal that is not satisfied by h is $x''.A = 0$. As a result, $F_A(v'').A = \sum_{i=1}^{n} a_i b_1^{n_{1,i}} \ldots b_m^{n_{m,i}} \neq 0$, contradicting to the assumption that $(b_1, \ldots, b_m)$ is a feasible solution to the Diophantine equation.

Strong satisfiability. The proof for the satisfiability problem above also suffices to verify the undecidability of the strong satisfiability problem, since the set Σ used in the reduction there consists of a single extended NGD. Hence, the satisfiability and strong satisfiability of Σ coincide.

Implication. We show that the implication problem for extended NGDs is undecidable by reduction from the complement of HTP (see HTP in the proof of the satisfiability

problem). Given a Diophantine equation $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$, we construct a set Σ of extended NGDs, and another extended NGD φ_1 such that $\Sigma \not\models \varphi_1$ if and only if $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$ has an integer solution.

Recall the graph pattern Q and the extended NGD $\varphi = Q[\bar{x}](\emptyset \to (Z_1 \wedge Z_2))$ defined in the proof of the satisfiability problem above. We define $\Sigma = \{\varphi_2\} = \{Q[\bar{x}](\emptyset \to (Z_1 \wedge Z_2'))\}$, where Z_2' is obtained from Z_2 by removing literal $x''.A = 0$, to encode the computation of polynomials in the equation as in the proof for satisfiability, and $\varphi_1 = Q[\bar{x}](x''.A = 0 \to x''.B = 1)$ to check the existence of solutions to the equation. We next show that $\Sigma \not\models \varphi_1$ if and only if $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$ has an integer solution.

($\Rightarrow$) First suppose that $\Sigma \not\models \varphi_1$. We show that there exists an integer solution $(b_1, \dots, b_m)$ to the Diophantine equation $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$. By $\Sigma \not\models \varphi_1$, there exists a graph G such that $G \models \Sigma$ but $G \not\models \varphi_1$. Since $G \not\models \varphi_1$, there exists a match h of Q in G such that $h(x'').A = 0$ and $h(x'').B \neq 1$. Based on h, we define $(b_1, \dots, b_m)$ such that $b_i = h(x_{i,1}^1).A$ for each $i \in [1, m]$. It remains to show that $(b_1, \dots, b_m)$ is a feasible solution to the equation. Assume by contradiction that $(b_1, \dots, b_m)$ is not an integer solution. Then $\sum_{i=1}^{n} a_i (h(x_{1,1}^1).A)^{n_{1,i}} \dots (h(x_{m,1}^1).A)^{n_{m,i}} \neq 0$. Since φ_1 and φ_2 share the same graph pattern Q and $G \models \Sigma$, we have that $h \models (Z_1 \wedge Z_2')$. Therefore, $h(x'').A = h(x_{1,m}').A + \dots + h(x_{n,m}').A$. Moreover, since h satisfies all the literals in Z_1, one can verify that $h(x'').A$ also equals $\sum_{i=1}^{n} a_i (h(x_{1,1}^1).A)^{n_{1,i}} \dots (h(x_{m,1}^1).A)^{n_{m,i}}$ by applying the rules of recursive encoding, *i.e.,* literals in Z_1, to express $h(x'').A$ by a_i's and $h(x_{i,1}^1).A$'s only for $i \in [1,n]$ and $j \in [1,m]$. Then $h(x''.A) \neq 0$ by the assumption that $(b_1, \dots, b_m) = (h(x_{1,1}^1).A, \dots, h(x_{m,1}^1).A)$ is not a feasible solution. It contradicts to that $h(x'').A = 0$ as argued above. Therefore, $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$ has a feasible integer solution of $(h(x_{1,1}^1).A, \dots, h(x_{m,1}^1).A)$.

($\Leftarrow$) Conversely, suppose that $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \dots y_m^{n_{m,i}} = 0$ has an integer solution $(b_1, \dots, b_m)$. We construct a graph G' such that $G' \models \Sigma$, but $G' \not\models \varphi_1$. Recall the graph $G = (V, E, L, F_A')$ constructed in the proof of the satisfiability problem above. Graph $G' = (V, E, L, F_A')$ is the same as G except that an additional B-attribute value 2 is associated with node v'', *i.e.,* $F_A'(v'').B = 2$. Since $G \models \varphi$ (see the proof of the satisfiability problem) and φ_2 is obtained from φ by removing one literal, $G' \models \varphi_2$, *i.e.,* $G' \models \Sigma$. Moreover, by the construction of G', we have that there exists only one match h of Q in G. It remains to show that $G' \not\models \varphi_1$, *i.e.,* $h \not\models \varphi_1$. Assume by contradiction that

$h \models \varphi_1$. Then $h(x'').A \neq 0$ since $h(x'').B = F'_A(v'').B = 2$. However, $h(x'').A$ must be 0 as argued in the proof of the satisfiability problem, hence a contradiction. Therefore, $G' \not\models \varphi_1$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

4.4 Detecting Errors with NGDs

We have seen that NGDs provide uniform rules for capturing inconsistencies in graphs, numeric or not (Section 4.2). We next study error detection in graphs by using NGDs as data quality rules.

4.4.1 Detecting Inconsistencies in Graphs

To state the error detection problem, we borrow the following notations from [FWX16]. Given an NGD $\varphi = Q[\bar{x}](X \to Y)$ and a graph G, we say that a match $h(\bar{x})$ of Q in G is a *violation* of φ if $G_h \not\models \varphi$, where G_h is the subgraph induced by $h(\bar{x})$. For a set Σ of NGDs, we denote by $\mathsf{Vio}(\Sigma, G)$ the set of all violations of NGDs in G, *i.e.,* $h(\bar{x}) \in \mathsf{Vio}(\Sigma, G)$ if there exists an NGD φ in Σ such that $h(\bar{x})$ is a violation of φ in G. That is, $h(\bar{x})$ violates at least one NGD in Σ.

The *error detection problem* is stated as follows.
- ◦ *Input*: A set Σ of NGDs and a graph G.
- ◦ *Output*: The set $\mathsf{Vio}(\Sigma, G)$ of violations.

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

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

It is known that the validation problem for GFDs is coNP-complete [FWX16]. The good news is that the problem gets no harder for NGDs, despite their increased expressive power.

Corollary 4.4: *The validation problem for* NGDs *remains* coNP-*complete.* $\qquad\square$

Proof: Here we only show that the validation problem for NGDs is in coNP. Since NGDs subsume GFDs, and the validation problem for GFDs is coNP-complete, the lower bound follows.

Upper bound. We use the following NP algorithm to check, given a graph G and a set Σ of NGDs, whether $G \not\models \Sigma$.

(1) Guess an NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, and a mapping h from Q to G.

(2) Check whether h is a match of Q in G; if so, continue; otherwise, reject the current guess.

(3) Check whether $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$; if so, return true; otherwise, reject the current guess.

The correctness of the algorithm follows from the semantics of NGDs. For its complexity, step (2) is in PTIME. Step (3) is also in PTIME since $|X| + |Y| \leq |\Sigma|$. Thus the algorithm is in NP, and the validation problem is in coNP.   □

Using GFDs as data quality rules, parallel algorithms have been developed for error detection [FWX16]. The algorithms are *parallel scalable*, *i.e.,* they guarantee to reduce the running time of a yardstick sequential algorithm when more processors are used (see Section 4.5.1). Hence the algorithms can scale with real-life graphs by adding resources when the graphs grow big. The experimental study of [FWX16] has validated the parallel scalability and efficiency of the algorithms.

A close examination of the algorithms of [FWX16] reveals that the algorithms can be readily extended to NGDs. Indeed, for the algorithms to work with NGDs on a graph G that is fragmented and distributed across different processors, the only change involves local checking of NGDs in each fragment of G, by adding arithmetic and comparison calculations; the generation of matches of graph patterns, which dominates the cost of the algorithms, remains unchanged. The workload estimation and balancing strategies of [FWX16] remain intact for NGDs. These strategies make the algorithms parallel scalable. As a result, the algorithms remain parallel scalable when they employ NGDs instead of GFDs.

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

4.4.2   Incremental Error Detection

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

We formalize the problem as follows. We consider *batch update* ΔG consisting of a sequence of insertions and deletions of edges, which can simulate modification.

Denote by

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

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

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

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

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

Theorem 4.5: *It is* coNP*-complete to decide, given a set Σ of* NGDs*, a graph G and a batch update ΔG, whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ is empty, even when both G and ΔG have constant sizes.* □

Proof: We next show that it is coNP-complete to decide whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G) = \emptyset$, even if G and ΔG have constant sizes.

Upper bound. We provide an NP algorithm to check, given a set Σ of NGDs, a graph G, and batch update ΔG, whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ is not empty, as follows.

(1) Guess two NGDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \to Y_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to Y_2)$ in Σ, a mapping h_1 of Q_1 in G, and a mapping h_2 of Q_2 in $G \oplus \Delta G$;

(2) Check whether (a) h_1 is a match of Q_1 in G, $h_1(\bar{x}_1) \models X_1$, $h_1(\bar{x}_1) \not\models Y_1$, and (b) h_1 is not a match in $G \oplus \Delta G$; if so, return true; otherwise, continue;

(3) Check whether (a) h_2 is a match of Q_2 in $G \oplus \Delta G$, $h_2(\bar{x}_2) \models X_2$, $h_2(\bar{x}_2) \not\models Y_2$, and (b) h_2 is not a match in G; if so, return true; otherwise, reject the guess.

The correctness of the algorithm follows from the definition of $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$; more specifically, steps (2) and (3) take care of edge deletions and insertions, respectively.

For its complexity, steps (2) and (3) are both in PTIME since $|X_1| + |Y_1| \le |\Sigma|$ and $|X_2| + |Y_2| \le |\Sigma|$. Therefore, the algorithm is in NP, and as a result, the incremental error detection problem for NGDs is in coNP.

Lower bound. We show that it is coNP-hard to decide whether $\Delta\text{Vio}(\Sigma, G, \Delta G)$ is empty, even when both G and ΔG are of constant sizes. It is verified by reduction from the complement of the 3-colorability problem, which is known to be NP-complete [Pap94]. The 3-colorability problem is to decide, given an undirected graph G, whether there exists a proper 3-coloring γ of G such that for each edge (u, v) in G, $\gamma(u) \neq \gamma(v)$. It is known that the problem is still NP-complete when G_1 is a connected graph [GJS76].

Given an undirected connected graph G, we construct a graph G', a set Σ' of NGDs and a batch update $\Delta G'$, such that $\Delta\text{Vio}(\Sigma', G', \Delta G')$ is not empty if and only if G has a proper 3-coloring. Intuitively, G' is used to encode all possible proper 3-colors, and Σ' is used to encode the structure of G and verify possible 3-colorings, while we also use an extra node v_{n+1} and edges directing to it in Σ to ensure that $\Delta\text{Vio}(\Sigma', G', \Delta G')$ is not empty. More specifically, the graph $G' = (V', E', L', F'_A)$ is defined as follows:

- $V' = \{v'_1, v'_2, v'_3, v'_4\}$; here each node in $\{v'_1, v'_2, v'_3\}$ represents a color;
- $E' = \{(v'_1, v'_2), (v'_2, v'_1), (v'_2, v'_3), (v'_3, v'_2), (v'_3, v'_1), (v'_1, v'_3)\}$, these edges make v'_1, v'_2, and v'_3 forming a 3-clique;
- $L'(v'_1) = r$, $L'(v'_2) = g$, and $L'(v'_3) = b$ for three colors; $L'(v'_4) = \chi$; and for each edge $e \in E'$, $L'(e) = \text{`a'}$;
- $F'_A(v'_1).A = 1$, $F'_A(v'_2).A = 1$, $F'_A(v'_3).A = 1$, and $F'_A(v'_4).A = 1$.

The set Σ consists of a singe NGD $\varphi = Q[x_1, \ldots, x_n, x_{n+1}](\emptyset \to (x_1.A = 3))$, where $n = |V|$ and the graph pattern $Q[x_1, \ldots, x_n, x_{n+1}] = (V_Q, E_Q, L_Q, \mu)$ is defined as follows:

- $V_Q = V \cup \{v_{n+1}\}$, *i.e.,* it takes the same set of nodes as G, and an extra node v_{n+1};
- $E_Q = \{(u, v), (v, u) \mid (u, v) \in E\} \cup \{(v, v_{n+1}) \mid v \in V\}$, *i.e.,* each undirected edge (u, v) in G is encoded with two directed edges (u, v) and (v, u), and all nodes in V have an edge directed to v_{n+1};
- each node v_i in V is labeled wildcard: $L_Q(v_i) = \text{`\_'}$, and $L'(v_{n+1}) = \chi$; and for each edge $e \in E_Q$, $L_Q(e) = \text{`a'}$;
- for each node $v_i (i \in [1, n])$ in V_Q, $\mu(x_i) = v_i$.

The batch update $\Delta G'$ is defined as three edge insertions of $(v'_1, v'_4), (v'_2, v'_4)$, and (v'_3, v'_4) in G'.

Note that both G' and $\Delta G'$ are of constant sizes. We next show that this makes a reduction, *i.e.,* $\Delta\text{Vio}(\Sigma', G', \Delta G')$ is not empty if and only if G has a proper 3-coloring. It suffices to prove that $G' \oplus \Delta G' \not\models \Sigma'$, *i.e.,* $\text{Vio}(\Sigma', G' \oplus \Delta G')$ is not empty if and only if G has a proper 3-coloring. Indeed, $\text{Vio}(\Sigma', G')$ is empty before edge insertions as

Q' does not have any match in G'. To see this, observe that (a) all nodes in Q' are connected to v_{n+1} except itself, which are labeled χ, and (b) there is no edge directing to v_4', which is the only node labeled χ, *i.e.,* the only possible match of v_{n+1}, in G'.

In the following, we show that $\mathsf{Vio}(\Sigma', G' \oplus \Delta G')$ is not empty if and only if G has a proper 3-coloring.

($\Rightarrow$) First assume that $\mathsf{Vio}(\Sigma', G' \oplus \Delta G')$ is not empty. Then there exists a match h of Q in $G' \oplus \Delta G'$ such that $h \not\models x_1.A = 3$. Based on h, we construct a 3-coloring γ of G as follows: for each node $v \in V$, $\gamma(v) = L'(h(v))$. This is well-defined by the semantic of graph pattern matching. It remains to show that $\gamma(u) \neq \gamma(v)$ for each edge (u, v) in G, *i.e.,* γ is indeed a proper 3-coloring of G. Assume by contradiction that there exists an edge (v_1, v_2) in G such that $\gamma(v_1) = \gamma(v_2)$. Then $L'(h(v_1)) = L'(h(v_2))$. Since nodes in G' are labeled 3 distinct labels, we have that $h(v_1) = h(v_2)$. However, based on the definition of graph pattern matching, $h(v_1)$ and $h(v_2)$ must not be the same as there should be an edge from $h(v_1)$ to $h(v_2)$ in G', a contradiction. Therefore, γ is a proper 3-coloring of G.

($\Leftarrow$) Conversely, assume that G has a proper 3-coloring γ. We assume *w.l.o.g.* that the 3 colors are r, g and b. It suffices to show that there exists a match h of Q in $G' \oplus \Delta G'$ such that $h \not\models \Sigma$. We define $h(v) = L'^{(-1)}(\gamma(v))$ for each $v \in V$, where $L'^{(-1)}$ is the inverse function of L' in G'; and $h(v_{n+1}) = v_4'$. This is well-defined as L' is a bijection. Since all A-attributes in $G' \oplus \Delta G'$ are assigned 1, if h is a match of Q in $G' \oplus \Delta G'$, we have that $h \not\models \Sigma$. For all nodes in V have outgoing edges, all nodes in V are mapped to v_1', v_2', and v_3'. It remains to show that $h(u) \neq h(v)$ for each edge $(u, v) \in E_Q$ with $u, v \in V$. For if it holds, h is a match of Q in $G' \oplus \Delta G'$, and hence that $\mathsf{Vio}(\Sigma', G' \oplus \Delta G')$ is not empty. Assume by contradiction that there exists an edge $(v_1, v_2) \in E_Q$ with $u, v \in V$ such that $h(v_1) = h(v_2)$. By the definition of h, we have that $\gamma(v_1) = \gamma(v_2)$, which contradicts to the assumption that γ is a proper 3-coloring of G since v_1 and v_2 are connected in G. □

In the rest of the chapter we focus on (parallel) algorithms for incrementally detecting semantic inconsistencies in graphs, by using NGDs. The algorithms complement the batch algorithms of [FWX16], for NGDs used as data quality rules. As remarked earlier in Section 1.3, we are not aware of prior work on incremental error detection in graphs.

4.5   Incremental Detection Algorithms

Despite the challenges noted in Theorem 4.5, we develop two practical algorithms to incrementally detect errors in graphs with NGDs. We show that the algorithms have certain performance guarantees.

We first review the performance guarantees (Section 4.5.1). We then present a sequential incremental error detection algorithm (Section 4.5.2), followed by a parallel algorithm (Section 4.5.3).

To simplify the discussion, we focus on NGDs defined with graph patterns Q that are connected, *i.e.,* there exists a path between any two vertices in Q when Q is treated as an undirected graph. As will be seen in Section 4.5.4, the algorithms can be readily extended to process NGDs that are defined with possibly disconnected patterns.

4.5.1   Performance Guarantees

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

(1) Locality. The first criterion was introduced in [FHT17]. We borrow the following notations from [FHT17]. (a) In a graph G, we say that a node v' is *within d hops* of v if $\mathrm{dist}(v,v') \leq d$ by taking G as an undirected graph, where $\mathrm{dist}(v,v')$ is the shortest distance between v and v' in G. (b) We denote by $V_d(v)$ the set of all nodes in G that are within d hops of v. (c) The *d-neighbor* of v, denoted by $G_d(v)$, is the subgraph of G induced by $V_d(v)$ (see Section 4.1).

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

An incremental error detection algorithm $\mathcal{A}$ is *localizable* if given a set Σ of NGDs, a graph G, and a batch update ΔG to G, its cost is determined only by the size $|\Sigma|$ of NGDs and the sizes of the d_Σ-neighbors of those nodes on the edges of ΔG [FHT17].

Intuitively, a localizable $\mathcal{A}$ can compute $\Delta \mathrm{Vio}(G, \Sigma, \Delta G)$ by inspecting only $G_{d_\Sigma}(v)$, *i.e.,* nodes and edges within d_Σ hops of those nodes v that appear in ΔG. In practice, $G_{d_\Sigma}(v)$ is often small. Indeed, (a) Q is typically small; *e.g.,* 98% of real-life patterns have radius 1, and 1.8% have radius 2 [GFMPdlF11]; since pattern verification is a crucial step in rule mining [GTHS13], practical queries indicate reasonable patterns in

NGDs; hence d_Σ is typically small in practice; and (b) real-life graphs are often sparse; for instance, the average node degree is 14.3 in social graphs [BW13]. Hence, $\mathcal{A}$ can reduce the computations on possibly big graph G to smaller $G_{d_\Sigma}(v)$.

(2) Parallel scalability. The second criterion is adapted from [KRS90], which has been widely used in practice to characterize the effectiveness of parallel algorithms. Consider a sequential algorithm $\mathcal{A}$ for incremental error detection, with cost $t(|G|, |\Sigma|, |\Delta G|)$ measured in the sizes of graph G, Σ of NGDs and batch update ΔG.

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

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

where the notation $\tilde{O}$ hides $\log(p)$ factors (see, *e.g.,* [WZ13]), and $p \ll |G|$, *i.e.,* the number of processors is much smaller than real-life graphs G, as commonly found in the real world.

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

4.5.2 A Sequential Localizable Algorithm

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

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

A number of subgraph matching algorithms have been developed for graphs. As indicated in [LHKL12], most of these algorithms follow a backtracking-based procedure Match_n. Given a pattern Q and a graph G, Match_n first identifies a set $C(u)$ of candidate matches for each pattern node u in Q. Then its main subroutine SubMatch_n is recursively invoked to expand partial solution M, by matching one pattern node of Q with a

node of G in each round, where M is a set of node pairs (u,v) indicating that v matches pattern node u. Note that subgraph homomorphism algorithms [FHK07, Rza14] can also be characterized by the generic Match_n and $\mathsf{SubMatch}_n$.

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

Algorithm. $\mathsf{IncDect}$ incrementalizes batch algorithm Match_n to process G, Σ and $\Delta G = (\Delta G^+, \Delta G^-)$, where ΔG^+ and ΔG^- include $\mathsf{insert}(v,v')$ and $\mathsf{delete}(v,v')$, respectively. (1) It starts with $\Delta\mathsf{Vio}^+(\Sigma,G,\Delta G) = \emptyset$ and $\Delta\mathsf{Vio}^-(\Sigma,G,\Delta G) = \emptyset$. (2) For each NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, it invokes a procedure $\mathsf{IncMatch}$ revised from Match_n to expand $\Delta\mathsf{Vio}^+(\Sigma,G,\Delta G)$ (resp. $\Delta\mathsf{Vio}^-(\Sigma,G,\Delta G)$) with those matches $h(\bar{x})$ of Q in $G \oplus \Delta G$ (resp. G) such that (a) $h(u) = v$ and $h(u') = v'$ for some $(u,u') \in E_Q$ and $\mathsf{insert}(v,v')$ in ΔG^+ (resp. $\mathsf{delete}(v,v')$ in ΔG^-), and (b) $h(\bar{x}) \not\models X \to Y$. Here the pattern $Q[\bar{x}]$ in NGD φ is (V_Q, E_Q, L_Q, μ) (see Section 4.1).

Intuitively, edge insertions may introduce new violations and hence expand $\Delta\mathsf{Vio}^+(\Sigma,G,\Delta G)$, but do not remove existing violations from $\mathsf{Vio}(\Sigma,G,\Delta G)$; on the other hand, deletions expand $\Delta\mathsf{Vio}^-(\Sigma,G,\Delta G)$ only. $\mathsf{IncMatch}$ computes the violations of each NGD φ that are newly added (resp. removed); this is done by identifying those matches of $Q[\bar{x}]$ that have some nodes connected by edges involved in ΔG^+ (resp. ΔG^-) and violating the attribute dependency $X \to Y$. We assume *w.l.o.g.* that no insert e and delete e in ΔG are about the same edge e, which can be easily detected.

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

(1) Given pattern Q, $\mathsf{IncMatch}$ first finds out whether each edge (v,v') in ΔG is a candidate match of some pattern edge (u,u') in Q, i.e., $L_Q(u) = L(v)$, $L_Q(u') = L_Q(v')$ and $L_Q(u,u') = L(v,v')$. This is in contrast to Match_n that searches candidate matches

in the entire graph G. If (v, v') makes a candidate, this match forms an initial partial solution $h_{\mathsf{up}}(u, u') = (v, v')$, referred to as an *update pivot* of Q *triggered by* unit update of edge (v, v'). IncMatch then expands $h_{\mathsf{up}}(u, u')$ by recursively invoking IncSubMatch as in Match_n to compute update-driven violations $h(\bar{x})$.

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

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

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

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

Example 4.6: Suppose that the edge $(\text{NatWest Help}, 1)$ labeled status is deleted from G_4 of Fig. 4.1. Given NGD φ_4 of Example 4.3, IncDect calls IncMatch to detect update-driven violations. It first finds that the deleted edge is a candidate match of pattern edge (x, s_1) in Q_4. That is, an update pivot $h_{\mathsf{up}}(x, s_1) = (\text{NatWest Help}, 1)$ of Q_4 is built. IncMatch then expands $h_{\mathsf{up}}(x, s_1)$ recursively by inspecting the neighbors of candidate matches until all pattern nodes of Q_4 are matched. For instance, node 22000

in G_4 is the only candidate match for m_1. Finally, it returns violation $h_{\mathsf{up}}(\bar{x})$ that includes all the nodes of G_4, where NatWest_Help is found a fake account. This is an update-driven violation to be removed. Note that there is another update pivot $h'_{\mathsf{up}}(y, s_2) = (\text{NatWest Help}, 1)$ triggered by the deletion. However, this partial solution does not yield any violation as $a * (m_1.\mathsf{val} - m_2.\mathsf{val}) + b * (n_1.\mathsf{val} - n_2.\mathsf{val}) > c$ in φ_4 is evaluated false when expanding $h'_{\mathsf{up}}(y, s_2)$ in IncMatch.

Besides delete(NatWest Help, 1), suppose that four edges are inserted into G_4 to indicate that another account NatWest_Help$_1$ has 1 following and 2 followers, and refers to company NatWest with status 1. Given this batch update, IncDect computes the same violation to be removed as above. Indeed, there are no newly introduced violations since all matches expanded from update pivots triggered by edge insertions are pruned by literal validation.   □

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

Algorithm IncDect runs in $O(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|})$ time in the worst case, where $G_{d_\Sigma}(\Delta G)$ denotes the union of d_Σ-neighbors of nodes involved in ΔG. Hence it is localizable. Indeed, observe the following. (a) The computation performed by each invocation of procedure IncMatch is confined in the d_Σ-neighbors of an unit update in ΔG, since partial solutions are expanded by accessing the neighbors only. (b) The cost of checking linear arithmetic expressions is subsumed by the cost of candidate selection in the matching process.

4.5.3   A Parallel Scalable Algorithm

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

Overview. Algorithm PIncDect works with p processors $S_1, \ldots, S_p$ on a graph G that is partitioned via edge-cut [AR06] or vertex-cut [KC12]; the fragments of G are distributed across p processors. In a nutshell, PIncDect first finds update pivots of patterns in Σ triggered by unit updates, and distributes these partial solutions as work units

to p processors. Then each processor handles its workload and identifies violations *in parallel*, driven by updates like in IncDect.

However, there are two challenges. (1) The d_Σ-neighbor of a node may reside in different fragments. (2) The workloads of some processors may be skewed, since (a) the workload assignment may be unbalanced; and (b) some work unit may take much longer, *e.g.,* when accessing a large d_Σ-neighbor. Note that work stealing and shedding [HCD⁺16, BL99] do not solve (b) by re-assigning work units.

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

Candidate neighborhood. Similar to IncDect, initially PIncDect checks whether each unit update $\mathsf{insert}(v, v')$ or $\mathsf{delete}(v, v')$ in ΔG triggers an update pivot, *i.e.,* partial solution $h_{\mathsf{up}}(u, u') = (v, v')$ for some pattern nodes u and u' in Q from Σ, at each processor. It then identifies the d_{Q_u}-neighbor of node v in $G \oplus \Delta G^+$, referred to as the *candidate neighborhood* $N_C(h_{\mathsf{up}}(u, u'))$ for $h_{\mathsf{up}}(u, u')$. Here d_{Q_u} denotes the length of the longest shortest path between u and other nodes in Q. When v is involved in multiple update pivots, only the union of their neighborhoods is extracted. Multiple processors coordinate to extract such a neighborhood when it is fragmented, by notifying each other the remaining size of the region to be identified via messages passed through "borders", *e.g.,* crossing edges in edge cut [AR06] or entry and exit nodes in vertex cut [KC12].

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

Moreover, for each node v in $N_C(\Delta G, \Sigma)$, PIncDect evenly "partitions" its adjacency list $v.\mathsf{adj}$ across processors by annotating local partition (instead of physically breaking it up). At each processor S_i, its partition of $v.\mathsf{adj}$ is referred to as a *partial copy* $v.\mathsf{adj}_i$, which is disjoint from $v.\mathsf{adj}_j$ for $i \neq j$. The update pivots are also evenly partitioned into p disjoint sets. Each S_i maintains one set BVio_i as its *workload*. A partial solution

to be expanded is a *work unit*.

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

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

Candidate filtering. Consider $h_{\mathsf{up}}(u_0,\ldots,u_k) \in \mathsf{BVio}_i$, a partial solution for Q of NGD $Q[\bar{x}](X \to Y)$ to be expanded at processor S_i, where u_j is a pattern node in Q that is already matched for $j \in [0,k]$. The next pattern node to be matched is u_{k+1} such that u_{k+1} is connected to u_r in Q, where $r \in [0,k]$. The candidates for u_{k+1} are selected from the neighbors of $h_{\mathsf{up}}(u_r)$, just like in procedure IncSubMatch (Section 4.5.2). Here PIncDect estimates the *sequential cost* as $|h_{\mathsf{up}}(u_r).\mathsf{adj}|$, and the *parallel cost* as

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

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

Verification. After $h_{\mathsf{up}}(u_0,\ldots,u_k)$ is expanded with u_{k+1} at processor S_i, PIncDect checks the edges between the candidate match $h_{\mathsf{up}}(u_{k+1})$ and other matches $h_{\mathsf{up}}(u_0)$, $\ldots, h_{\mathsf{up}}(u_k)$, to verify that the expansion yields a valid partial solution. It may split the verification work based on the size of the adjacency list. Here the sequential cost is estimated as $|h_{\mathsf{up}}(u_{k+1}).\mathsf{adj}|$ and the parallel cost is

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

If the parallel cost is smaller, it broadcasts $h_{\mathsf{up}}(u_0,\ldots,u_k,u_{k+1})$ to check at all processors S_j by using their partial copy $h_{\mathsf{up}}(u_{k+1}).\mathsf{adj}_j$; the results of checking are sent back to S_i to decide the qualification of the partial solution. If qualified, it is added to BVio_i at S_i for further expansion, unless it makes a complete match of Q.

Algorithm: PIncDect

Input: A fragmented graph G across p processors $S_1, \ldots S_p$,

       a set Σ of NGDs, and a batch update ΔG.

Output: The set $\Delta \mathsf{Vio}(\Sigma, G, \Delta G)$ of violations.

1.   annotate the edges involved in ΔG;

2.   **for each** unit update of (v, v') in ΔG and pattern edge (u, u') in

      Q of NGD $\psi = Q[\bar{x}](X \to Y) \in \Sigma$ **having** $L_Q(u) = L(v)$,

      $L_Q(u') = L(v)$, **and** $L_Q(u, u') = L(v, v')$ **do**

3.   construct update pivot $h_{\mathsf{up}}(u, u') = (v, v')$;

4.   identify the d_{Q_u}-neighbor of v;

5.   construct $N_C(\Delta G, \Sigma)$ in parallel and replicate it at all processors;

6.   evenly partition adjacency lists and work units across p processors;

7.   invoke $\mathsf{PIncMatch}(\mathsf{BVio}_i)$ at processor S_i for all $i \in [1, p]$;

8.   **repeat**

9.   periodically balance workload across p processors at interval intvl;

10. **until** all S_i's return Vio_i;

11. $\Delta \mathsf{Vio}(\Sigma, G, \Delta G) := \bigcup_i \mathsf{Vio}_i$;

12. **return** $\Delta \mathsf{Vio}(\Sigma, G, \Delta G)$;

Figure 4.3: Algorithm PIncDect

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

To cope with this, PIncDect checks the skewness of processors at a time interval intvl (45 seconds in our experiments). If the skewness of S_i exceeds a threshold η ($\eta = 3$ in our experiments), PIncDect evenly distributes the work units in BVio_i to those processors S_j's having skewness below η' ($\eta' = 0.7$ in experiments), extending BVio_j's. We allow processors to send and receive work units at any time, without being blocked by synchronization barriers.

Algorithm. Putting these together, we present the main driver of algorithm PIncDect in Fig. 4.3. It first identifies the candidate neighborhood for each update pivot triggered by unit updates in ΔG and Σ (lines 2-4), and replicates the union of all candidate

---

Procedure PIncMatch /* executed at each worker S_i in parallel*/

Input: Workload $BVio_i$.

Output: The set Vio_i of local violations.

1. $Vio_i := \emptyset$;

2. **while** there **exists** a partial solution to be expanded **do**

3.   **for each** $h_{up}(u_0, \ldots, u_k) \in BVio_i$ by matching u_{k+1}
          with neighbors of $h_{up}(u_r)$ **do**

4.   **if** $|h_{up}(u_r).adj| \leq C(k+1) + |h_{up}(u_r).adj|/p$ **then**

5.   expand $h_{up}(u_0, \ldots, u_k)$ at S_i;

6.   **else** broadcast $h_{up}(u_0, \ldots, u_k)$ and expand it in parallel;

7.   **for each** $h_{up}(u_0, \ldots, u_k, u_{k+1})$ to be verified at S_i **do**

8.   **if** $|h_{up}(u_{k+1}).adj| \leq C(k+2) + |h_{up}(u_{k+1}).adj|/p$ **then**

9.   verify $h_{up}(u_0, \ldots, u_{k+1})$ at S_i;

10.   **else** broadcast $h_{up}(u_0, \ldots, u_{k+1})$ and verify it in parallel;

11.   **if** $h_{up}(u_0, \ldots, u_k, u_{k+1})$ is a valid partial solution **then**

12.   **if** it is a complete match **then** add it to Vio_i;

13.   **else** add it to $BVio_i$;

14. **return** Vio_i;

Figure 4.4: Procedure PIncMatch

neighborhoods at all processors (line 5). The update pivots are also evenly distributed (line 6). Then PIncDect invokes procedure PIncMatch (shwon in Fig. 4.4) at each processor S_i with initial workload $BVio_i$, in parallel for $i \in [1, p]$ (line 7). It periodically balances workload (line 9), until all processors complete their work (line 10). At this point, PIncDect collects local violations Vio_i's from all processors. The union of all Vio_i's is $\Delta Vio(\Sigma, G, \Delta G)$ (line 11) and is returned (line 12).

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

Example 4.7: Consider a graph G revised from G_4 of Fig. 4.1 by including additional

98 accounts NatWest_Help$_i$ for $i \in [1, 98]$, where each NatWest_Help$_i$ has 1 following and 2 followers, and it refers to company NatWest with status 1. Assume that G is fragmented across 4 processors. Recall NGD φ_4 and delete(NatWest Help, 1) from Example 4.6. After generating update pivot $h_{\mathsf{up}}(x, s_1)$ as in Example 4.6, algorithm PIncDect identifies in parallel $N_C(h_{\mathsf{up}}(x, s_1))$, which is the 3-neighbor of node NatWest Help. This subgraph is replicated at all 4 processors. Moreover, the adjacency lists are evenly "partitioned" by annotating partial copies. For instance, each processor maintains a partial copy of 25 nodes (*i.e.,* accounts) for the adjacency list of the company node NatWest. Then it expands $h_{\mathsf{up}}(x, s_1)$ to find update-driven violations.

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

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

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

Theorem 4.6: PIncDect *is parallel scalable relative to* IncDect. $\qquad \square$

Proof: We show that with p processors, algorithm PIncDect runs in $\tilde{O}(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}/p)$ time, where $p < |G_{d_\Sigma}(\Delta G)|$. Obviously, identifying the candidate neighborhoods for update pivots triggered by ΔG and Σ takes $\tilde{O}(|G_{d_\Sigma}(\Delta G)|)$ time. We next analyze the cost for parallel expansion of partial solutions. The work units (partial solutions) are dynamically balanced across p processors. The total time for candidate filtering in processing partial solutions of size k is at most $N_k(k+1)(Ck + |G_{d_\Sigma}(\Delta G)|/p)$, and their verification needs at most $N_{k+1}(C(k+1) + |G_{d_\Sigma}(\Delta G)|/p))$ time, where N_k denotes the number of partial matches of size k. Moreover, it inspects partial solutions with size at most $|V_\Sigma| - 1$, where V_Σ denotes the set of all pattern nodes in Σ. Hence parallel expansion

takes at most $\sum_{k=2}^{|V_\Sigma|-1} \left(N_k(k+1)(Ck + \frac{|G_{d_\Sigma}(\Delta G)|}{p}) + N_{k+1}(C(k+1) + \frac{|G_{d_\Sigma}(\Delta G)|}{p})\right) <$ $\sum_{k=1}^{|\Sigma|-1} 2C|\Sigma||G_{d_\Sigma}(\Delta G)|^k(|\Sigma| + \frac{|G_{d_\Sigma}(\Delta G)|}{p}) < \frac{4C|\Sigma|(1-|G_{d_\Sigma}(\Delta G)|^{|\Sigma|-1})|G_{d_\Sigma}(\Delta G)|^2}{(1-|G_{d_\Sigma}(\Delta G)|)p} =$ $\tilde{O}(\frac{|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}}{p})$ time, which dominates the cost of PIncDect.   This verifies the parallel scalability of algorithm PIncDect relative to sequential IncDect. $\square$

4.5.4 Handling disconnected patterns

We show that the sequential algorithm IncDect and parallel algorithm PIncDect in Section 4.5 can be easily extended to detect violations of NGDs with disconnected patterns.

Semi-locality. Locality is sometimes too strong to evaluate incremental algorithms. We next introduce a weaker notion, *semi-locality*, by incorporating auxiliary information.

We say an incremental error detection algorithm $\mathcal{A}$ is *semi-localizable* if given a set Σ of NGDs, a graph G, updates ΔG, and in addition, auxiliary structure AUX that has size polynomial in $|G|$ and $|\Sigma|$ and is distributed over a set V_{AUX} of nodes in G, its cost can be expressed by $|\Sigma|$ and the sizes of d_Σ-neighbors of nodes involved in ΔG and V_{AUX}.

We start with the auxiliary structure used by IncDect.

Auxiliary structure. For each disconnected pattern $Q = (Q_1, \ldots, Q_m)$ in Σ, where Q_i's are the (maximum) connected components in Q for $i \in [1, m]$, we maintain a set of *match pivots* for every Q_i. Each match pivot for Q_i in G is a triple $((u, u'), (v, v'), N)$ stored at v or v' indicating that (v, v') in G is a *confirmed* match of (u, u') from Q_i, and is included in N different matches of Q_i in G in total. All match pivots for Q_i share the same pattern edge (u, u'). The number N is obtained when validating NGDs of Σ in the batch process, using *e.g.,* algorithms of [FWX16]. Thus it is guaranteed that the size of all match pivots maintained on G for disconnected patterns in Σ is bounded by $O(|\Sigma||G|)$.

We next extend sequential IncDect and parallel PIncDect to cope with disconnected patterns by using match pivots. Intuitively, both are revised by incorporating the combination of matches of distinct connected components.

Sequential algorithm. We extend procedure IncMatch of IncDect to process disconnected patterns. More specifically, for each NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ with disconnected pattern $Q = (Q_1, \ldots Q_m)$, IncMatch first identifies update-driven matches of each connected Q_i as described in Section 4.5.2. If such update-driven matches exist,

it continues to find *prior matches* of other Q_j's ($j \neq i$), starting from the match pivots for Q_j in G, also along the same lines as for connected patterns. The prior matches are computed without inspecting any edge involved in ΔG, *i.e.,* nodes that connected by annotated edges are pruned from the candidates in IncSubMatch. Then IncMatch constructs the matches of Q by combining each update-driven match of Q_i with (a) prior matches, or (b) other update-driven matches of the same type, *i.e.,* both triggered by edge insertion or deletion, of Q_j's ($j \neq i$). IncMatch evaluates those literals of $Q[\bar{x}]$ having variables from multiple components of Q to find those violating $X \to Y$. These matches are included in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ and $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$ accordingly, *i.e.,* triggered by insertions and deletions, at last.

Parallel algorithm. Algorithm PIncDect first finds the update-driven matches of each (maximum) connected component as in Section 4.5.3, and then discovers the prior matches in case of the existence of update-driven matches, starting from match pivots. It identifies the candidate neighborhood for each match pivot, followed by parallel expansion as in processing connected ones to obtain these prior matches. For each disconnected pattern $Q = (Q_1, \ldots, Q_m)$ of NGD φ in Σ, PIncDect replicates the update-driven matches and prior matches computed for Q_i ($i \in [1, m]$), *i.e.,* partial solutions of Q, by broadcasting. These partial solutions are maintained using sorted arrays. Then all processors combine the matches of distinct connected components in parallel to compute the violations of φ along the same lines as in the sequential counterpart.

Since the number of matches for each connected component is already known, the computation workload can be evenly partitioned by combining partial solutions with certain indices in the arrays at each processor S_j, where the indices are decided by the predefined order on the p processors. More specifically, the indices $\{\text{id}_i \mid i \in [1, m]\}$ of each combination of partial solutions processed by S_j satisfy $\sum_{i=1}^{m-1} (\text{id}_i \cdot \prod_{s=i+1}^{m} N_{Q_s}) + \text{id}_m$ mod $p = j$, where N_{Q_s} denotes the number of matches of connected Q_s for $s \in [1, m]$ and p is the number of processors available. Each group of indices can be computed efficiently as the productions need to be computed only once for all possible combinations after broadcasting the partial solutions. Finally, the quantified combinations triggered by insertions (resp. deletions) are included in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (resp. $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$).

Both algorithms above apply a "lazy" strategy that prior matches are enumerated only when necessary, *i.e.,* there exist update-driven matches for some connected component.

Example 4.8: Consider a graph G' derived from G of Example 4.7 by removing the company node "NatWest, Help" along with all the edges adjacent to it. Consider an NGD φ'_4 revised from φ_4 of Example 4.3 by dropping the pattern node w and its adjacent edges from Q_4, while keeping the attribute dependency of φ_4 unchanged. Denote by Q'_4 the graph pattern of φ'_4, and denote by Q^1_4 and Q^2_4 the connected components of Q'_4 having nodes x and y, respectively. Suppose that 297 edges are inserted into G' to express that account "NatWest Help$_i$" has 22000 followings, 75900 followers with status 1 for each $i \in [0, 98]$, making updates $\Delta G'$.

Given $\Delta G'$, IncDect first finds that each introduced account and its properties constitute an update-driven match of Q^1_4. Therefore, it continues to identify prior matches of Q^2_4 using the match pivots maintained. For instance, each triple $((y, m_2), (\text{NatWest\_Help}_i, 1), 1)$ for $i \in [1, 98]$ is a match pivot of Q^2_4. There are also 99 prior matches of Q^2_4 computed by IncDect, in which y is mapped to the company node "NatWest_Help$_i$" ($i \in [1, 98]$) and "NatWest_Help". These prior matches are combined with the 99 update-driven matches at last. Since there exists a literal that has variables from the 2 components, validation of this expression is conducted after the combination. The qualified $99 \times 99 = 9801$ pairs are indeed the violations introduced by $\Delta G'$.

Now consider PIncDect. As opposed to sequential IncDect, PIncDect combines this two sets of matches in parallel. Assume that there are 9 processors available and the update-driven and prior matches are sorted according to the subscripts of the company nodes. Here node "NatWest_Help" is assigned a subscript of 0. Then each processor S_k ($k \in [0, 8]$) outputs only 1089 violations, where each pair contains company nodes "NatWest Help$_i$" and "NatWest_Help$_j$" such that $99i + j \mod 9 = k$ for $i, j \in [0, 98]$.   □

Analysis. (1) Algorithms IncDect and PIncDect correctly compute $\text{Vio}(\Sigma, G, \Delta G)$ since each possible violation for NGDs with disconcerted patterns, *i.e.,* combination of matches of distinct connected components, is identified and checked. (2) The time complexity of IncDect is $O(|\Sigma||G_{d_\Sigma}(\Delta G \cup V_{\text{PM}(\Sigma, G)})|^{|\Sigma|})$ if there are NGDs with disconnected patterns in Σ, where $\text{PM}(\Sigma, G)$ refers to the set of match pivots in G for connected components of patterns from Σ, and is distributed over $V_{\text{PM}(\Sigma, G)}$. Indeed, computing prior matches is also done within the d_Σ-neighbors of some match pivot via procedure IncMatch. Therefore, IncDect is semi-localizable. (3) Algorithm PIncDect runs in $\tilde{O}(|\Sigma||G_{d_\Sigma}(\Delta G \cup V_{\text{PM}(\Sigma, G)})|^{|\Sigma|}/p)$ time with p processors when processing dis-

connected patterns. Hence it is parallel scalable relative to IncDect. To see this, (a) one can verify along the same lines as in Section 4.5.3 that discovering matches of distinct connected components needs at most $\tilde{O}(t^s(|G|,|\Sigma|,|\Delta G|)/p)$ time, where $t^s(|G|,|\Sigma|,|\Delta G|)$ refers to the corresponding time consumed by IncDect, and (b) the dominating cost of combining matches is evenly partitioned.

Incremental maintenance. To facilitate the computation of prior matches, match pivots of each connected component are maintained dynamically. Initially they are obtained by extracting the results, *i.e.,* matches of individual connected components, after executing batch algorithm on G. Thereafter, the counter N of each match pivot increases (resp. decreases) when the incremental algorithms find update-driven matches triggered by edge insertion (resp. deletion) containing it, upon which they also decide the addition and removal of match pivots for subsequent processing.

4.6 Experimental Study

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

Experimental setting. We used three real-life graphs: (a) DBpedia [DBp], a knowledge base with 28 million entities of 200 types and 33.4 million edges of 160 types; (b) YAGO2, an extended knowledge graph of YAGO [SKW07] with 3.5 million nodes of 13 types and 7.35 million edges of 36 types; and (c) Pokec [Pok], a social network with 1.63 million nodes of 269 types and 30.6 million links of 11 types.

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

<u>NGDs</u>. We generated a set Σ of 100 meaningful NGDs for each graph. Attributes were drawn from the real-life data for DBpedia, YAGO2 and Pokec, and from $\mathcal{L}$ for synthetic graphs.

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

size $|G|$ remains unchanged.

Algorithms. In Java, we implemented (1) sequential IncDect (Section 4.5.2) vs. Dect, a batch error detection algorithm with NGDs, extended from the algorithm for GFDs [FWX16]; (2) parallel PIncDect (Section 4.5.3) vs. PDect, an extension of the parallel batch detection algorithm in [FWX16] to NGDs; and (3) parallel PIncDect$_{ns}$, PIncDect$_{nb}$ and PIncDect$_{NO}$, variants of PIncDect with no work unit splitting, no workload balancing, and neither of the two, respectively.

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

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

Exp-1: Effectiveness of incremental error detection. We first evaluated the incremental algorithms against their batch counterparts. Fixing $\|\Sigma\| = 50$ and $d_\Sigma = 5$, we varied the size $|\Delta G|$ of updates from 5% up to 40% in 5% increments. The results are reported in Figures 4.5(a)–4.5(d) over DBpedia, YAGO2, Pokec and Synthetic G, respectively (*y*-axis in logarithmic scale). We find the following.

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

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

(c) The larger $|\Delta G|$ is, the slower all incremental algorithms are, while the batch algorithms Dect and PDect are indifferent to $|\Delta G|$, as expected. In all cases, PIncDect
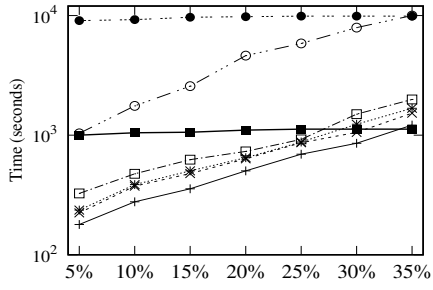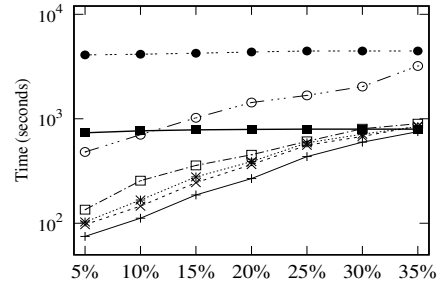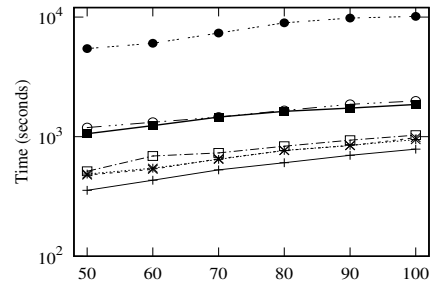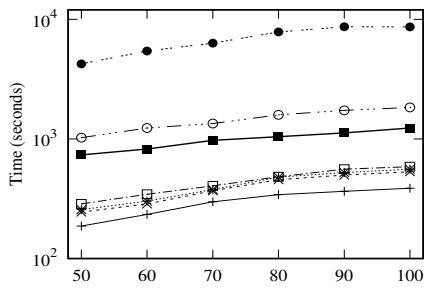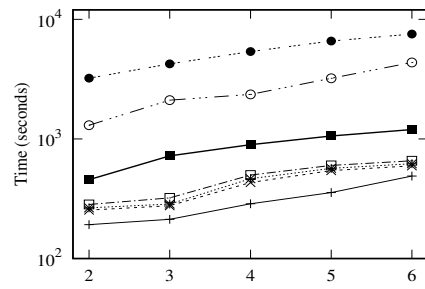
Figure 4.5: Performance evaluation

performs the best.

(d) Incremental error detection is feasible in practice: PIncDect takes 693s on DBpedia when $|\Delta G|$ is 25% of $|G|$, and IncDect takes 5840s, as opposed to 1121s (resp. 9878s) by PDect (resp. Dect).
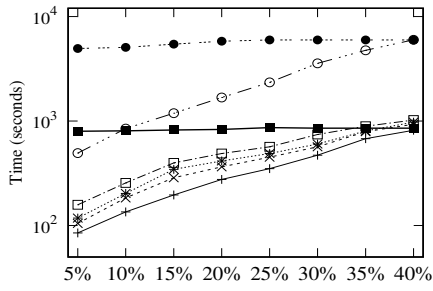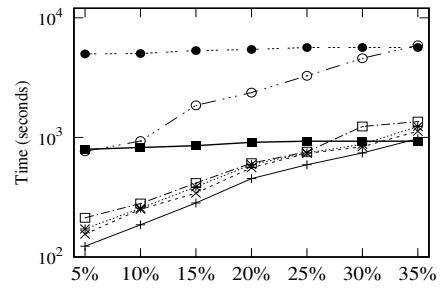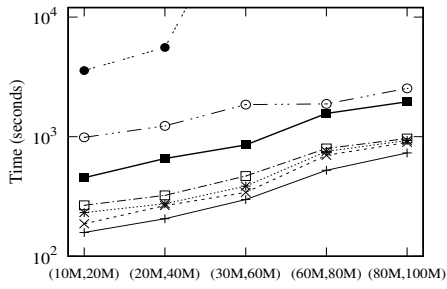
(e) All incremental algorithms are insensitive to the ratio γ of edge insertions to deletions, which is verified by varying γ (not shown).

Exp-2: Impact of $|G|$. We evaluated the impact of $|G|$ using synthetic graphs. Fixing $|\Delta G|$ as 15% of $|G|$ and using the same NGDs as in Exp-1, we varied $|G|$ from (10M,20M) to (80M,100M). As shown in Fig. 4.5(e), (a) all the algorithms take longer on larger G, as expected, (b) incremental algorithms are less sensitive to $|G|$ than their batch counterparts, and (c) PIncDect does the best among all.

     The results over real-life graphs are consistent (not shown).

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

Varying $\|\Sigma\|$. Fixing $d_{\Sigma} = 5$, we varied $\|\Sigma\|$ from 50 to 100 (our industry collaborator uses 95 rules [Bai17]). As shown in Figures 4.5(f) and 4.5(g) on DBpedia and YAGO2, respectively, we can see that (a) the more NGDs are in Σ, the longer is taken by all the algorithms, as expected, and (b) PIncDect and IncDect scale well with $\|\Sigma\|$.

     The results on Pokec and Synthetic are consistent (not shown).

Varying d_{Σ}. Fixing $\|\Sigma\| = 50$, we varied d_{Σ} from 2 to 6 (up to 36 edges). Figures 4.5(h) and 4.6(a) show that all algorithms take longer over larger d_{Σ} on DBpedia and Pokec, respectively. This is consistent with our analysis that the costs of our localizable incremental algorithms increase when d_{Σ} gets larger. Nonetheless, PIncDect is feasible with real-life NGDs, *e.g.,* it takes 489s on DBpedia when $d_{\Sigma} = 6$, as opposed to 1197s by PDect and 7532s by Dect.

     The results on YAGO2 and Synthetic are consistent (not shown).

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

Varying p. Fixing $C = 60$ and intvl $= 45s$, we varied p from 4 to 20. As shown in Figures 4.6(b), 4.6(c), 4.6(d) and 4.6(e) over DBpedia, YAGO2, Pokec and Synthetic, re-
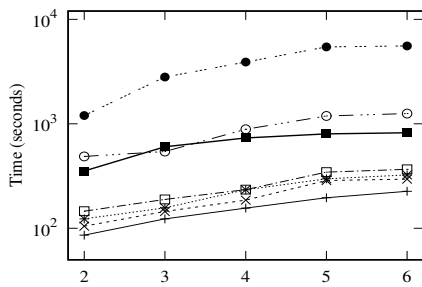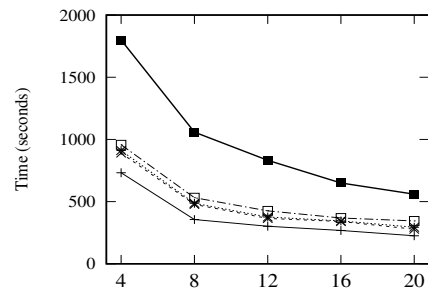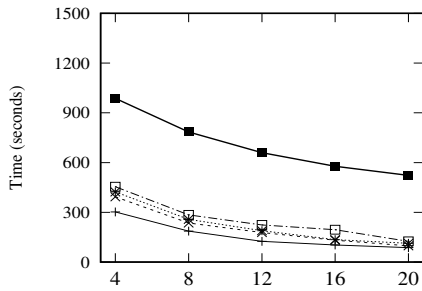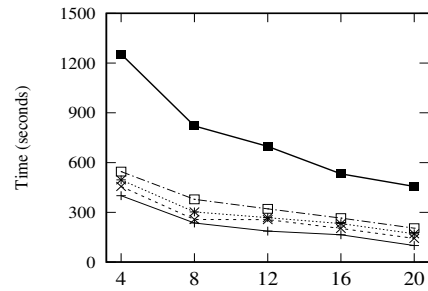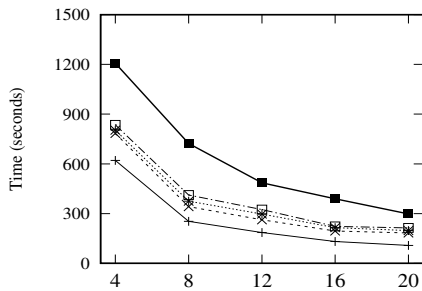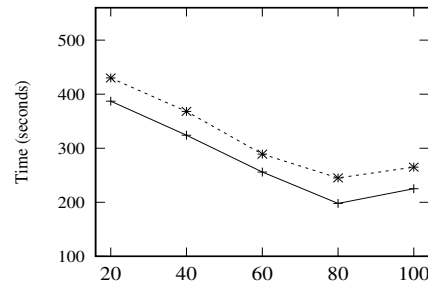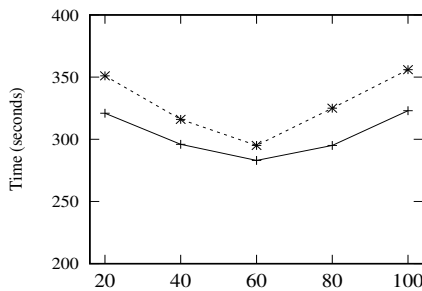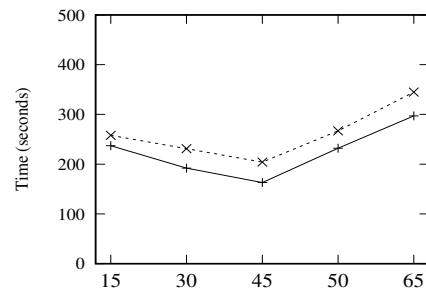
Figure 4.6: Performance evaluation (cont.)

spectively, when p changes from 4 to 20, (a) PIncDect and PDect perform much better and are on average 3.7 and 3.8 times faster than IncDect and Dect, respectively, and (b) PIncDect consistently outperforms PDect, PIncDect_{ns}, PIncDect_{nb} and PIncDect_{NO}: on average it is 2.47 to 3.14, 1.32 to 1.37, 1.44 to 1.53, and 1.53 to 1.72 times better, respectively.

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

Varying C. Fixing $p = 8$ and intvl $= 45s$, we evaluated the impact of latency parameter on PIncDect and PIncDect_{nb} by tuning C from 20 to 100 in 20 increments. As shown in Figures 4.6(f) and 4.6(g) over Pokec and Synthetic, PIncDect performs the best when C is 80 and 60, respectively, taking 198s and 283s. On one hand, PIncDect favors parallel computation with smaller C to split work units; on the other hand, PIncDect has a bias towards local computation with larger latency C to reduce the communication cost.

The results on DBpedia and YAGO2 are consistent (not shown).

Varying intvl. Fixing $p = 8$ and $C = 60$, we varied intvl from 15s to 65s in 15s increments, to evaluate impact of intervals for monitoring workloads on PIncDect and PIncDect_{ns}. As shown in Figure 4.6(h) on *YAGO2*, the "optimal" intvl is 45s for PIncDect. Similar to latency C, while smaller intvl helps workload balancing, it incurs more communication cost. Hence we need to strike a balance.

The results on Pokec, DBpedia and Synthetic are consistent.

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

Chapter 5

Cleaning Graphs with Certainty

We have seen in Chapters 3 and 4 that dependencies such as keys and NGDs can be used as data quality rules to tell us whether the graphs are dirty or clean, *i.e.,* whether there exists duplicate entities or semantic inconsistencies. However, they cannot be used directly to fix the errors. To see this, consider the following example.

Example 5.1: A fraction of DBpedia is depicted as graph G in Fig. 5.1. It shows two football clubs f_1 and f_2, both named Arsenal, along with their stadiums (s_1-s_4) and the football leagues (c_1-c_2) that they play in. Each node in G denotes an entity with type labeled in brackets and carries a tuple of attributes. It is known that

φ_1: a football club cannot participate in two (domestic) football leagues of different countries; and

φ_2: if a stadium is used as a venue of one football league, and is owned by a football club that plays in the same league, then the stadium and the league must be in the same country.

These two rules detect *inconsistencies* among the values of attribute country in G for (a) football leagues c_1 and c_2 (by φ_1); and (b) stadium s_3 and football league c_2 (by φ_2). However, they do 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 c_2.country to "Argentina" by taking c_1.country adds a new violation of φ_2 by football league c_2 and stadium s_4. □

The example raises the following questions. Is there a systematic method to clean a graph, *i.e.,* to generate fixes to errors in graphs detected by dependencies? Moreover, can we ensure the fixes to be *certain*, *i.e.,* guaranteed correct? While there has been work on cleaning relations (*e.g.,* [BFFR05, CFG+07, PSC+15, YEN+11, FLM+12,
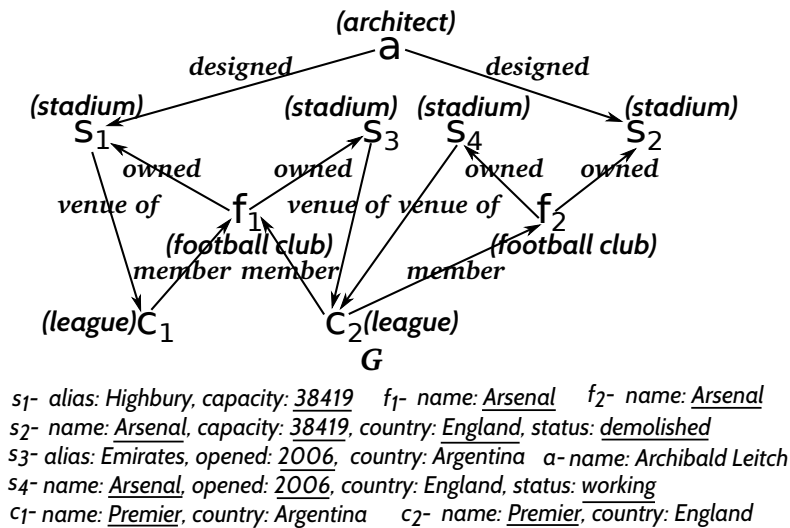
Figure 5.1: A labeled directed graph

FLM$^+$11]), no prior work has studied how to clean graphs with correctness guarantee. It is more challenging to clean graphs than relations. Cleaning a graph requires not only modifications to its attribute values but also changes to its topological structures. Worse yet, real-life graphs are typically semi-structured and schemaless.

This chapter answers these questions. We propose a rule-based method Analogist to clean graphs by combining object identification and data repairing. More specifically, we extend graph entity dependencies (GEDs) as data quality rules, and Analogist leverages GEDs to find certain fixes to the violations detected, based on a given set of ground truth (confirmed attribute values and entity matches).

Example 5.2: A block Γ of ground truth for G is underlined in Fig. 5.1, *e.g.,* the names of football leagues and stadiums, which are verified by, *e.g.,* domain experts or crowd-sourcing. Consider the rules φ_1 and φ_2 of Example 5.1 and the following keys for identifying entities:

φ_3: a stadium can be identified by its attribute value of capacity and the id of its architect (entity); and

φ_4: two football leagues refer to the same one if they have the same name and country attribute values.

Given the rules and Γ, Analogist fixes the inconsistencies of Example 5.1 as follows. (a) It first identifies stadiums s_1 and s_2 by φ_3 since they have the same capacity and architect that are asserted correct by Γ. (b) As a result, the country value of s_1 is *enriched* by taking "England" from s_2, which is assured correct by Γ. (c) This correct value is then *propagated* to c_1.country via rule φ_2, and one step further to c_2.country

by φ_1, *i.e.,* country of "England" is confirmed correct for both leagues c_1 and c_2. (d) It next identifies c_1 and c_2 by using φ_4 and the confirmed values of country above. (e) Finally, φ_2 tells us to correct s_3.country by taking the value of c_2.country.

Analogist *interleaves* data repairing and object identification: identifying s_1 and s_2 in step (a) helps it correct the country values of s_1, c_1 and c_2 in steps (b) and (c), which in turn allows Analogist to identify c_1 and c_2 in step (d), followed by repairing in step (e).   □

To the best of our knowledge, this work is the first effort to clean graph-structured data with certain fixes, from foundation to (parallel) algorithms. As confirmed by our industry collaborators [Bai17], rule-based methods account for 90% of the industry effort for data cleaning in practice. Our empirical study suggests that the method is promising for cleaning large-scale real-life graphs.

5.1 GEDs as Data Quality Rules

We extend GEDs defined in [FL17] as data quality rules.

5.1.1 Preliminaries

We first review basic notations of data graphs, patterns and graph pattern matching. Assume three countably infinite sets Θ, Υ and U, denoting (node and edge) labels, attributes and constant values, respectively.

Data Graphs. We consider directed graphs $G = (V, E, L, F_A)$, where (a) V is a finite set of nodes, and each node $v \in V$ carries a label $L(v) \in \Theta$, (b) $E \subseteq V \times V$ is a finite set of edges, in which $e = (v, v')$ is an edge from node v to v', e is identified by an edge id, and carries a label $L(e) \in \Theta$; (c) each node $v \in V$ carries a tuple $F_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$ of *attributes* of a finite arity, 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 v has a special attribute id denoting its *node identity*.

That is, we consider finite directed graphs in which nodes and edges are labeled. Each node v carries v.id and attributes for, *e.g.,* properties, keywords and rating, as in property graphs.

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 alphabet Θ. Moreover, we allow wildcard '_' as a special label in Q.

Graph 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 in Θ.

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

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 in graph G.

5.1.2 Graph Entity Dependencies

We now introduce an extension of the GEDs of [FL17], also referred to as GEDs, by supporting limited negation in terms of inequality.

A *graph entity dependency* (GED) is defined as $\varphi = Q[\bar{x}](X \rightarrow Y)$, where $Q[\bar{x}]$ is a graph pattern, and X and Y are (possibly empty) sets of literals of $\bar{x}$. A *literal of $\bar{x}$* is one of the following: for $x, y \in \bar{x}$,

    (a) *constant literal $x.A = c$ or $x.A \neq c$*, where c is a constant, and A is an attribute in Υ that is not id;

    (b) *variable literal $x.A = y.B$ or $x.A \neq y.B$*, where A and B are attributes in Υ that are not id; or

    (c) id *literal $x.\text{id} = y.\text{id}$ or $x.\text{id} \neq y.\text{id}$*.

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

Intuitively, GED φ is a combination of (1) a *pattern Q*, to identify entities in a graph, and (2) an *attribute constraint $X \rightarrow Y$* that extends functional dependencies (FDs), to be applied to the entities identified by Q. Constant literals $x.A = c$ enforce constant bindings like in CFDs [FGJK08], which have proven useful in data cleaning. An id literal $x.\text{id} = y.\text{id}$ states that x and y denote the same node (entity).
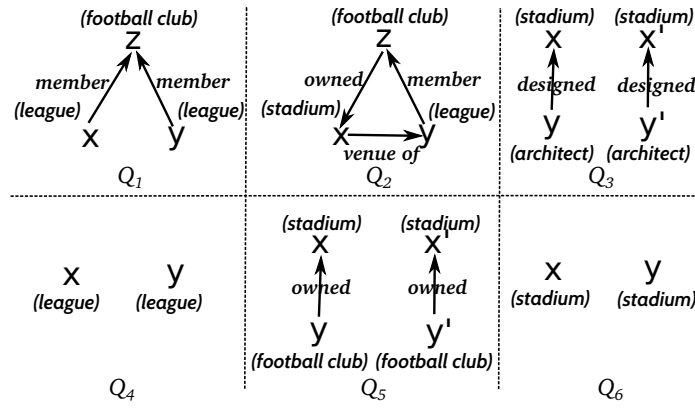
Figure 5.2: Graph patterns of GEDs

In contrast to [FL17], GEDs support inequality literals $x.A \neq c$, $x.A \neq y.B$ and $x.\text{id} \neq y.\text{id}$. These allow us to reduce false positives in object identification, *i.e.,* entities that do not match. For example, if x and y are male and female, respectively, then x and y may not be identified to be the same person, *i.e.,* $x.\text{id} \neq y.\text{id}$.

Example 5.3: Recall the data quality rules φ_1–φ_4 introduced in Examples 5.1 and 5.2. These rules can be expressed as GEDs by using the graph patterns shown in Fig. 5.2 as follows.

(1) $\varphi_1 = Q_1[x,y,z](\emptyset \to x.\text{country} = y.\text{country})$. It states that if a football club z is a member of two (domestic) leagues x and y, then x and y must be based in the same country. Here X is $\emptyset$.

(2) $\varphi_2 = Q_2[x,y,z](\emptyset \to x.\text{country} = y.\text{country})$. It says that if a football club z participates in a league y, z has a home ground stadium x and if x is used as a venue for y, then x and y must be in the same country. Here Q_2 is a cyclic pattern.

(3) $\varphi_3 = Q_3[x,x',y,y'](X_3 \to x.\text{id} = x'.\text{id})$, where X_3 consists of id literal $y.\text{id} = y'.\text{id}$ and variable literal $x.\text{capacity} = x'.\text{capacity}$. It says that stadiums x and x' can be identified by their capacity attributes and the ids of their architects y and y'.

(4) $\varphi_4 = Q_4[x,y](X_4 \to x.\text{id} = y.\text{id})$, where $X_4 = \{x.\text{name} = y.\text{name}, x.\text{country} = y.\text{country}\}$. It states that league entities x and y can be identified by their name and country attributes.

We define a set $\Sigma = \{\varphi_i \mid i \in [1,6]\} \cup \{\varphi_5'\}$ by including three GEDs below, which are also defined with the patterns in Fig. 5.2:

(5) $\varphi_5 = Q_5[x, x', y, y'](X_5 \to x.\text{id} = x'.\text{id})$, where $X_5 = \{y.\text{id} = y'.\text{id}, x.\text{opened} = x'.\text{opened}\}$. It says that a stadium can be identified by its date opened and the id of its owner (football club).

(6) $\varphi_5' = Q_5[x, x', y, y'](X_5' \to y.\text{id} = y'.\text{id})$, where $X_5' = \{x.\text{id} = x'.\text{id}, y.\text{name} = y'.\text{name}\}$. Conversely, each football club is uniquely identified by its name and the id of its stadium.

(7) $\varphi_6 = Q_6[x, y](X_6 \to x.\text{id} \neq y.\text{id})$, where $X_6 = \{x.\text{status} = \text{"working"}, y.\text{status} = \text{"demolished"}\}$. It states that two stadium entities x and y cannot refer to the same one when they have status of "working" and "demolished", respectively.

Note that φ_5 and φ_5' are recursively defined: to identify stadiums, we check the identities of their football club's, and vice versa.   □

Semantics. Consider a GED $\varphi = Q[\bar{x}](X \to Y)$, a match $h(\bar{x})$ of Q in a 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 $x.A = c$, attribute A *exists* at node $v = h(x)$, and $v.A = c$; (b) when l is $x.A = y.B$, attributes A and B *exist* at $v = h(x)$ and $v' = h(y)$, respectively, and $v.A = v'.B$; and (c) when l is $x.\text{id} = y.\text{id}$, $h(x)$ and $h(y)$ denote the same node; hence, they have the same set of attributes and edges. Similarly we define $h(\bar{x}) \models l$ when l is an inequality literal, *e.g.*, when l is $x.A \neq c$, $h(\bar{x}) \models l$ if attribute A *exists* at $v = h(x)$, 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 $\emptyset$, then $h(\bar{x}) \models X$ for any match $h(\bar{x})$ of Q in G; 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* GED φ, denoted by $G \models \varphi$, if *for all* matches $h(\bar{x})$ of Q in G, $h(\bar{x}) \models X \to Y$. A graph G *satisfies* a set Σ of GEDs if for all $\varphi \in \Sigma$, $G \models \varphi$, *i.e.,* G satisfies every GED in Σ.

Example 5.4: Recall graph G of Fig. 5.1 and the GEDs of Example 5.3. Here $G \models \varphi_6$, since s_4 and s_2, the only stadiums having status attributes, have different id's. Similarly, G satisfies φ_4, φ_5 and φ_5'. In contrast, $G \not\models \varphi_1$, as $h(x).\text{country} \neq h(y).\text{country}$ at match $h : x \mapsto c_1, y \mapsto c_2$ and $z \mapsto f_1$. Similarly, $G \not\models \varphi_2$ and $G \not\models \varphi_3$.   □

Special cases. We highlight the following special cases of GEDs.

(1) GFDs. Following [FWX16], we refer to GEDs without id literals as GFDs, *i.e.,* $Q[\bar{x}](X \to Y)$ in which neither X nor Y contains $x.\text{id} = y.\text{id}$ or $x.\text{id} \neq y.\text{id}$. In Exam-

ple 5.3, φ_1 and φ_2 are GFDs.

As shown in [FL17], GFDs extend CFDs, which are widely used in repairing relational data [FG12]. GFDs support *data repairing* by correcting attribute values along the same lines as CFDs, *e.g.,* φ_1 and φ_2 can correct the country values of c_1, c_2, and s_3 in G of Fig. 5.1.

(2) Keys. We say that a pattern $Q_2[\bar{y}]$ is a *copy* of $Q_1[\bar{x}]$ via an isomorphism $f\colon \bar{x} \mapsto \bar{y}$ if $Q_2[\bar{y}]$ is $Q_1[\bar{x}]$ with variables renamed by f. That is, let $Q_1[\bar{x}] = (V_{Q1}, E_{Q1}, L_{Q1})$ and $Q_2[\bar{y}] = (V_{Q2}, E_{Q2}, L_{Q2})$. Then for each $x \in \bar{x}$, $L_{Q1}(x) = L_{Q2}(f(x))$; and $e = (x_1, x_2)$ is an edge in E_{Q_1} if and only if $e' = (f(x_1), f(x_2))$ is an edge in E_{Q_2}, and $L_{Q1}(e) = L_{Q2}(e')$. We assume *w.l.o.g.* that $\bar{x}$ and $\bar{y}$ are disjoint.

A *key* is a GED of the form $Q[\bar{z}](X \rightarrow x_0.\mathrm{id} = y_0.\mathrm{id})$, where (a) $Q[\bar{z}]$ is composed of patterns $Q_1[\bar{x}]$ and $Q_2[\bar{y}]$, and $Q_2[\bar{y}]$ is a copy of $Q_1[\bar{x}]$ via an isomorphism $f\colon \bar{x} \mapsto \bar{y}$, (b) $\bar{z}$ consists of $\bar{x}$ followed by $\bar{y}$, and (c) $x_0 \in \bar{x}$ and $y_0 = f(x_0)$ are two designated nodes in Q.

For instance, φ_3, φ_4, φ_5 and φ_5' of Example 5.3 are keys.

Keys are used to identify vertices that refer to the same entity, *i.e., object identification* [FFTD15]. For instance, φ_3 and φ_5 help us identify stadiums, and φ_4 and φ_5' allow us match football leagues. As shown in Example 5.3, keys may be recursively defined.

*(3) Forbidding constraints (*FCs*)*. An FC is a GED $Q[\bar{x}](X \rightarrow x.\mathrm{id} \neq y.\mathrm{id})$, to deduce node pairs that should not be identified. For instance, φ_6 tells us when two stadiums should not be matched.

Summing up, GFDs help us repair inconsistent attribute values like CFDs. Keys are used in object identification to find pairs of nodes in the graph that refer to the same real-life entity. FCs are negative rules and 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.,* [FG12] for a survey).

Indeed, GEDs subsume the keys of Chapter 3 and GFDs of [FWX16]. As we aim to clean graphs by using GEDs in this chapter, a chase-based method (to be shown in Section 5.2) is developed to derive certain fixes, which extends the chase of Section 3.2, *i.e.,* entity matching with keys, by incorporating both data repairing and object identification into a single process. Note that NGDs of Chapter 4 cannot be expressed by GEDs since GEDs support neither arithmetic expressions nor comparison predicates beyond '=' and '$\neq$'.

5.2   Certain Fixes with the Chase

Consider a graph $G = (V, E, L, F_A)$ and a finite set Σ of GEDs such that $G \not\models \Sigma$, *i.e.,* there exist inconsistencies in G as violations of the GEDs of Σ. We want to clean G by fixing the inconsistencies, using the GEDs of Σ as data quality rules. In addition, we want to assure that the fixes are certain based on a block Γ of ground truth.

We first present fixes by revising the chase (Section 5.2.1). We then propose cleaning method Analogist with certain fixes (Section 5.2.2).

5.2.1   The Chase Revised

We start with a representation of (candidate) fixes to inconsistencies. We then revise the chase (cf. [AHV95]) for Analogist to deduce fixes.

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

Intuitively, for each node $y \in [x]_{\mathsf{Eq}}$, the pair (x, y) is a match for object identification, *i.e.,* $x.\mathrm{id} = y.\mathrm{id}$. For each $y.B \in [x.A]_{\mathsf{Eq}}$, $x.A = y.B$, and if $c \in [x.A]_{\mathsf{Eq}}$, $x.A$ has value c, for data repairing.

We use another relation NEq to keep track of entities that should not be matched. Here $[x]_{\mathsf{NEq}}$ includes nodes y such that $x.\mathrm{id} \neq y.\mathrm{id}$, and $[x.A]_{\mathsf{NEq}}$ includes constants d and attributes $y.B$ such that $x.A \neq d$ and $x.A \neq y.B$. This relation is symmetric and "transitive" via Eq, *e.g.,* for any $z \in [x]_{\mathsf{Eq}}$ and $w \in [x]_{\mathsf{NEq}}$, $z.\mathrm{id} \neq w.\mathrm{id}$.

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

We will deduce sequences of fixes $(\mathsf{Eq}_i, \mathsf{NEq}_i)$ by applying GEDs of Σ. The initial Eq_0 is the block Γ of *ground truth of G*, represented as an equivalence relation as above, in which $[x]_{\mathsf{Eq}}$ and $[x.A]_{\mathsf{Eq}}$ involve only nodes x and attributes $x.A$ in G. The ground truth may be validated by asking domain experts, crowd-sourcing, or using highly accurate knowledge bases as reference data. In this chapter we assume the availability of ground truth, and focus on how to deduce (certain) fixes to errors in G, from Γ by using the rules of Σ. For simplicity we assume an initial $\mathsf{NEq}_0 = \emptyset$ to start with.

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 and y are identified, we merge x and y into a single node, denoted by x_{Eq}, which

- $\circ$ inherits attributes of both $F_A(x)$ and $F_A(y)$ in G; and
- $\circ$ retains all edges pertaining to x or y, *i.e.,* if $e_1 = (x,z)$ is an edge in G, then $e_1' = (x_{\mathsf{Eq}},z)$ is an edge carrying the same label $L(e)$; similarly for edges $e_2(z,x)$, $e_3(y,z)$ and $e_4(z,y)$.

(2) For each $[x.A]_{\mathsf{Eq}}$, if $c \in [x.A]_{\mathsf{Eq}}$, we generate necessary new attribute $x.A$ and repairs $x.A$ with correct value c:

- $\circ$ add attribute A to x_{Eq} if $x.A$ does not exist in G, and
- $\circ$ let $x_{\mathsf{Eq}}.A = c$ no matter whether $x.A$ has a value or not.

(3) For each $[x.A]_{\mathsf{Eq}}$ and $y.B \in [x.A]_{\mathsf{Eq}}$, we equalize $x.A$ and $y.B$:

- $\circ$ add $x.A$ (resp. $y.B$) to x_{Eq} (resp. y_{Eq}) if it does not exist; and
- $\circ$ let $x_{\mathsf{Eq}}.A = y_{\mathsf{Eq}}.B = c$ if there exists $c \in [x.A]_{\mathsf{Eq}}$; otherwise let $x_{\mathsf{Eq}}.A = y_{\mathsf{Eq}}.B = \#$, denoting value to be assigned.

Here the symbol $\#$ indicates that an attribute value is yet to be determined, *i.e.,* instantiated, by some subsequent fixes in Eq, acting like labeled nulls. Note that $\#$ can be replaced by any constant from Eq by following (2) or (3) above, where the scope of each instantiation is limited to a single attribute. That is, the $\#$'s associated with different attributes could be possibly assigned different values.

The process proceeds until all $[x]_{\mathsf{Eq}}$ 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 $y.B = x.A$) at the same time, and may generate and enrich new attributes.

The chase. We compute fixes by chasing graph G with GEDs of Σ, starting with the block Γ of ground truth, *i.e.,* Eq_0. The chase is a classical tool in the relational database theory [AHV95]. It was recently revised for chasing a graph by GEDs [FL17]. Below we further extend it and make it a tool for cleaning graphs with (certain) fixes.

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

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

Here $\varphi = Q[\bar{x}](X \to Y)$ is a GED in Σ, and $h(\bar{x})$ is a match of pattern Q in the repair G_{Eq} by Eq, satisfying conditions (1)-(2) 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 $x.\mathsf{id} = y.\mathsf{id}$,

then $h(y) \in [h(x)]_{\mathsf{Eq}}$; if l is $x.A = c$, then $c \in [h(x).A]_{\mathsf{Eq}}$; and if l is $x.A = y.B$, then $h(y).B \in [h(x).A]_{\mathsf{Eq}}$; similarly for inequality literals to be deduced from NEq; and

(2) either Eq' extends Eq or NEq' extends NEq by adding a fix:

(2.1) Eq' extends Eq by instantiating one equality literal $l \in Y$, if any; more specifically, l and Eq' satisfy one of the following conditions:

(a) if l is $x.A = c$ and $c \notin [h(x).A]_{\mathsf{Eq}}$, then Eq' extends Eq by (a) including a new equivalence class $[h(x).A]_{\mathsf{Eq}'}$ if $h(x).A$ is not in Eq, and (b) adding c to $[h(x).A]_{\mathsf{Eq}'}$;

(b) if l is $x.A = y.B$ and $h(y).B \notin [h(x).A]_{\mathsf{Eq}}$, then Eq' extends Eq by adding (a) $[h(x).A]_{\mathsf{Eq}'}$ if $h(x).A$ is not in Eq; similarly for $[h(y).B]_{\mathsf{Eq}'}$; and (b) $h(y).B$ to $[h(x).A]_{\mathsf{Eq}'}$; and

(c) if l is $x.\mathsf{id} = y.\mathsf{id}$ and $h(y) \notin [h(x)]_{\mathsf{Eq}}$, then Eq' extends Eq by adding $h(y)$ to $[h(x)]_{\mathsf{Eq}'}$. Moreover, for each attribute A of $h(y)$ in G_{Eq}, add $[h(x).A]_{\mathsf{Eq}'}$ if it does not yet exist, and add $h(y).A$ to $[h(x).A]_{\mathsf{Eq}'}$. That is, when two nodes are identified, so are their corresponding attributes.

We compute the equivalence relation of Eq', also denoted by Eq', by making it reflexive, symmetric and transitive.

(2.2) NEq' extends NEq by instantiating an inequality literal $l \in Y$, if any, *i.e.*, $x.A \neq c$, $x.A \neq y.B$ and $x.\mathsf{id} \neq y.\mathsf{id}$ along the same lines as in (2.1) above. For instance, 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 $[h(x)]_{\mathsf{NEq}'}$.

Example 5.5: Recall graph G, ground truth Γ of Example 5.1, key φ_3, GFD φ_2 and FC φ_6 of Example 5.3. Initially $[x]_{\mathsf{Eq}_0} = \{x\}$ 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 $x.A$, where $\Gamma(x.A)$ (resp. $\Gamma(y.B)$) is the confirmed value of $x.A$ (resp. $y.B$) in Γ if exists . Consider three chase steps.

(1) $(\mathsf{Eq}_0, \mathsf{NEq}) \Rightarrow_{(\varphi_3, h_3)} (\mathsf{Eq}_1, \mathsf{NEq})$, where (a) $\mathsf{NEq} = \emptyset$; (b) $h_3 \colon x \mapsto s_1, x' \mapsto s_2, y \mapsto a$, $y' \mapsto a$; and (c) Eq_1 extends Eq_0 by letting $[s_1]_{\mathsf{Eq}_1} = [s_2]_{\mathsf{Eq}_1} = \{s_1, s_2\}$, $[s_1.\mathsf{alias}]_{\mathsf{Eq}_1} = [s_2.\mathsf{alias}]_{\mathsf{Eq}_1} = \{s_1.\mathsf{alias}, s_2.\mathsf{alias}\}$, and $[s_1.B]_{\mathsf{Eq}_1} = [s_2.B]_{\mathsf{Eq}_1} = \{s_1.B, s_2.B, \Gamma(s_2.B)\}$ for attribute B in $\{\mathsf{name}, \mathsf{capacity}, \mathsf{country}, \mathsf{status}\}$.

The chase step computes fixes in Eq_1, which (a) identify stadiums s_1 and s_2, along with their edges and attributes; and (b) enrich attributes, *e.g.,* $s_1.\mathsf{country}$ by taking "England" from $\Gamma(s_2.\mathsf{country})$. Moreover, s_1 and s_2 are merged into a single $s_{\mathsf{Eq}1}$ in the repair G_{Eq_1}.

(2) $(\mathsf{Eq}_1, \mathsf{NEq}) \Rightarrow_{(\varphi_2, h_2)} (\mathsf{Eq}_2, \mathsf{NEq})$, where (a) h_2 is a match of Q_2 in G_{Eq_1}: $x \mapsto s_{\mathsf{Eq}_1}$, $y \mapsto c_1$ and $z \mapsto f_1$; and (b) Eq_2 extends Eq_1 with $[s_1.\text{country}]_{\mathsf{Eq}_2} = [s_2.\text{country}]_{\mathsf{Eq}_2} = [c_1.\text{country}]_{\mathsf{Eq}_2} = \{s_1.\text{country}, s_2.\text{country}, c_1.\text{country}, \text{``England''}\}$.

This step propagates "England" to attribute $c_1.\text{country}$, by letting $c_1.\text{country} =$ "England" in the repair G_{Eq_2} of G.

(3) $(\mathsf{Eq}_0, \mathsf{NEq}) \Rightarrow_{(\varphi_6, h_6)} (\mathsf{Eq}_0, \mathsf{NEq}')$, where (a) h_6: $x \mapsto s_4, y \mapsto s_2$; and (b) NEq' extends NEq with $[s_4]_{\mathsf{NEq}'} = \{s_2\}$ and $[s_2]_{\mathsf{NEq}'} = \{s_4\}$.

By applying FC φ_6, this step finds that stadiums s_2 and s_4 do not match ($s_2.\text{status}$ is demolished while $s_4.\text{status}$ is working), although they have the same name and are owned by the same club. □

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

- ○ *label conflict*: there exists node $y \in [x]_{\mathsf{Eq}'}$ such that $L(x) \neq L(y)$, *i.e.,* Eq' attempts to merge nodes with distinct labels;

- ○ *attribute conflict*: there exists $y.B \in [x.A]_{\mathsf{Eq}'}$ such that $x.A = c$ and $y.B = d$ for $c \in [x.A]_{\mathsf{Eq}'} \neq d \in [y.B]_{\mathsf{Eq}'}$, *i.e.,* Eq' assigns distinct values to the same attribute; or

- ○ either $[x]_{\mathsf{Eq}'} \cap [x]_{\mathsf{NEq}'} \neq \emptyset$ or $[x.A]_{\mathsf{Eq}'} \cap [x.A]_{\mathsf{NEq}'} \neq \emptyset$. That is, 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, then $x.A = y.A$, *i.e.,* the attributes of $x_{\mathsf{Eq}'}$ are well-defined.

Chasing. We deduce fixes by a *chasing sequence* ρ of G by (Σ, Γ):

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

where $\mathsf{Eq}_0 = \Gamma$, $\mathsf{NEq}_0 = \emptyset$, and for all $i \in [0, k-1]$, there exist a GED $\varphi = Q[\bar{x}](X \to Y)$ in Σ and a match h of Q in the repair G_{Eq_i} such that $(\mathsf{Eq}_i, \mathsf{NEq}_i) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}_{i+1}, \mathsf{NEq}_{i+1})$ is a valid chase step.

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

Example 5.6: A chasing sequence ρ consists of the chase steps (1) and (2) given in Example 5.5, followed by steps (a)–(f) below:
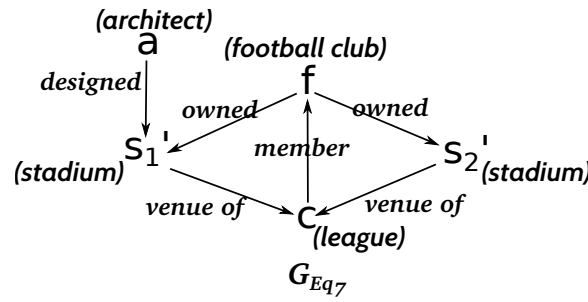
Figure 5.3: The repair of graph G

(a) $(\mathsf{Eq}_2, \mathsf{NEq}) \Rightarrow_{(\varphi_1, h_1)} (\mathsf{Eq}_3, \mathsf{NEq})$ by GFD φ_1 such that Eq_3 extends Eq_2 by including "England" in $[c_2.\text{country}]_{\mathsf{Eq}_3}$;

(b) by enforcing key φ_4, Eq_4 is derived from Eq_3 by merging $[c_1]_{\mathsf{Eq}_3}$ and $[c_2]_{\mathsf{Eq}_3}$ along with their attributes;

(c) by applying GFD φ_2, $(\mathsf{Eq}_4, \mathsf{NEq}) \Rightarrow_{(\varphi_2, h'_2)} (\mathsf{Eq}_5, \mathsf{NEq})$, in which "England" is added to $[s_3.\text{country}]_{\mathsf{Eq}_5}$;

(d) by enforcing key φ'_5, football clubs f_1 and f_2 are identified by adding f_1 (resp. f_2) to $[f_2]_{\mathsf{Eq}_6}$ (resp. $[f_1]_{\mathsf{Eq}_6}$);

(e) stadiums s_3 and s_4 are identified by key φ_5, by merging their equivalence classes and attributes in Eq_7; and

(f) NEq is extended to NEq' as step (3) of Example 5.5.

The sequence ρ is terminal, and ends up with $(\mathsf{Eq}_7, \mathsf{NEq}')$. The repair G_{Eq_7} of G by Eq_7 is depicted in Fig. 5.3, in which all the country attributes are assigned value "England", and moreover, leagues c_1 and c_2 in graph G (resp. football clubs f_1 and f_2, stadiums s_1 and s_2, and s_3 and s_4) are identified as c (resp. f, s'_1, and s'_2). The sequence interleaves data repairing and object identification. $\square$

Chasing sequence ρ terminates in one of the following two cases.

(a) No GEDs in Σ can be further applied. If so, we say that ρ is *valid*, and refer to $(\mathsf{Eq}_k, \mathsf{NEq}_k, G_{\mathsf{Eq}_k})$ as *its result*. One can verify that in a valid ρ, for all $i \in [0, k]$, G_{Eq_i} is well-defined.

(b) Either Eq_0 is inconsistent or there exist φ, h, Eq_{k+1}, NEq_{k+1} such that $(\mathsf{Eq}_k, \mathsf{NEq}_k) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1})$ 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 of graph G by (Σ, Γ) can always terminate with the same fixes.

Following the relational database theory [AHV95], we say that chasing with GEDs has the *Church-Rosser property* if for all Σ, Γ and G, all terminal chasing sequences of G by (Σ, Γ) converge at the same result, regardless of in what order the GEDs in Σ are applied.

Theorem 5.1: *For any graph G, any set Σ of* GEDs*, and any block Γ of ground truth of G, all chasing sequences of G by (Σ, Γ) are terminal, and moreover, converge at the same result.* $\qquad\qquad\square$

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

For instance, $(\mathsf{Eq}_7, \mathsf{NEq}', G_{\mathsf{Eq}_7})$ given in Fig. 5.3 is the result of chasing graph G of Fig. 5.1 by Σ of Example 5.3 and the ground truth of Fig. 5.1, no matter what rules of Σ are used and how we apply them.

Proof of Theorem 5.1: We prove the two statements of Theorem 5.1, respectively, one by one.

(1) All chasing sequences of G by (Σ, Γ) are finite. Consider a chasing sequence $\rho = (\mathsf{Eq}_0, \mathsf{NEq}_0), \ldots, (\mathsf{Eq}_k, \mathsf{NEq}_k)$ of G by (Σ, Γ). To analyze the length of ρ, we first study the maximum cardinality of $\|\mathsf{Eq}_i\|$ and $\|\mathsf{NEq}_i\|$, which indicate the maximum number of chase steps since each step extends either $|\mathsf{Eq}|$ or $|\mathsf{NEq}|$. One can verify the following by extending the analysis of [FL17]: (a) $\|\mathsf{Eq}_i\| \leq 4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$, and (b) $\|\mathsf{NEq}_i\| \leq (4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|)^2$. Note that an element z can appear in distinct $[x]_{\mathsf{NEq}_i}$ and $[y]_{\mathsf{NEq}_i}$. Hence there exist at most $4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$ many $[x]_{\mathsf{NEq}_i}$, and each $[x]_{\mathsf{NEq}_i}$ contains at most $4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$ many elements.

We now study the length $|\rho|$ of ρ. In a chase step $(\mathsf{Eq}_{i-1}, \mathsf{NEq}_{i-1}) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}_i, \mathsf{NEq}_i)$, Eq_i (resp. NEq_i) extends Eq_{i-1} (resp. NEq_{i-1}) by adding one fix l (Section 5.2). Assume *w.l.o.g.* that l is $x.A \neq c$; the proofs for other cases are similar. Then $\|\mathsf{NEq}_i\| \geq \|\mathsf{NEq}_{i-1}\| + 2$. Indeed, (a) when neither c nor $x.A$ is in NEq_{i-1}, $\|\mathsf{NEq}_i\| = \|\mathsf{NEq}_{i-1}\| + 2$, since we need to create two new classes; (b) when only one of c and $x.A$ exists in NEq_{i-1}, $\|\mathsf{NEq}_i\| = \|\mathsf{NEq}_{i-1}\| + 2$, (c) when both c and $x.A$ are in NEq_{i-1}, $\|\mathsf{NEq}_i\| \geq \|\mathsf{NEq}_{i-1}\| + 2$, since we need to add at least one element to $[x.A]_{\mathsf{NEq}}$, and $[c]_{\mathsf{NEq}}$, respectively. Since $\|\mathsf{NEq}_i\| \leq (4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|)^2$, to extend NEq we need at most $\frac{(4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|)^2}{2} = 2 \cdot (2 \cdot |G| \cdot |\Sigma| + |\Gamma|)^2$ chase steps. Similarly, we need at most $8 \cdot |G| \cdot |\Sigma| + 4 \cdot |\Gamma|$ steps to extend Eq. Thus $|\rho| \leq k \leq (8 \cdot |G| \cdot |\Sigma| + 4 \cdot |\Gamma|) \cdot 2 \cdot (2 \cdot |G| \cdot |\Sigma| + |\Gamma|)^2 \leq 8 \cdot (2 \cdot |G| \cdot |\Sigma| + |\Gamma|)^3$. Hence any chasing sequence ρ is finite.

(2) Church-Rosser property. Assume by contradiction that there exist two terminal chasing sequences $\rho_1 = (\mathsf{Eq}_0, \mathsf{NEq}_0)$, ..., $(\mathsf{Eq}_k, \mathsf{NEq}_k)$ and $\rho_2 = (\mathsf{Eq}_0, \mathsf{NEq}_0)$, ..., $(\mathsf{Eq}'_l, \mathsf{NEq}'_l)$ such that they have different results. Assume *w.l.o.g.* that $(\mathsf{Eq}_0, \mathsf{NEq}_0)$ is consistent, since otherwise both ρ_1 and ρ_2 are invalid and end up with the same result. Then if $(\mathsf{Eq}_k, \mathsf{NEq}_k)$ and $(\mathsf{Eq}'_l, \mathsf{NEq}'_l)$ are different, then we have one of the following cases: (1) $\mathsf{Eq}_k \setminus \mathsf{Eq}'_l \neq \emptyset$; (2) $\mathsf{Eq}'_l \setminus \mathsf{Eq}_k \neq \emptyset$; (3) $\mathsf{NEq}_k \setminus \mathsf{NEq}'_l \neq \emptyset$; and (4) $\mathsf{NEq}'_l \setminus \mathsf{NEq}_k \neq \emptyset$. To show that these lead to contradiction, we first show the following lemma.

Lemma 5.2: *If ρ_1 is valid, and $\mathsf{Eq}'_l \setminus \mathsf{Eq}_k \neq \emptyset$ or $\mathsf{NEq}'_l \setminus \mathsf{NEq}_k \neq \emptyset$, then ρ_1 is not terminal.*   □

Proof of Lemma 5.2: When either $S = \mathsf{Eq}'_l \setminus \mathsf{Eq}_k \neq \emptyset$ or $N = \mathsf{NEq}'_l \setminus \mathsf{NEq}_k \neq \emptyset$, we show that there exist a GED $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ, a match h of Q in G_{Eq_k}, and Eq_{k+1} and NEq_{k+1} such that $(\mathsf{Eq}_k, \mathsf{NEq}_k) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1})$ is a valid chase step. Hence ρ_1 is not terminal.

We define φ, h, Eq_{k+1}, NEq_{k+1} as follows. Suppose that $(\mathsf{Eq}'_j, \mathsf{NEq}'_j)$ $(1 \leq j \leq l)$ is the first case in ρ_2 such that $S \cap \mathsf{Eq}'_j \neq \emptyset$ or $N \cap \mathsf{NEq}'_j \neq \emptyset$, and $(\mathsf{Eq}'_{j-1}, \mathsf{NEq}'_{j-1}) \Rightarrow_{(\varphi_j, h_j)}$ $(\mathsf{Eq}'_j, \mathsf{NEq}'_j)$ is the corresponding chase step in ρ_2, where $\varphi_j = Q_j[\bar{x}_j](X_j \rightarrow Y_j)$ is a GED in Σ, h_j is a match of Q_j in $G_{\mathsf{Eq}'_{j-1}}$, and $h_j(\bar{x}_j) \models X_j$. Since both sequences start with $(\mathsf{Eq}_0, \mathsf{NEq}_0)$, we have that $j > 0$, $\mathsf{Eq}'_{j-1} \subseteq \mathsf{Eq}_k$ and $\mathsf{NEq}'_{j-1} \subseteq \mathsf{NEq}_k$. We let $\varphi = \varphi_j$, and define h as follows: for each node x in Q_j, if $h_j(x) = y_{\mathsf{Eq}'_{j-1}}$ (*i.e.*, $[y]_{\mathsf{Eq}'_{j-1}}$, see Section 5.2 for the definition of the repair of Eq on G), then $h(x) = y_{\mathsf{Eq}_k}$ (denoting $[y]_{\mathsf{Eq}_k}$). Moreover, Eq_{k+1} and NEq_{k+1} can be extended accordingly. Similar to the proof in [FL17], we can verify that $(\mathsf{Eq}_k, \mathsf{NEq}_k) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1})$ is a valid chase step.

Using Lemma 5.2, we prove statement (2) by considering two cases.

(a) Both ρ_1 and ρ_2 are valid but have different results. By Lemma 5.2, we must have that $\mathsf{Eq}'_l \setminus \mathsf{Eq}_k = \emptyset$, $\mathsf{Eq}_k \setminus \mathsf{Eq}'_l = \emptyset$, $\mathsf{NEq}'_l \setminus \mathsf{NEq}_k = \emptyset$, and $\mathsf{NEq}_k \setminus \mathsf{NEq}'_l = \emptyset$, since otherwise one of the two sequences is not terminal. But then $\mathsf{Eq}_k = \mathsf{Eq}'_l$ and $\mathsf{NEq}_k = \mathsf{NEq}'_l$, *i.e.*, ρ_1 and ρ_2 have the same result, a contradiction to the assumption.

(b) One of them is valid and the other is not. Assume *w.l.o.g.* that ρ_1 is valid but ρ_2 is not. We show that ρ_1 must be invalid as well. Since ρ_2 is invalid, there exist a GED $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ and a match h of Q in $G_{\mathsf{Eq}'_l}$ such that $(\mathsf{Eq}'_l, \mathsf{NEq}'_l) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}'_{l+1}, \mathsf{NEq}'_{l+1})$, $h(\bar{x}) \models X$ and Eq'_{l+1} is inconsistent in $G_{\mathsf{Eq}'_l}$, where either Eq'_{l+1} extends Eq'_l or NEq'_{l+1} extends NEq'_l by adding a fix (u, v) from Y. By

Lemma 5.2, $\mathsf{Eq}'_l \subseteq \mathsf{Eq}_k$ and $\mathsf{NEq}'_l \subseteq \mathsf{NEq}_k$. As in the proof of Lemma 5.2, one can verify that $(\mathsf{Eq}_k, \mathsf{NEq}_k) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}_{k+1}, \mathsf{NEq}_{k+1})$ is a chase step, where Eq_{k+1} extends Eq_k or NEq_{k+1} extends NEq_k by adding (u,v). Since Eq'_{l+1} has conflicts and $\mathsf{Eq}'_l \subseteq \mathsf{Eq}_k$, Eq_{k+1} also has conflicts, which contradicts to the assumption that ρ_1 is valid. $\qquad \square$

As opposed to [FL17], (a) a chase step by GED $\varphi = Q[\bar{x}](X \to Y)$ 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) while [FL17] does not study graph cleaning, we revise the chase to identify objects and repair attributes, and (c) Theorem 5.1 extends the result of [FL17] to GEDs with inequality literals.

5.2.2 A Method for Cleaning Graphs

Based on the chase, we propose method Analogist for cleaning graphs with certain fixes. We start with a notion of certain fixes.

Certain fixes. When $\mathsf{Chase}(G, \Sigma, \Gamma) \neq \bot$, we say that (Σ, Γ) is *consistent for G*. Given consistent (Σ, Γ), all terminal chasing sequences of G by (Σ, Γ) are valid, and end up at the same result $\mathsf{Chase}(G, \Sigma, \Gamma) = (\mathsf{Eq}, \mathsf{NEq}, G_{\mathsf{Eq}})$ by Theorem 5.1. In this case, we refer to Eq and G_{Eq} as *the certain fixes* and *the repair* of graph G with (Σ, Γ), respectively.

Intuitively, a chasing sequence ρ deduces fixes from confirmed entity matches and attribute values by iteratively applying GEDs, starting from the block Γ of ground truth. The fixes in Eq are *certain* since they are the logical consequences of (Σ, Γ). That is, as long as Σ and Γ are correct, then so are the certain fixes in Eq.

Graph cleaning. *Graph cleaning with certain fixes* is as follows.
   ○ *Input*: A graph G, a set Σ of GEDs and a block Γ of ground truth such that (Σ, Γ) is consistent.
   ○ *Output*: The certain fixes Eq and repair of G by (Σ, Γ).

We now present Analogist, a rule-based method for cleaning graphs. Given a graph G in an application, it works as follows. (a) It first discovers a set Σ of GEDs by employing discovery algorithms [FLL$^+$17], and solicit ground truth Γ by consulting experts or crowd-sourcing. (b) It validates the rules and ground truth by a consistency analysis (see Section 5.3). (c) It then finds certain fixes Eq and repair G_{Eq} by chasing G with (Σ, Γ).

While all fixes are guaranteed correct as long as Σ and Γ are validated, Analogist

may not fix all violations of Σ in G due to inadequate ground truth in Γ. In other words, $G_{\mathsf{Eq}} \not\models \Sigma$. Consider $G \not\models \varphi$ for GED $\varphi = Q[\bar{x}](X \to Y)$ in Σ, *i.e.,* there is a match $h(\bar{x})$ of Q in G such that $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$. It is possible that after the chase, $h(\bar{x})$ remains a match of Q in G_{Eq}, but Γ and Σ *may not have enough information* to cover all the literals of X at $h(\bar{x})$, and hence $h(\bar{x}) \not\models Y$ because we cannot deduce the true values of some $x.A$ in Y, *e.g.,* # in the cleaning process is not instantiated.

When this happens, Analogist interacts with the users and expands Γ with more ground truth, until all errors can be "covered" by (Σ, Γ) (see Section 5.3 for coverage analysis). Users may also opt to correct particular errors by providing relevant ground truth only.

We study consistency, cleaning and coverage analyses (Section 5.3) and give algorithms for step (c) of Analogist (Sections 5.4 and 5.5).

5.3 Fundamental Problems

We next study three fundamental problems for graph cleaning with certain fixes, compared to their relational counterparts [FLM$^+$12].

Consistency. We start with *the consistency problem*.
- *Input*: A set Σ of GEDs and a block Γ of ground truth.
- *Question*: Is (Σ, Γ) consistent for G?

It is to decide whether GEDs discovered and ground truth provided have no conflicts, *i.e.,* they are not dirty. The problem has the same complexity as its relational counterpart using extended CFDs [FLM$^+$12].

Theorem 5.3: *The consistency problem is* coNP-*complete.* $\qquad\qquad \square$

Proof: We first prove the upper bound, *i.e.,* coNP membership, of the consistency problem, and then show its coNP-hardness.

Upper bound. We give an NP algorithm that, given a graph G, a set Σ of GEDs, and a block Γ of ground truth, checks whether (Σ, Γ) is not consistent for G. It works as follows: (1) guess a chasing sequence $(\mathsf{Eq}_0, \mathsf{NEq}_0) = (\Gamma, \emptyset) \Rightarrow_{(\varphi_1, h_1)} \cdots \Rightarrow_{(\varphi_k, h_k)}$ $(\mathsf{Eq}_k, \mathsf{NEq}_k)$ of G by (Σ, Γ), where $k \leq 8 \cdot (2 \cdot |G| \cdot |\Sigma| + |\Gamma|)^3$; (2) for each $i \in [0, k-2])$, check whether $(\mathsf{Eq}_i, \mathsf{NEq}_i) \Rightarrow_{(\varphi_{i+1}, h_{i+1})} (\mathsf{Eq}_{i+1}, \mathsf{NEq}_{i+1})$ is a valid chase step; (3) if so, check whether $(\mathsf{Eq}_{k-1}, \mathsf{NEq}_{k-1}) \Rightarrow_{(\varphi_k, h_k)} (\mathsf{Eq}_k, \mathsf{NEq}_k)$ is invalid; if so, return true.

Here we do not construct $G_{\mathsf{Eq}_0}, \ldots, G_{\mathsf{Eq}_{k-1}}$. Instead, we first guess mappings

$h'_1, \ldots, h'_k$ in G, and then define $h_i(x) = [h'_i(x)]_{\mathsf{Eq}_{i-1}}$ (*i.e.*, $h'_i(x)_{\mathsf{Eq}_{i-1}}$; see Section 5.2 for $G_{\mathsf{Eq}_{i-1}}$) for $i \in [1, k]$. One can verify that h_i is a mapping from the pattern of φ_i to $G_{\mathsf{Eq}_{i-1}}$.

The correctness of the algorithm follows from Theorem 5.1 and its proof above. For its complexity, since $|\mathsf{Eq}| \leq 4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$ and $|\mathsf{NEq}| \leq (4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|)^2$, steps (2) and (3) are in PTIME. Thus the algorithm is in NP and the consistency problem is in coNP.

Lower bound. We show the coNP-hardness by reduction from the satisfiability problem for GEDs, which is coNP-complete [FL17]. The latter problem is to decide, given a set Σ_1 of GEDs, whether there exists a graph G such that $G \models \Sigma_1$, and moreover, for each $Q[\bar{x}](X \to Y)$ in Σ_1, Q has a match in G. Given Σ_1, we will construct a graph G, a set Σ of GEDs, and a block Γ of ground truth such that Σ_1 is satisfiable if and only if (iff) (Σ, Γ) is consistent.

The reduction makes use of the following characterization of the satisfiability problem for GEDs [FL17]: Σ_1 is satisfiable iff $\mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1) \neq \bot$, where Chase^0 is the chase of [FL17], and G_{Σ_1} is the *canonical graph* of Σ_1, defined as $G_{\Sigma_1} = (V_{\Sigma_1}, E_{\Sigma_1}, L_{\Sigma_1}, F_A^{\Sigma_1})$, which is a disjoint union of patterns in Σ_1 except that $F_A^{\Sigma_1}$ is empty.

We define G as G_Σ, Σ as Σ_1, and $\Gamma = \emptyset$. To show this makes a reduction, it suffices to prove that $\mathsf{Chase}(G, \Sigma, \emptyset) = \mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1)$. For if it holds, by the characterization above and the statement of the consistency problem, (Σ, Γ) is consistent for G iff $\mathsf{Chase}(G, \Sigma, \Gamma) \neq \bot$ iff $\mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1) \neq \bot$ iff Σ_1 is satisfiable.

It remains to show $\mathsf{Chase}(G, \Sigma, \emptyset) = \mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1)$. A close examination of Chase^0 reveals that $\mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1) \subseteq \mathsf{Chase}(G, \Sigma, \emptyset)$, since Chase extends Chase^0. Moreover, the GEDs in Σ are those defined in [FL17] and do not have inequality literals; in addition, $\Gamma = \emptyset$. Hence if $(\mathsf{Eq}, \mathsf{NEq}) \Rightarrow_{(\varphi, h)} (\mathsf{Eq}', \mathsf{NEq}')$ is a chase step in $\mathsf{Chase}(G, \Sigma, \emptyset)$, then $\mathsf{Eq} \Rightarrow_{(\varphi, h)} \mathsf{Eq}'$ is also a step in $\mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1)$. By the Church-Rosser property, we have that $\mathsf{Chase}(G, \Sigma, \emptyset) \subseteq \mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1)$. Hence $\mathsf{Chase}(G, \Sigma, \emptyset) = \mathsf{Chase}^0(G_{\Sigma_1}, \Sigma_1)$. □

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

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

Here Eq is the set of certain fixes in $\mathsf{Chase}(G, \Sigma, \Gamma)$.

This problem is to settle the complexity of cleaning graphs with certain fixes. We consider (a) the combined complexity, when graph G, GEDs Σ and ground truth Γ may all vary, and (b) the data complexity, when the GEDs are predefined and fixed, but G and Γ may vary. The problem was not studied for relations [FLM$^+$12].

Theorem 5.4: *For the certain fix problem, the combined complexity is* NP-*complete, and the data complexity is in* PTIME. $\square$

Proof: We first consider the upper bound including data complexity, followed by proving the NP-hardness.

Upper bound. We give an NP algorithm that, given a graph G, a set Σ of GEDs, a block Γ of ground truth, and a fix l, where (Σ, Γ) is consistent, checks whether $l \in \mathsf{Eq}$ or $l \in \mathsf{NEq}$. It works as follows: (1) guess a chasing sequence $(\mathsf{Eq}_0, \mathsf{NEq}_0) \Rightarrow_{(\varphi_1, h_1)} \ldots \Rightarrow_{(\varphi_k, h_k)} (\mathsf{Eq}_k, \mathsf{NEq}_k)$ of G by (Σ, Γ) such that $k \leq 8 \cdot (2 \cdot |G| \cdot |\Sigma| + |\Gamma|)^3$; (2) for each $i \in [0, k-1]$, check whether $(\mathsf{Eq}_i, \mathsf{NEq}_i) \Rightarrow_{(\varphi_{i+1}, h_{i+1})} (\mathsf{Eq}_{i+1}, \mathsf{NEq}_{i+1})$ is a valid chase step; if not, reject the guess; (3) check whether $l \in \mathsf{Eq}$ or $l \in \mathsf{NEq}$; if so, return true.

The correctness of the algorithm follows from Theorem 5.1 and that condition that (Σ, Γ) is consistent for G. For its complexity, since $|\mathsf{Eq}| \leq 4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$ and $|\mathsf{NEq}| \leq (4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|)^2$ (see the proof of Theorem 5.1), both steps (2) and (3) are in PTIME. Hence the algorithm is in NP, and the certain fix problem is in NP.

We show that the data complexity of the problem is in PTIME, by giving the following algorithm: (1) compute $\mathsf{Chase}(G, \Sigma, \Gamma)$; (2) check whether $l \in \mathsf{Eq}$ or $l \in \mathsf{NEq}$; if so, return true; otherwise, return false. The algorithms is correct by the statement of the certain fix problem. For its complexity, step (1) is in PTIME. Indeed, since Σ is fixed, we can enumerate all matches of patterns in Σ in G, and then compute $\mathsf{Chase}(G, \Sigma, \Gamma)$. Since $|\mathsf{Eq}| \leq 4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$, step (2) can also be done in PTIME. Hence the algorithm is in PTIME.

Lower bound. We prove the NP-hardness by reduction from the 3-colorability problem, which is NP-complete (cf. [GJ79]). The 3-colorability problem is to decide, given an undirected graph G_1, whether there exists a 3-coloring μ of G_1 such that for each edge (u, v) in G, $\mu(u) \neq \mu(v)$. The problem remains to be NP-complete when G_1 is a connected graph [GJS76].

Given a connected undirected $G_1 = (V, E)$, we construct a graph G, a set Σ of GFDs, and a block Γ of ground truth satisfying that (Σ, Γ) is consistent, and a fix l of the form
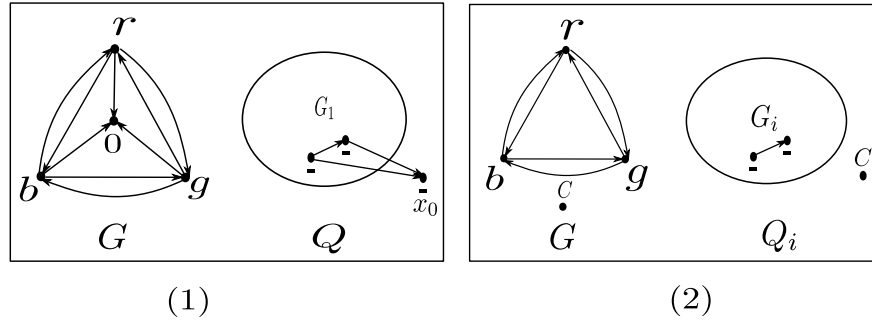
Figure 5.4: Graph patterns in the proofs

$u = v$ (proof for $u \neq v$ is similar). We show that $v \in [u]_{\text{Eq}}$ iff G_1 has a proper 3-coloring.

(1) The graph G is given in Fig. 5.4(1), which is to encode 3-coloring. More specifically, $G = (V_1, E_1, L_1, F_A^1)$, where (a) $V_1 = \{v_0, v_1, v_2, v_3\}$; it consists of three nodes to represent the three colors, and a designated node v_0 to handle the fix l; (b) $E_1 = \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_3, v_1), (v_2, v_3), (v_3, v_2), (v_1, v_0), (v_2, v_0), (v_3, v_0)\}$; these edges make v_1, v_2 and v_3 form a clique, and v_1, v_2 and v_3 have edges linking to v_0; (c) $L_1(v_1) = r$, $L_1(v_2) = g$, $L_1(v_3) = b$ for three colors, and $L_1(v_0) = 0$; and (d) F_A^1 is empty.

(2) The set Σ has only one GED $\varphi = Q[x_0, x_1, \ldots, x_m](\emptyset \rightarrow (x_0.A = 2))$, to encode the structure of G_1. As shown in Fig. 5.4 (1). $Q[\bar{x}] = (V_2, E_2, L_2)$, where $V_2 = V \cup \{v_0\}$, *i.e.,* the set of nodes in G pluses one node v_0; $E_2 = \{(u, v), (v, u) \mid (u, v) \in E\} \cup \{(v, v_0) \mid v \in V\}$, *i.e.,* each undirected edge (u, v) in G is encoded with two directed edges (u, v) and (v, u), and all nodes in V have an edge to v_0; each vertex $v_i \in V_2$ is labeled wildcard, *i.e.,* $L_2(v_i) = $ '_'.

(3) The block Γ is set to be empty. Since Γ is empty and φ in Σ has no id literal, it is easy to verify that (Σ, Γ) is consistent for G.

(4) The fix l is defined as $x_0.A = 2$.

We next show that $2 \in [x_0.A]_{\text{Eq}}$ iff G_1 has a proper 3-coloring.

($\Rightarrow$) Assume that $2 \in [x_0.A]_{\text{Eq}}$. Since Γ is empty, there exists one chase step on G. Thus there exists a match h of Q in G. Based on h, we can now deduce a 3-coloring μ of G_1 as follows: for each node $x \in V$, if $h(x) = v_1$, then $\mu(x) = r$; if $h(x) = v_2$, then $\mu(x) = g$; and if $h(x) = v_3$, then $\mu(x) = b$. Note that no node in $x \in V$ is mapped to v_0 by h, since v_0 has no outgoing edge. By the definitions of G, Q and h, one can verify

that μ is a proper 3-coloring of G_1.

($\Leftarrow$) Conversely, assume that μ is a proper 3-coloring of G_1. It suffices to show that there exists a chasing sequence $\rho = (\emptyset, \emptyset) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}_1, \emptyset)$ of G by $(\Sigma, \emptyset)$ such that $x_0.A = 2$ is in Eq_1. For if it holds, since (Σ, Γ) is consistent, $x_0.A = 2$ is deduced by $\mathsf{Chase}(G, \Sigma, \Gamma)$ and hence $2 \in [x_0.A]_{\mathsf{Eq}}$. The match h is defined as follows: $h(x_0) = v_0$; for each other node $x \in V_2$, if $\mu(x) = r$, then $h(x) = v_1$; if $\mu(x) = g$, then $h(x) = v_2$; if $\mu(x) = b$, then $h(x) = v_3$. One can verify that h is a match of Q in G. As a consequence, Eq_1 deduces $x_0.A = 2$.   □

Coverage. As remarked in Section 5.2, in practice we want to check whether all violations of Σ in G can be fixed with certainty by $\mathsf{Chase}(G, \Sigma, \Gamma)$. This motivates us to study *the coverage problem*.

   ○ *Input*: G, Σ and Γ as in the certain fix problem.
   ○ *Question*: Does $G_{\mathsf{Eq}} \models \Sigma$, for the repair G_{Eq} of G by (Σ, Γ)?

For relations, the data complexity was not studied, and the combined complexity of a stronger problem is coNP-complete [FLM$^+$12], to decide whether all instances of a relation schema can be fixed with certainty. We show that unless $\mathsf{P} = \mathsf{NP}$, 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 that can make polynomially many queries to an NP oracle in parallel, *i.e.,* all queries are formed before knowing the results of the others [Wag90]. It is known that $\mathsf{P}_\|^{\mathsf{NP}}$ is a subclass of $\Delta_2^p = \mathsf{P}^{\mathsf{NP}}$ [PZ82].

Theorem 5.5: *The combined complexity of the coverage problem is $\mathsf{P}_\|^{\mathsf{NP}}$-complete; and its data complexity is in* PTIME.   □

Proof: This proof is quite involved. We first prove the upper bound and show the data complexity of the coverage problem is in PTIME, and then prove the lower bound by reduction from the odd max true 3SAT problem.

Upper bound. We give a $\mathsf{P}_\|^{\mathsf{NP}}$ algorithm to check whether $G_{\mathsf{Eq}} \models \Sigma$. It works as follows: (1) check in parallel whether $u \in [v]_{\mathsf{Eq}}$ for all possible fixes $u = v$; if $u \in [v]_{\mathsf{Eq}}$, then add $u = v$ to Eq_1; (2) use the equivalence relation of Eq_1 to construct G_{Eq_1}; (3) check whether $G_{\mathsf{Eq}_1} \models \Sigma$; if so, return true; false otherwise.

   The correctness of algorithm follows from the definition of G_{Eq} and $\mathsf{Chase}(G, \Sigma, \Gamma)$. It computes correct Eq by checking all possible fixes, by Theorem 5.4. For the complexity of the algorithm, step (1) is in $\mathsf{P}_\|^{\mathsf{NP}}$, since there exists at most $(|G| + |\Sigma| + |\Gamma|)^2$ many fixes, and checking whether $u \in [v]_{\mathsf{Eq}}$ is in NP (see Theorem 5.4). Since there ex-

ist at most $(|G| + |\Sigma| + |\Gamma|)^2$ many fixes, step (2) can be done in PTIME. Step (3) calls an NP oracle to the validation problem for GEDs [FL17]. Note that the algorithm uses two rounds of parallel computations. It is known that using constant rounds of parallel computations is equivalent to using one round of parallel computation [BH91]. Hence the algorithm above is in $P_{||}^{NP}$.

We next show that the data complexity is in PTIME. We use the following algorithm to check whether $G_{Eq} \models \Sigma$: (1) compute $\mathsf{Chase}(G, \Sigma, \Gamma)$; (2) construct the repair G_{Eq} of G; (3) check whether $G_{Eq} \models \Sigma$; if so, return true; false otherwise.

The correctness of the algorithm follows from the statement of the coverage problem. For its complexity, step (1) is in PTIME because Σ is fixed, and we can enumerate in PTIME all matches of patterns in Σ in G. Since $|\mathsf{Eq}| \le 4 \cdot |G| \cdot |\Sigma| + 2 \cdot |\Gamma|$ (see the proof of Theorem 5.1), step (2) can be done in PTIME. Step (3) is in PTIME, since Σ is fixed, and we can enumerate in PTIME all matches of patterns in Σ in G_{Eq}. Therefore, the algorithm above is in PTIME. Hence the data complexity of the coverage problem is in PTIME.

Lower bound. For the combined complexity, we show that the problem is $P_{||}^{NP}$-hard by reduction from the odd max true 3SAT (OMT3) problem, which is $P_{||}^{NP}$-complete [Spa05]. The OMT3 problem is to decide, given a 3-CNF formula φ with variables $x_1, \ldots, x_n$, let $\mu(\bar{x})$ be a satisfying assignment of φ with the maximal number of variables set true, whether $\mu(\bar{x})$ sets an odd number of variables true. A 3-CNF formula has the form $C_1 \wedge \ldots \wedge C_l$, each clause C_i is a disjunction of three variables or their negations taken from $\{x_1, \ldots, x_n\}$.

Given φ, we define a graph G, a set Σ of GEDs, and a block Γ of ground truth such that (Σ, Γ) is consistent. We show that $G_{Eq} \models \Sigma$ if and only if $\mu(\bar{x})$ sets an odd number of variables true.

The reduction takes several steps, through the problems below.

(1) The first problem, denoted by $\langle \varphi, k \rangle$, is to decide, given φ and a number k, whether all satisfying assignments of φ have no more than k variables set true? We can see that $\langle \varphi, k \rangle$ is in coNP [Spa05].

Since φ has n variables, we have problems: $\langle \varphi, 1 \rangle, \ldots, \langle \varphi, n \rangle$.

(2) We build the connection between an instance $\langle \varphi, k \rangle$ and a 3-colorability instance (see the proof of Theorem 5.4 for the problem). Since 3-colorability problem is NP-complete [GJ79], given $\langle \varphi, k \rangle$, we can construct a graph G_k such that G_k has a proper

3-coloring iff $\langle \varphi, k \rangle$ returns false. Suppose that $G_1, \ldots, G_n$ are the constructed graphs. Then we can verify the following: $\mu(\bar{x})$ sets an odd number of variables true iff there exists an odd number k such that for all $k_1 \leq k$, G_{k_1} has a proper 3-coloring, and for all $k_2 > k$, G_{k_2} has no proper 3-coloring. Moreover, by the connection between $\langle \varphi, i \rangle$ and $\langle \varphi, i+1 \rangle$, if G_{i+1} is 3-colorable, then G_i is 3-colorable.

Having $G_1, \ldots, G_n$, we can construct G, Σ, and Γ as follows. (a) The graph G is to encode all proper 3-coloring, and it contains n attributes to indicate whether $G_1, \ldots, G_n$ have proper 3-coloring. (b) The set Σ consists of two groups of GEDs: the first one is to check whether $G_1, \ldots, G_n$ are 3-colorable; and the second one is to check whether there exists an odd number k such that for all numbers $k_1 \leq k$, G_{k_1} has a proper 3-coloring, and for all $k_2 > k$, G_{k_2} does not have a proper 3-coloring. (c) The block Γ is empty.

More specifically, we construct G, Σ and Γ as follows.

(1) As shown in Fig. 5.4(2), we define $G = (V, E, L, F_A)$ to encode 3-coloring, where (a) $V = \{v_0, v_1, v_2, v_3\}$, consisting of three nodes to represent the three colors, and another node to check whether $G_{\mathsf{Eq}} \models \Sigma$; (b) $E = \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_3, v_1), (v_2, v_3), (v_3, v_2)\}$; these edges make v_1, v_2 and v_3 form a clique; (c) $L(v_1) = r$, $L(v_2) = g$, $L(v_3) = b$ for three colors, and $L(v_0) = C$; and (d) $F_A(v_0).A = 1$, $F_A(v_0).A_1 = 0$, $\ldots$, and $F_A(v_0).A_n = 0$.

Intuitively, we will use attribute $F_A(v_0).A$ to separate the Chaseprocess and the checking of $G_{\mathsf{Eq}} \models \Sigma$. The other attributes $F_A(v_0).A_i$ $(i \in [1, n])$ are to indicate whether G_i is 3-colorable.

(2) The set Σ consists of two groups of GEDs as remarked above.

The first group consists of n GEDs, one for each of $G_1, \ldots, G_n$. For each $i \in [1, n]$, GED $\varphi_i = Q_i[x_0, x_1, \ldots, x_{m_i}] \big(\emptyset \to (x_0.A_i = 1) \big)$, where Q_i is given in Fig. 5.4 (2). More specifically, pattern $Q_i[\bar{x}] = (V_i, E_i, L_i)$, where (a) $V_i = V_i \cup \{v_0\}$, i.e., nodes in G_i pluses one node v_0; (b) $E_i = \{(u, v), (v, u) \mid (u, v) \in E\}$, i.e., each undirected edge (u, v) in G is encoded with two directed edges (u, v) and (v, u); and (c) each vertex $v_j \in V_i$ is labeled wildcard, i.e., $L_i(v_j) = $ '_'. This GED ensures that if G_i is 3-colorable, then $v_0.A_i$ must be 1.

The second group consists of $\lceil \frac{n}{2} \rceil$ GEDs, which are used to check the existence of the required odd number. To this end, we only need to ensure that for any even number i, if $v_0.A_i = 1$, then $v_0.A_{i+1}$ must also be 1, That is, if G_i is 3-colorable, then G_{i+1} is also 3-colorable. In light of this, given any i $(i \in [1, \lceil \frac{n}{2} \rceil])$, we define the following

GED $\varphi'_i = Q'_i[x_0]\big((x_0.A = 1) \wedge (x_0.A_{2i} = 1) \to (x_0.A_{2i+1} = 1)\big)$. The pattern Q'_i consists of only one node labeled C.

By the construction of $G_1, \ldots, G_n$, we know that for any number $i \in [1, n-1]$, if $v_0.A_i = 0$, then $v_0.A_{i+1}$ must also be 0; and for any $i \in [2, n]$, if $v_0.A_i = 1$, then $v_0.A_{i-1}$ is also 1. Indeed, for any number $i \in [1, n-1]$, if G_{i+1} is 3-colorable, then G_i is also 3-colorable. That is, all possible values for $v_0.A_1, \ldots, v_0.A_n$ are: $(0, 0, 0, \ldots, 0)$, $(1, 0, 0, \ldots, 0)$, $\ldots$, $(1, 1, \ldots, 1)$.

(3) The block Γ is set to be empty.

We first verify that $\mathsf{Chase}(G, \Sigma, \Gamma)$ is consistent, *i.e.*, (Σ, Γ) is consistent. Since Γ does not contain attribute $v_0.A$, and GEDs in Σ do not generate such attribute, only GEDs in the first group can be used during the computation of $\mathsf{Chase}(G, \Sigma, \Gamma)$. Because such GEDs only set values 1, $\mathsf{Chase}(G, \Sigma, \Gamma)$ is consistent.

We next show that $G_{\mathsf{Eq}} \models \Sigma$ if and only if $\mu(\bar{x})$ sets an odd number of variables true, *i.e.*, the construction above is a reduction.

($\Rightarrow$) Suppose that $G_{\mathsf{Eq}} \models \Sigma$. Assume by contradiction that $\mu(\bar{x})$ sets an even number k_e of variables true. We show that $G_{\mathsf{Eq}} \not\models \varphi'_{\frac{k_e}{2}}$, which contradicts to $G_{\mathsf{Eq}} \models \Sigma$. Since $v_0.A = 1$, we only need to show that $v_0.A_{k_e} = 1$, and $v_0.A_{k_e+1} = 0$ or $v_0.A_{k_e+1}$ does not exist. (i) We first show that $v_0.A_{k_e} = 1$. Because $\mu(\bar{x})$ sets k_e variables true, by the construction of $G_1, \ldots, G_n$, we know that G_{k_e} is 3-colorable. As in the lower bound proof of Theorem 5.4, one can show that $1 \in [v_0.A_{k_e}]_{\mathsf{Eq}}$. By the definition of the repair G_{Eq}, $v_0.A_{k_e} = 1$. (ii) When $k_e = n$, obviously $v_0.A_{k_e+1}$ does not exist. When $k_e \neq n$, we show that $v_0.A_{k_e+1} = 0$. Since $\mu(\bar{x})$ sets k_e variables true, $\langle \varphi, k_e + 1 \rangle$ return true. By the construction of G_{k_e+1}, we have that G_{k_e+1} is not 3-colorable. Again as in the lower bound proof of Theorem 5.4, we can show that $1 \in [v_0.A_{k_e+1}]_{\mathsf{Eq}}$ cannot be deduced from $\mathsf{Chase}(G, \Sigma, \Gamma)$. By the definition of G and G_{Eq}, we know that $v_0.A_{k_e+1} = 0$, since we initialize $v_0.A_{k_e+1} = 0$ in G. Therefore, $v_0.A = 1$, $v_0.A_{k_e} = 1$ and $v_0.A_{k_e+1} = 0$ in G_{Eq}, which contradicts to $G_{\mathsf{Eq}} \models \varphi'_{\frac{k_e}{2}}$.

($\Leftarrow$) Suppose that $\mu(\bar{x})$ sets an odd number k_o of variables true. Assume by contradiction that $G_{\mathsf{Eq}} \not\models \Sigma$. We show that $\mu(\bar{x})$ sets an even number of variables true, which contradicts to the hypothesis. It suffices to show that $G_{\mathsf{Eq}} \not\models \varphi'$ for some φ' in the second group of GEDs in Σ. For if it holds, by the construction of Σ, then there exist an even number k' such that $v_0.A_1, \ldots, v_0.A_n$ are in the form of $(1, 1, 1, \ldots, 1, 0, \ldots, 0)$, where only the first k' elements are 1. By the construction of Σ, we also know that

$G_1, \ldots, G_{k'}$ are 3-colorable, while $G_{k'+1}, \ldots, G_n$ are not 3-colorable. By the construction of $G_1, \ldots, G_n$, there exist k' variables set true by $\mu(\bar{x})$. That is, $\mu(\bar{x})$ sets an even number of variables true, a contradiction.

We now show that there exists φ' in the second GED group of Σ such that $G_{\mathsf{Eq}} \not\models \varphi'$. Since $G_{\mathsf{Eq}} \not\models \Sigma$, it suffices to show that $G_{\mathsf{Eq}} \models \varphi$ for all φ in the first GED group of Σ. We prove this by contradiction. Suppose that there exists a GED $\varphi_i = Q_i[x_0, x_1, \ldots, x_{m_i}] \big(\emptyset \to (x_0.A_i = 1) \big)$ in the first group such that $G_{\mathsf{Eq}} \not\models \varphi_i$. That is, there exists a match h of Q_i in G_{Eq} such that $h(x_0).A_i = 0$ in G_{Eq}. But because there exists a match of Q_i in G, $1 \in [x_0.A_i]_{\mathsf{Eq}}$ is in $\mathsf{Chase}(G, \Sigma, \Gamma)$. Since $\mathsf{Chase}(G, \Sigma, \Gamma)$ is consistent, by the definition of G_{Eq} and φ_i (note that X in its attribute constraint is $\emptyset$), $G_{\mathsf{Eq}} \models h(x_0).A_i = 1$, a contradiction. Hence, $G_{\mathsf{Eq}} \models \varphi$.   □

5.4   Deducing Certain Fixes

We next develop an algorithm, denoted by Clean, for cleaning graphs with certainty. Given a graph G, a set Σ of GEDs and a block Γ of ground truth such that (Σ, Γ) is consistent, Clean computes certain fixes Eq and the repair G_{Eq} of G in $\mathsf{Chase}(G, \Sigma, \Gamma)$.

While the chase (Section 5.2) provides a conceptual method, it is too costly to be practical. Each chase step starts from scratch: it (a) nondeterministically picks a GED $\varphi = Q(\bar{x})(X \to Y)$ from Σ, (b) finds a mapping h from $Q[\bar{x}]$ to G, and (c) checks whether X is entailed by $(\mathsf{Eq}, \mathsf{NEq})$ at $h(\bar{x})$. Among these, step (b) is costly given the intractability of graph homomorphism (cf. [GJ79]). After computing h, the chase may find that X is not entailed by $(\mathsf{Eq}, \mathsf{NEq})$ and φ cannot be enforced; it has to start steps (b) and (c) again.

To make the method practical, algorithm Clean proposes two strategies. (a) It computes an order on the GEDs of Σ to determine which GEDs can be used to expand the fixes in the next chase step. The order is computed by using a *precedence graph* that captures the impacts of GEDs enforced in the prior steps. (b) It *incrementally* expands Eq and NEq, *i.e.,* a chase step with a GED is taken *only if* it is "triggered" by newly generated fixes in Eq or NEq. These reduce redundant computation of costly graph homomorphism.

We start with precedence graphs, followed by algorithm Clean.

Precedence graphs. Under Γ, we define *the precedence graph G_p of Σ* as a directed graph (V_p, E_p), where (1) $V_p = \Sigma$, *i.e.,* the vertices are GEDs in Σ; (2) edge $(\varphi_1, \varphi_2) \in E_p$

indicates that applying φ_2 may make φ_1 applicable at a later chase step; conversely, if (φ_1, φ_2) is not an edge in G_p, then the application of φ_2 has no impact on whether φ_1 can be enforced. That is, the precedence graph G_p depicts the paths of *change propagation* when computing Eq and NEq.

We want to characterize edges with a simple condition that can be checked efficiently. To formalize E_p, we use a few notations.

(1) For an equality literal l in a GED, we define its *template $t_l(l)$* by substituting label $L_Q(x)$ for each variable x in l, e.g., $L_Q(x).A = c$ for $x.A = c$, $L_Q(x).A = L_Q(x).B$ for $x.A = y.B$, and $L_Q(x).\mathrm{id} = L_Q(y).\mathrm{id}$ for $x.\mathrm{id} = y.\mathrm{id}$; similarly for inequality literals.

We also extend t_l to ground truth (*i.e.,* fixes (u, v)) in Γ, upgrading nodes to their labels. We write $t_l(c) = c$ and $t_l(x.A) = L_Q(x).A$ for constants c and attributes $x.A$, referred to as a *term*.

We will use templates to catch precedence on the attribute dependencies of GEDs, by "generalizing" nodes to their labels.

(2) Given Σ and Γ, we denote by (a) T_Γ the set of templates of fixes in Γ; (b) T_Σ the set of $t_l(l)$ for all literals l that appear in Y of some GED $Q[\bar{x}](X \to Y) \in \Sigma$; and (c) $T_{(\Sigma,\Gamma)}$ the set of terms $t_l(x.A)$ and $t_l(c)$ for all attributes $x.A$ and constants c in Σ and Γ.

(3) Given a term t, we define the *closure* of t under Σ and Γ as the set $(t, \Sigma, \Gamma)^* = \{(\mathrm{op}, t') \mid T_\Sigma \cup T_\Gamma \models t \text{ op } t', \mathrm{op} \text{ is } = \text{ or } \neq, t' \in T_{(\Sigma,\Gamma)}\}$. That is, the closure includes *all possible* candidate templates $t = t'$ and $t \neq t'$ that can be deduced from literal templates of Γ and consequences Y of some attribute constraint $X \to Y$ in Σ.

Precedence graphs. We are now ready to formalize the edge relation E_p in the precedence graph G_p. For GEDs $\varphi_i = Q_i[\bar{x}](X_i \to Y_i)$ for $i \in [1, 2]$, (φ_1, φ_2) is an edge in E_p if and only if either

(a) there exist a template $t_2 \text{ op } t_2'$ of a literal in Y_2 and a template $t_1 \text{ op}' t_1'$ of a literal in X_1 such that (i) $(\mathrm{op}', t_1') \in (t_1, \Sigma, \Gamma)^*$, and (ii) either $(=, t_1') \in (t_2, \Sigma, \Gamma)^* \cup (t_2', \Sigma, \Gamma)^*$ or $(=, t_1) \in (t_2, \Sigma, \Gamma)^* \cup (t_2', \Sigma, \Gamma)^*$; or

(b) there exist an id literal $x.\mathrm{id} = y.\mathrm{id}$ in Y_2 and a node u in $Q_1[\bar{x}]$ such that $L_{Q_2}(x) \asymp L_{Q_1}(u)$.

Condition (a) states that after literals from Y_2 represented by template $t_2 \text{ op } t_2'$ are validated in a chase step, literals of $t_1 \text{ op}' t_1'$ from X_1 may be entailed by Eq and NEq at a later step. Closures $(t_2, \Sigma, \Gamma)^*$ and $(t_2', \Sigma, \Gamma)^*$ capture the equivalence relation of Eq and the impact of NEq. Condition (b) is to find new matches of $Q_1[\bar{x}]$ after its node u
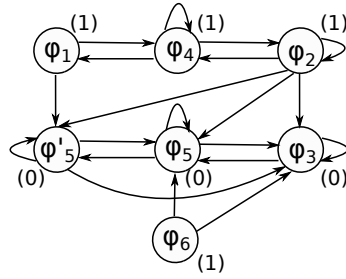
Figure 5.5: A precedence graph

is mapped to a node that is merged by enforcing φ_2.

Example 5.7: Recall Σ from Example 5.3. Under Γ of Fig. 5.1, the precedence graph G_p of Σ is shown in Fig. 5.5. By condition (a), edge (φ_4, φ_2) exists due to $(=, \text{league.country})$, where league.country, the term of x.country and y.country in X_4 of φ_4 and y.country in φ_2, is in closure $(\text{league.country}, \Sigma, \Gamma)^* = \{(=, \text{league.country}), (=, \text{stadium.country})\}$. That is, after φ_2 is applied, the change of y.country may make φ_4 applicable at a later chase step.

Edge (φ_2, φ_4) is in G_p since there exists a node y labeled league in $Q_2[\bar{x}]$, and φ_4 identifies league entities (condition (b) above). $\square$

Algorithm. We now present algorithm Clean in Fig. 5.6. Given graph G, GEDs Σ and ground truth Γ, it first builds the precedence graph G_p of Σ under Γ (line 1), and computes the *topological ranks* of the strongly connected components (scc's) in G_p (line 2), by procedures buildPrecedence and rankSCC, respectively (not shown). The topological rank of an scc is such defined that (a) scc.rank $= 0$ if it is a leaf node in the contracted graph G'_p that collapses each scc of G_p into a single node, and (b) scc.rank $= \max\{\text{scc}'.\text{rank} + 1 \mid (\text{scc}, \text{scc}') \in E'_p\}$, where E'_p is the edge set of G'_p. Each GED $\varphi \in \Sigma$ is in an scc. The *rank* of φ is defined as the topological rank of the scc in which φ belongs. A set $\varnothing$ collects the GED ranks (line 2).

Then algorithm Clean applies GEDs of Σ iteratively and incrementally, following the topological ranks of the GEDs, starting from the lowest rank (lines 3-14). For each i, it collects a set $\mathcal{R}$ of GEDs of rank i (line 4). For each GED $\varphi = Q[\bar{x}](X \rightarrow Y)$ in $\mathcal{R}$, it identifies a set S_φ of *partial* matches of Q by procedure filterPAns (line 7). It completes the matches and finds fixes via procedure Match_n (line 8). The partial matches are selected if X is entailed by Eq and NEq, in response to a set δ_{Eq} of fixes that are *newly added* (line 11).

Algorithm: Clean

Input: A graph G, a set Σ of GEDs, and a block Γ of ground truth.

Output: The certain fixes Eq and the repair G_{Eq} of G by (Σ, Γ).

1. $G_p := \mathsf{buildPrecedence}(\Sigma, \Gamma)$; Eq $:= \Gamma$; NEq $:= \emptyset$; $G_{\mathsf{Eq}} := G$;

2. $\emptyset := \mathsf{rankSCC}(G_p)$; $i := 0$;

3. **while** $i \leq \emptyset.\mathsf{maxrank}$ **do**

4.   set $\mathcal{R} := \{\varphi \mid \varphi \in \Sigma, \emptyset[\varphi].\mathsf{rank} = i\}$; $\delta_{\mathsf{Eq}} := (\mathsf{Eq}, \mathsf{NEq})$; $\delta_{\mathsf{cur}} := \emptyset$;

5.   **repeat**

6.   **for each** GED $\varphi = Q[\bar{x}](X \to Y)$ in $\mathcal{R}$ **do**

7. $S_\varphi := \mathsf{filterPAns}(X, \mathsf{Eq}, \mathsf{NEq}, \delta_{\mathsf{Eq}})$;

8. $\delta_{\mathsf{cur}} := \delta_{\mathsf{cur}} \cup \mathsf{Match}_{\mathsf{n}}(Q[\bar{x}], G_{\mathsf{Eq}}, S_\varphi)$;

9. $(\mathsf{Eq}_{\mathsf{pre}}, \mathsf{NEq}_{\mathsf{pre}}) := (\mathsf{Eq}, \mathsf{NEq})$;   expand Eq and NEq with δ_{cur};

10. $(\mathsf{Eq}, \mathsf{NEq}) := \mathsf{compEqv}(\delta_{\mathsf{cur}}, \mathsf{Eq}, \mathsf{NEq})$;

11. $\delta_{\mathsf{Eq}} := (\mathsf{Eq} \cup \mathsf{NEq}) \setminus (\mathsf{Eq}_{\mathsf{pre}} \cup \mathsf{NEq}_{\mathsf{pre}})$;

12. $G_{\mathsf{Eq}} := \mathsf{Mutate}(G_{\mathsf{Eq}}, \delta_{\mathsf{Eq}})$;

13.   **until** δ_{Eq} is an empty set;

14. $i := i + 1$;

15. **return** $(\mathsf{Eq}, G_{\mathsf{Eq}})$;

Figure 5.6: Algorithm Clean

Clean expands Eq and NEq with fixes deduced in the current iteration (line 9). It computes the equivalence relation of Eq and makes NEq symmetric and "transitive" via Eq by procedure compEqv (line 10), and repairs graph G_{Eq} by applying the newly included fixes in δ_{Eq} (line 11) by procedure Mutate (not shown; line 12). The process continues until no more fixes can be added to Eq or NEq (line 13). It returns Eq and G_{Eq} after all GEDs in Σ are processed (line 15).

We next present procedures filterPAns and $\mathsf{Match}_{\mathsf{n}}$.

Procedure filterPAns. The procedure takes as input X from GED $Q[\bar{x}](X \to Y)$, Eq, NEq, and the set δ_{Eq} of fixes added in the *last* iteration. It finds a set S_φ of homomorphic mappings from the pattern nodes that appear in X to the nodes in Eq $\cup$ NEq, refereed to as *partial matches*. It ensures that every mapping h in S_φ satisfies X and contains at least one node z from δ_{Eq}, *i.e.,* $h(\bar{x}_s) \models X$ and $h(y) = z$ for some node z in δ_{Eq}, where $\bar{x}_s \subseteq \bar{x}$ is a list of variables denoting pattern nodes in X, and $y \in \bar{x}_s$.

To speed up this process, Clean maintains an inverted index from templates t_l to fixes that have the same t_l. Hence filterPAns *joins* the newly enforced fixes with the previous ones to *incrementally* find fresh partial matches where X can be validated.

Procedure $\mathsf{Match_n}$. $\mathsf{Match_n}$ completes partial matches S_φ of $Q[\bar{x}]$ in G_{Eq} that were computed by filterPAns. It follows the generic subgraph matching process [LHKL12], and verifies graph homomorphism by checking edge connections. It also deduces fixes to be included in Eq and NEq. Note that $\mathsf{Match_n}$ only incrementally enumerates those matches that can be expanded from freshly computed partial matches and must involve lately merged nodes when $S_\varphi = \emptyset$.

Example 5.8: Continuing with Example 5.7, algorithm Clean first computes the ranks of GEDs in Σ based on precedence graph G_p (the ranks are annotated in brackets in Fig 5.5). It then chases graph G by (Σ, Γ), starting with GEDs of rank 0, *i.e.*, φ_3, φ_5, and φ_5'.

Consider $\varphi_3 = Q_3[\bar{x}](X_3 \rightarrow Y_3)$ as an example. Templates for X_3 are $\mathsf{stadium.capacity} = \mathsf{stadium.capacity}$ and $\mathsf{architect.id} = \mathsf{architect.id}$. Procedure filterPAns finds fixes that have the templates in Eq_0 and δ_{Eq}, including $s_1.\mathsf{capacity} = s_2.\mathsf{capacity}$ and $a.\mathsf{id} = a.\mathsf{id}$, where initial $\delta_{\mathsf{Eq}} = (\mathsf{Eq}_0, \emptyset)$. Two partial matches h_1 and h_2 are found: $h_1(\bar{x}_s)$: $x \mapsto s_1$, $x' \mapsto s_2$, $y \mapsto a$, $y' \mapsto a$, and $h_2(\bar{x}_s)$: $x \mapsto s_2$, $x' \mapsto s_1$, $y \mapsto a$, $y' \mapsto a$. Since these two make (complete) matches of $Q[\bar{x}]$ in G, *i.e.*, $\bar{x}_s = \bar{x}$, procedure $\mathsf{Match_n}$ simply checks the homomorphism condition, and deduces a new fix $s_1.\mathsf{id} = s_2.\mathsf{id}$ to be enforced at h_1 and h_2. Clean expands Eq with $s_1.\mathsf{id} = s_2.\mathsf{id}$, and updates G_{Eq} by identifying nodes s_1 and s_2. No partial matches are found for φ_5 and φ_5' in the first iteration.

Since a fix $s_1.\mathsf{id} = s_2.\mathsf{id}$ is added to Eq, Clean continues to process GEDs of rank 0. Nodes f_1 and f_2 are merged by φ_5' in the second iteration, in which $s_1.\mathsf{id} = s_2.\mathsf{id}$ is used to build partial matches of $Q_5[\bar{x}]$, and s_3 and s_4 are merged in the third iteration by φ_5.

After these, Clean processes the GEDs of rank 1, since no new fixes can be deduced by applying φ_3, φ_5 or φ_5'. For instance, φ_2 fixes attribute $c_1.\mathsf{country}$ as in Example 5.5, φ_1 is used to validate $c_2.\mathsf{country}$, and c_1 and c_2 are merged by φ_4 in the next iteration by using the newly validated $c_1.\mathsf{country} = c_2.\mathsf{country}$. Once no new fixes can be added, Clean returns Eq_7 and G_{Eq_7} of Example 5.6. $\square$

Analyses. Clean checks all the applicable chase steps that expand Eq or NEq at some

point, guided by the precedence graph. Its correctness follows from Proposition 5.6.

Proposition 5.6: *Given G, Σ and Γ, a fix (u,v) is deduced by chasing G by (Σ,Γ) iff (u,v) is added to* Eq *or* NEq *by algorithm* Clean. $\qquad\square$

Proof: We denote a fix (u,v) by l, and prove the two directions as follows.

($\Rightarrow$) Suppose that fix l is added to Eq or NEq by $\mathsf{Chase}(G,\Sigma,\Gamma)$. Let $(\mathsf{Eq}_0,\mathsf{NEq}_0) \Rightarrow_{(\varphi_1,h_1)} \cdots \Rightarrow_{(\varphi_k,h_k)} (\mathsf{Eq}_k,\mathsf{NEq}_k)$ be a terminal chasing sequence ρ. We show that l is included in Eq or NEq by Clean, by induction on the number of chase steps of ρ.

(1) When l is included in $(\mathsf{Eq}_0,\mathsf{NEq}_0)$, then l must be in Γ. Obviously, l is included in Eq or NEq by Clean (line 2).

(2) Suppose that fix l is added to $(\mathsf{Eq}_{i+1},\mathsf{NEq}_{i+1})$ by applying a GED $\varphi = Q[\bar{x}](X \to Y)$ at a match h, where the rank of φ is t, $h(X)$ uses fixes $l'_1,\ldots,l'_n$ in $(\mathsf{Eq}_i,\mathsf{NEq}_i)$, and $l'_1,\ldots,l'_n$ either exist in Γ, or are generated by $\varphi'_1,\ldots,\varphi'_m$. Then by the induction hypothesis, l'_j's are included in Eq or NEq by Clean. Since $h(X)$ uses $l'_1,\ldots,l'_n$, by the precedence graphs, the ranks of $\varphi'_1,\ldots,\varphi'_m$ are no larger than t. Hence l is also included in Eq or NEq by Clean (lines 5-14).

($\Leftarrow$) We show that all fixes included in Eq or NEq by Clean can be deduced by the chase when computing $\mathsf{Chase}(G,\Sigma,\Gamma)$, by induction on the computation steps of Clean. In the first step (line 1), Clean starts with $\mathsf{Eq}_0 = \Gamma$ and $\mathsf{NEq}_0 = \emptyset$. Thus the statement holds.

Assume that Eq_i and NEq_i are extended from Eq_{i-1} and NEq_{i-1} by Clean (lines 9-10). By the induction hypothesis, all fixes in Eq_{i-1} and NEq_{i-1} are deduced by the chase. We show by contradiction that all fixes in Eq_i and NEq_i are also deduced by the chase. Suppose that there exists a fix l in Eq_i or NEq_i that is not deduced by the chase. Since l is in Eq_i or NEq_i but not in Eq_{i-1} and NEq_{i-1}, Clean (lines 7-8) applies a GED φ at a match h to add l. Hence one can verify that $(\mathsf{Eq},\mathsf{NEq}) \Rightarrow_{(\varphi,h)} (\mathsf{Eq}',\mathsf{NEq}')$ is a valid chase step, where $(\mathsf{Eq}',\mathsf{NEq}')$ extends $(\mathsf{Eq},\mathsf{NEq})$ by adding l. In other words, l can be deduced by the chase, a contradiction. Hence, all fixes added to Eq_i and NEq_i by Clean can be deduced by the chase. $\qquad\square$

For the worst-case complexity, algorithm Clean is in $O(|\Sigma||G|^{|\Sigma|})$ time. Indeed, (a) it takes $O((|\Gamma| + |\Sigma|)^3)$ time to build precedence graph G_p (line 1), and $O(|\Sigma|^2)$ time

to compute the topological ranks of scc's in G_p (line 2) [Tar72]. (b) Identifying and completing partial matches take $O(|\varphi||G|^{|\varphi|})$ time, for processing GED φ (lines 7-8). (c) While the same GED may be involved in multiple iterations, each partial match is processed only once since procedure filterPAns outputs partial matches incrementally. Hence, the total cost for implementing the chase is at most $O(|\Sigma||G|^{|\Sigma|})$ (lines 3-14). Putting these together, Clean takes $O((|\Gamma|+|\Sigma|)^3 + |\Sigma|^2 + |\Sigma||G|^{|\Sigma|}) = O(|\Sigma||G|^{|\Sigma|})$ time, since $|\Sigma| \ll |G|$ and $|\Gamma| \ll |G|$ in practice.

5.5   A Parallel Scalable Algorithm

Theorem 5.4 tells us that graph cleaning is intractable. To clean large-scale graphs, it is often necessary to use parallel algorithms. Below we first review a characterization of parallel algorithms (Section 5.5.1). We then parallelize algorithm Clean and show a performance guarantee of the parallel algorithm (Section 5.5.2).

5.5.1   Parallel Scalability

One might be tempted to think that a parallel algorithm would run faster given more processors. However, few algorithms in the literature guarantee this. Worse still, for some computation problems, no parallel algorithms would run much faster no matter how many processors are added. For instance, it is known that the computational and communication costs of distributed graph simulation [HHK95] are functions of the size $|G|$ of the data graph G, which do not necessarily get smaller when more processors are added [FWW14a]. This suggests that we characterize the effectiveness of parallel algorithms. To this end, we revise a notion of parallel scalability introduced by [KRS90] and widely used in practice.

An algorithm $\mathcal{A}_p$ for graph cleaning is said to be *parallel scalable relative to algorithm* Clean if its running time can be expressed as:

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

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

Intuitively, a parallel scalable $\mathcal{A}_p$ linearly reduces the cost of Clean when p increases, by taking sequential algorithm Clean as a yardstick. The main conclusion we can draw from the parallel scalability is that $\mathcal{A}_p$ is able to run faster when adding

more processors, and hence makes it feasible to scale with large graphs G. In contrast, [FWW14a] shows that there exists no parallel scalable algorithm for graph simulation, and the proof given there holds in general distributed frameworks.

Note that this is analogous to the notion of parallel scalability on incremental error detection algorithms defined in Section 4.5.1.

5.5.2 Parallelizing Algorithm Clean

We provide an algorithm for cleaning graphs with certainty, denoted as PClean, by parallelizing Clean. It cleans graphs that are fragmented and distributed across processors $W_0, \ldots, W_{p-1}$. We show that PClean is parallel scalable relative to Clean.

Algorithm PClean runs in supersteps, where GEDs of the same rank are processed *in parallel in the same superstep*. Similar to algorithm Clean, PClean expands Eq and NEq incrementally guided by the ranks of GEDs, while the procedures of partial match identification and completion, *i.e.,* filterPAns and Match$_n$ in Clean, are parallelized by using a workload partition strategy. Relations Eq and NEq are also distributed across p processors, such that each processor W_i maintains Eq$^{(i)}$ and NEq$^{(i)}$ and moreover, each fix in the local copies contains at least one node stored at W_i for $i \in [0, p-1]$.

We next show how to parallelize filterPAns for partial match identification and Match$_n$ for match completion, to evenly partition their costs across p processors, which dominate the cost of Clean.

Parallel partial match identification. This is to identify partial matches ($h(\bar{x}_s)$ in Section 5.4) that satisfy the entailment condition of the chase. It is performed in parallel by procedure PFilter of PClean. The new challenges are that a partial match here may be deduced from fixes that are filtered at different processors.

Procedure PFilter takes a set $\mathcal{R}$ of GEDs as part of its input. Consider a GED $Q[\bar{x}](X \to Y)$ in $\mathcal{R}$, and a literal l in X. At each processor W_i, PFilter inspects the fixes in the local Eq$^{(i)}$ and NEq$^{(i)}$ that share the same template $t_l(l)$, referred to as *the fixes by* $t_l(l)$. Since fixes are scattered all over the p processors, in contrast to the sequential filterPAns that combines these fixes directly to construct the partial matches $h(\bar{x}_s)$, processor W_i broadcasts its local fixes to other processors, so that partial matches can be constructed in parallel. Upon receiving the fixes, all processors sort the set of fixes by $t_l(l)$, denoted as Eq$[t_l]$, based on a predefined order.

PFilter then assembles these fixes to construct partial matches of $Q[\bar{x}]$, *in parallel.*

Let $t_1, \ldots, t_m$ be the templates of all literals in X. Since the cardinality $\|Eq[t_l]\|$ of each $Eq[t_l]$ is known, PFilter evenly partitions the assembling work. More specifically, each processor W_j is assigned an index IDX_i (range) for each sorted $Eq[t_i]$, such that the indices $\{IDX_i \mid i \in [1,m]\}$ of its groups of fixes satisfy

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

That is, the combinations of the fixes are evenly partitioned across p processors in a "round-robin" manner. Let Δ_{Eq} denote the set of fixes included in Eq or NEq in the last superstep. The matches with nodes involved in Δ_{Eq} are returned by each processor W_j.

Example 5.9: Consider (a) GED $\varphi_7 = Q_5[\bar{x}](X_7 \to y.\text{id} \neq y'.\text{id})$, where X_7 is $\{x.\text{country} = \text{"England"}, x'.\text{country} = \text{"Scotland"}\}$, and Q_5 is given in Fig. 5.2; it states that two football clubs cannot be identified if their home grounds are in England and Scotland, respectively, and (b) a graph G' that consists of stadium's m_0–m_9 and football club's b_0–b_{99}, in which m_i is owned by b_j for $i \in [0,9]$ and $j \in [10i, 10i+9]$; moreover, as ground truth, m_0–m_4 have value "England" for attribute country, and m_5–m_9 are in "Scotland".

Given these, PFilter first picks from ground truth $Eq[t_1] = \{m_i.\text{country} = \text{"England"} \mid i \in [0,4]\}$ and $Eq[t_2] = \{m_i.\text{country} = \text{"Scotland"} \mid i \in [5,9]\}$, by templates t_1 of stadium.country = "England" and t_2 of stadium.country = "Scotland". Then the PFilter evenly distributes the work of combining the fixes (ground truth) in $Eq[t_1]$ and $Eq[t_2]$, 25 in total, to deduce partial matches. Assume that there are 5 processors, and that $Eq[t_1]$ (resp. $Eq[t_2]$) is such sorted that $m_i.\text{country} = \text{"England"}$ (resp. $m_i.\text{country} = \text{"Scotland"}$) is assigned index i (resp. $i-5$). By the equation above, each processor W_k validates 5 partial matches that are composed of m_i and m_j such that $5i + j \bmod 5 = k$, for $k \in [0,4]$, *i.e.,* pattern nodes x and x' in $Q_5[\bar{x}]$ are mapped to such m_i and m_j, respectively. $\square$

Parallel match completion. Algorithm PClean invokes procedure PMatch to complete the partial matches and to find fixes to be added to Eq and NEq, in parallel. PMatch works as follows.

(1) For each partial match $h(\bar{x}_s) \in S_{\mathcal{R}}$ of pattern $Q[\bar{x}]$ assembled at processor W_i, PMatch identifies the *candidates* for pattern nodes u that remain to be matched, by breadth depth search. Here the set $C_h(u)$ of candidates for u is a set of nodes in G_{Eq}, where for each node $v \in C_h(u)$, (a) $L(v)$ matches $L_Q(u)$, and (b) there exists $v' \in C_h(u')$ or v' matches u' in $h(\bar{x}_s)$ such that edge (v',v) (resp. (v,v')) is in G_{Eq} for a pattern edge

(u',u) (resp. $(u.u')$) adjacent to u in $Q[\bar{x}]$, with matching labels. When reaching "borders", *e.g.,* nodes with crossing edges to other fragments, it notifies other processors the next pattern node to match, and the traversal continues there.

Then processor W_i broadcasts $h(\bar{x}_s)$ along with the candidates of u and the edges connecting them. Each processor sorts the candidates for each pattern node, like how PFilter treats 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 $Q[\bar{x}]$, processor W_j groups candidates from $C_h(u)$ for each $u \in \bar{x} \setminus \bar{x}_s$, along the same lines as combining fixes in PFilter. Moreover, 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 except that the cardinality of $C_h(u)$ is used here. Like Match$_n$, it incrementally expands matches with newly added fixes. It then checks edges between candidates for homomorphism, and returns the fixes to be enforced by the qualified matches.

Example 5.10: Continuing with Example 5.9, PMatch completes those partial matches obtained by mapping y and y' in G'. For each partial match $h(x,x') = (m_i,m_j)$, it first finds candidate sets $C_h[y] = \{f_{i'} \mid i' \in [10i, 10i+9]\}$ and $C_h[y'] = \{f_{j'} \mid j' \in [10j, 10j+9]\}$ from G' since such $f_{i'}$'s and $f_{j'}$'s connect to $h(x)$ and $h(x')$, respectively. Then the nodes in $C_h[y]$ and $C_h[y']$ are grouped together in parallel using the same partition strategy as in Example 5.9, in which processor W_k verifies 20 matches with $y \mapsto f_{i'}$ and $y' \mapsto f_{j'}$ such that $10i' + j' \mod 5 = k$ for $k \in [0,4]$. Thus all the processors find the same number of matches in PMatch, and each $f_{i'}.\text{id} \neq f_{j'}.\text{id}$ is included to NEq since the matches are graph homomorphisms. □

Algorithm. The main driver of PClean is shown in Fig. 5.7. It first computes the ranks of the GEDs as in Clean, by building the precedence graph of Σ at a designated processor (line 1). It then deduces fixes to be enforced in supersteps based on GED ranks (lines 3-12). In each superstep, PFilter identifies partial matches (line 6) and PMatch completes the partial matches (line 7), in parallel at all processors. Each new fix l returned by PMatch is added to the local copy of Eq or NEq at the corresponding processors W_i (line 8), where at least one node in l is stored. After updating Eq and NEq (line 9), each processor mutates its fragment of G, by identifying objects and repairing attributes based on the fixes newly deduced (line 10). Once no more fix can

---

Algorithm: PClean

Input: A fragmented graph G, a set Σ of GEDs, and a block Γ of ground truth.

Output: The certain fixes Eq and repair G_{Eq} of G by (Σ, Γ).

1.   compute the topological ranks of GEDs in Σ;

2. $\mathsf{Eq} := \Gamma$; $\mathsf{NEq} := \emptyset$; $G_{\mathsf{Eq}} := G$; $i := 0$;

3.   **while** there exist GEDs in Σ that have not been processed **do**

4.   collect in $\mathcal{R}$ those GEDs in Σ with rank i; $\Delta_{\mathsf{Eq}} := (\mathsf{Eq}, \mathsf{NEq})$;

5.   **repeat**   /*one superstep*/

6. $S_{\mathcal{R}} := \mathsf{PFilter}(\mathcal{R}, \mathsf{Eq}, \mathsf{NEq}, \Delta_{\mathsf{Eq}})$; /*parallel partial match identification*/

7. $\Delta_{\mathsf{Eq}} := \mathsf{PMatch}(\mathcal{R}, S_{\mathcal{R}}, G_{\mathsf{Eq}})$;   /*parallel match completion*/

8.   expand Eq and NEq with Δ_{Eq};

9.   compute the equivalence relation of Eq;   update NEq;

10.   update G_{Eq} with Δ_{Eq};

11.   **until** there is no more fix included in Eq or NEq;

12. $i := i + 1$;

13. **return** $(\mathsf{Eq}, G_{\mathsf{Eq}})$;

Figure 5.7: Algorithm PClean

be deduced (line 11), GEDs of the next rank are processed (line 12). When all GEDs of Σ are processed, each processor returns its local copy of Eq and G_{Eq} (line 13).

Parallel scalability. To see that PClean is parallel scalable relative to Clean, it suffices to show that PFilter and PMatch take $O(|\varphi||G|^{|\varphi|}/p)$ time for each GED $\varphi \in \Sigma$. For if it holds, by $|\Sigma| \ll |G|$ and $|\Gamma| \ll |G|$, PClean is in $O((|\Gamma| + |\Sigma|)^3 + |\Sigma|^2 + |\Sigma||G|^{|\Sigma|}/p) = O(|\Sigma||G|^{|\Sigma|}/p) = O(t(|G|, |\Sigma|, |\Gamma|)/p)$ time; note that the costs of expanding Eq and NEq and updating G_{Eq} are much less.

Observe that PFilter (a) filters fixes from Eq and NEq by templates in φ, in $O(|\varphi||G|^2)$ time; (b) broadcasts fixes and sorts the sets of fixes in $O(|\varphi||G|^2 \log |G|)$ time; (c) joins fixes with indices, in $O(|\varphi||G|^{|\varphi|}/p)$ time, since the workload is evenly partitioned. For the cost of PMatch, note that (d) it takes $O(|\varphi||G|)$ time to explore the candidates for each partial match; (e) there are at most $O(|G|^{|\varphi|})$ many partial matches, and they are evenly partitioned across processors as assured by PFilter; (f) each processor takes $O(|\varphi||G|^{|\varphi|}/p)$ time to validate its partition of candidates, along the same lines as (c); and (g) the communication cost is also in $O(|\varphi||G|^{|\varphi|}/p)$, which can be ver-

ified by induction on the lengths of partial matches. Thus the total cost for processing φ is in $O(|\varphi||G|^{|\varphi|}/p)$.

5.6 Experimental Evaluation

Using real-life and synthetic graphs, we conducted three sets of experiments to evaluate (1) the effectiveness of the method by combining data repairing and object identification; (2) the impact of GEDs and ground truth on the quality of repairs; and (3) the efficiency and scalability of the algorithms for deducing certain fixes.

Experimental setting. We used the following datasets.

Graphs. We used two real-life graphs: (a) DBpedia [DBp], 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 [SKW07] with 2 million entities of 13 types and 5.7 million edges of 36 types.

We also designed a generator to produce synthetic graphs G, controlled by the number of nodes $|V|$ (up to 50 million) and number of edges $|E|$ (up to 100 million), with labels and attributes drawn from an alphabet $\mathcal{L}$ of 100 symbols. Each node is assigned 8 attributes with values drawn from a domain $\mathcal{D}$ of 300 elements.

GEDs. We generated a set Σ of GEDs for each graph by extending an algorithm of [FLL$^+$17], which discovers GEDs with high support. The support of GED φ in a graph G' indicates how often φ can be applied to G'. We mined 200, 150 and 100 GEDs from DBpedia, *YAGO2* and Synthetic, respectively, in which (a) the number of nodes in patterns, denoted by k, is at most 6 (up to 36 edges), and (b) the number of literals in attribute constraints, denoted by r, is at most 8.

Dirty datasets. Since there is no "gold standard" to compare with, we introduced noise to the "clean" datasets to evaluate the effectiveness of graph cleaning algorithms. To do this, we first constructed "clean" datasets by (1) applying the GEDs of Σ discovered to the original graphs, and (2) manually resolving the violations of Σ detected.

We then introduced noise to the clean graphs by (a) updating attribute values, controlled by *inconsistency rate* err%, the ratio of the number of updated attributes to the total number of attributes; and (b) adding duplicate entities, controlled by *duplicate rate* dup%, *i.e.,* the percentage of duplicate entities in the entire graph.

Ground truth. The block Γ is sampled from the clean graphs, controlled by the size

$|\Gamma|$, and a *relevance* ratio rlv%. Here rlv% measures the percentage of pairs (u,v) of confirmed attribute values and entity matches in Γ that are relevant to Σ. A pair (u,v) is *relevant* to Σ if it can be used in validating X at some matches for a GED $Q[\bar{x}](X \rightarrow Y) \in \Sigma$. Intuitively, rlv% indicates to what extent ground truth Γ can be involved in fixing the noise with GEDs Σ.

Algorithms. We implemented the following, all in Java. (1) Sequential Clean (Section 5.4) and three variants: (a) Clean_D, which applies GFDs to correct attribute values only; (b) Clean_O, which only identifies entities by using keys; and (c) Clean_E, which uses GEDs defined with equality literals only. (2) A sequential algorithm ChaseG that implements the chase (Section 5.2) directly. (3) The entity matching algorithm EM_{MR} of [FFTD15] and an entity resolution algorithm GraphER by graph clustering [BG06]. (4) Parallel PClean (Section 5.5), and a variant PClean_{nw} that does not support workload partition.

The experiments were conducted on Amazon EC2 r3.2xlarge instances, each was powered by Intel Xeon E5-2670v2, with 61 GB memory and 122 GB SSD storage. We used up to 20 instances. All the experiments were run 5 times, and the average is reported here.

Experimental results. We next report our findings. The accuracy of the algorithms are evaluated by F-measure defined as $2 \cdot (\text{precs} \cdot \text{recall})/(\text{precs} + \text{recall})$. Here (a) precs is the ratio of *true* fixes, including attributes correctly updated and true duplicate entities identified, to all the fixes derived by an algorithm; and (b) recall is the ratio of the noise correctly fixed to all the noise in the graph, *i.e.,* inconsistent attribute values and duplicate entities. Note that precs is always 100% for Clean and PClean, while recall depends on how informative GEDs and ground truth are.

Exp-1: Interaction between data repairing and object identification. We compared the accuracy of Clean against the methods for only (1) repairing or (2) entity resolution (duplicate detection). We used real-life graphs and all GEDs discovered. We fixed rlv% = 90% and $|\Gamma| = 120K$ (resp. 90K) for DBpedia (resp. *YAGO2*).

(a) Deduplication helps repairing. Fixing duplicate rate dup% = 8%, we varied the inconsistency rate err% from 3% to 15%. The results on DBpedia and *YAGO2* are reported in Figures 5.8(a) and 5.8(b), respectively. We find the following. (a) Clean consistently outperforms Clean_D, and is at least 35.5% more accurate. This verifies that object identification indeed improves data repairing. (b) The accuracy (F-measure) of

(a) DBpedia: varying err%

(b) *YAGO2*: varying err%

(c) DBpedia: varying dup%
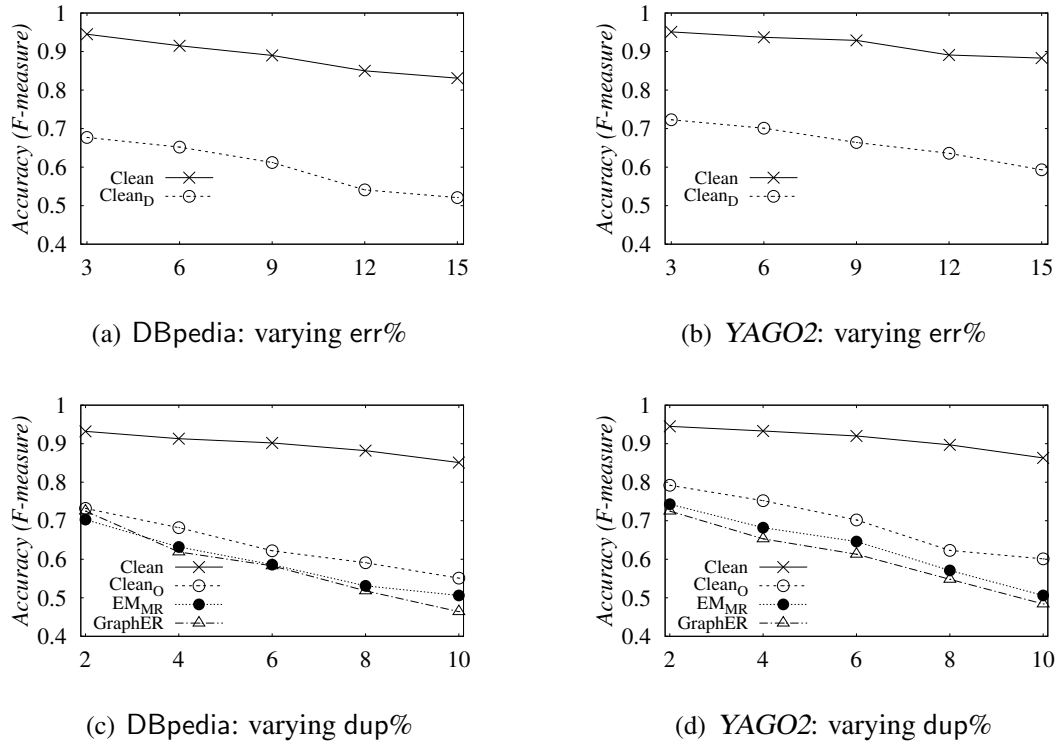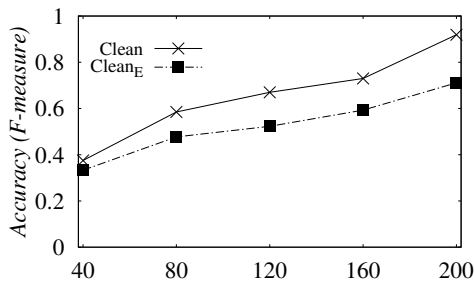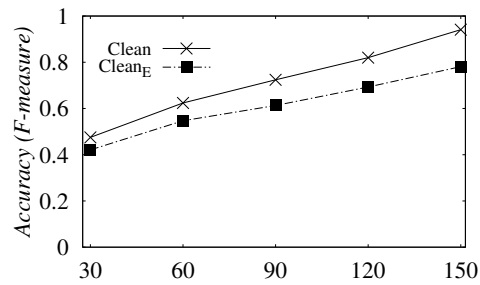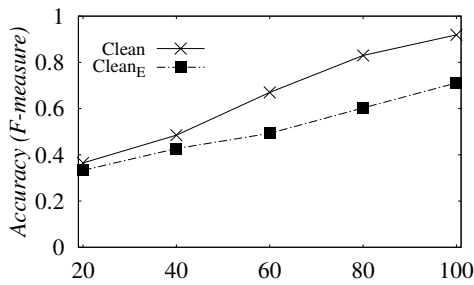
(d) *YAGO2*: varying dup%

Figure 5.8: Interaction between data repairing and object identification

both algorithms decreases when err% grows. However, Clean_D is less sensitive to the increase of err%.

(b) Repairing helps deduplication. Fixing inconsistency rate err% = 12%, we varied dup% from 2% to 10%. As shown in Figures 5.8(c) and 5.8(d) on DBpedia and *YAGO2*, respectively, (a) Clean achieves the best accuracy, and outperforms Clean_O, EM_{MR}, and GraphER by 36.5%, 49.1% and 54.1% on average, respectively. Indeed, the entity matching algorithms EM_{MR} and GraphER identify a number of false positives based on erroneous attribute values. (b) The larger dup% is, the less accurate is for each algorithm. However, the accuracy gaps between Clean and others get larger with the increase of dup%. These verify that data repairing helps object identification.

On average, the F-measure of Clean (and PClean) is 0.909 and 0.922 in the settings of (a) and (b) above, over real-life graphs.

Exp-2: Quality of repairs. We next evaluated the impact of GEDs Σ and ground truth Γ on the quality of the fixes derived by Clean, compared with the counterpart Clean_E that does not use GEDs with inequalities. We fixed err% = 15% and dup% = 10%.

(a) DBpedia: varying $\|\Sigma\|$

(b) *YAGO2*: varying $\|\Sigma\|$

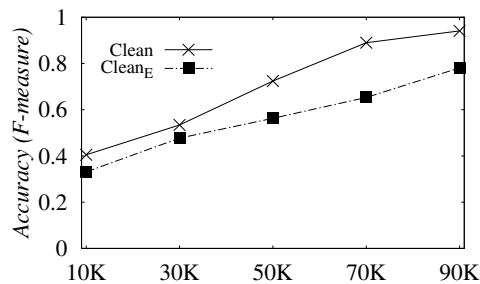(c) DBpedia: varying rlv%
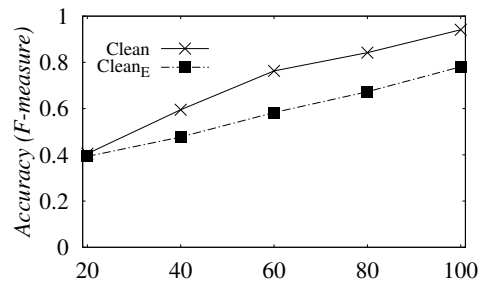
(d) *YAGO2*: varying rlv%

(e) *YAGO2*: varying $|\Gamma|$

Figure 5.9: Quality of repairs

Varying $\|\Sigma\|$. Using the same ground truth as in Exp-1, we varied the number $\|\Sigma\|$ of GEDs used from 40 to 200 (resp. 30 to 150) for DBpedia (resp. *YAGO2*). The results in Figures 5.9(a) and 5.9(b) show that (a) the more GEDs are available, the higher F-measure is achieved, as expected; and (b) Clean behaves better than Clean$_E$, and the improvement becomes more substantial with the increase of $\|\Sigma\|$. This verifies the effectiveness of supporting inequalities in GEDs.

Varying rlv%.   Fixing $\|\Sigma\| = 90$ and $|\Gamma| = 100K$ (resp. $80K$) for DBpedia (resp. *YAGO2*), we varied the relevance ratio rlv% from 20% to 100%. As shown

(a) DBpedia: varying *k*

(b) *YAGO2*: varying *r*

(c) DBpedia: varying $|\Gamma|$
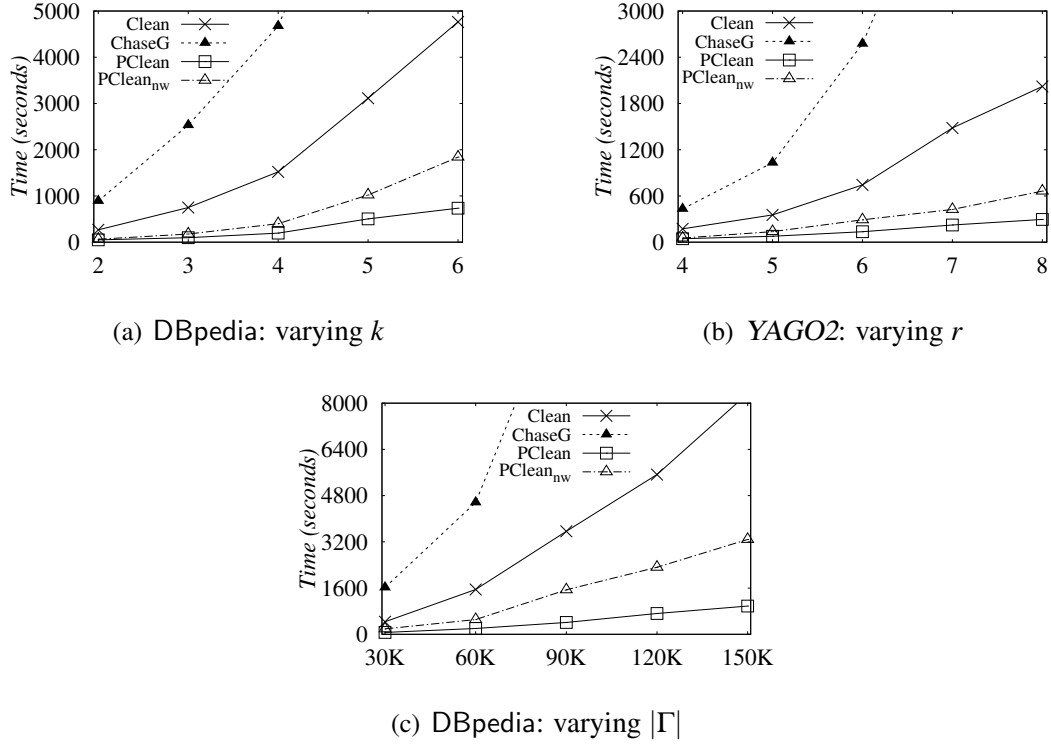
Figure 5.10: Efficiency of Clean, PClean

in Figures 5.9(c) and 5.9(d), (a) the larger rlv% is, the more certain fixes are found by Clean. This indicates that the number of fixes found highly depends on rlv%. (b) Clean outperforms Clean$_E$, consistent with Figures 5.9(a) and 5.9(b).

Varying $|\Gamma|$. Fixing $\|\Sigma\| = 100$ and rlv = 100%, We varied $|\Gamma|$ from 10*K* to 90*K* on *YAGO2*. As shown in Fig. 5.9(e), Clean does better when given a larger block of ground truth, *i.e.,* more confirmed attributes and entity matches help improve the quality of fixes, as expected. The results on DBpedia are consistent and hence not shown.

Exp-3 Efficiency and scalability. Finally, we evaluated the efficiency and scalability for cleaning graphs. In these experiments, we fixed err% = 15%, dup% = 10%, $\|\Sigma\| =$ 60, $k = 5$, $r = 6$, rlv% = 80%, $|\Gamma| = 50K$, and used $p = 16$ processors unless stated otherwise.

Impact of $|\Sigma|$. Varying k from 2 to 6 on DBpedia, and r from 4 to 8 on *YAGO2*, we evaluated the impact of $|\Sigma|$ on the graph cleaning algorithms. The results are reported in Figures 5.10(a) and 5.10(b), respectively. We find the following. (a) All the algorithms take longer to process GEDs with larger k or r, *i.e.,* having more pattern nodes and

(a) DBpedia: varying *p*

(b) *YAGO2*: varying *p*

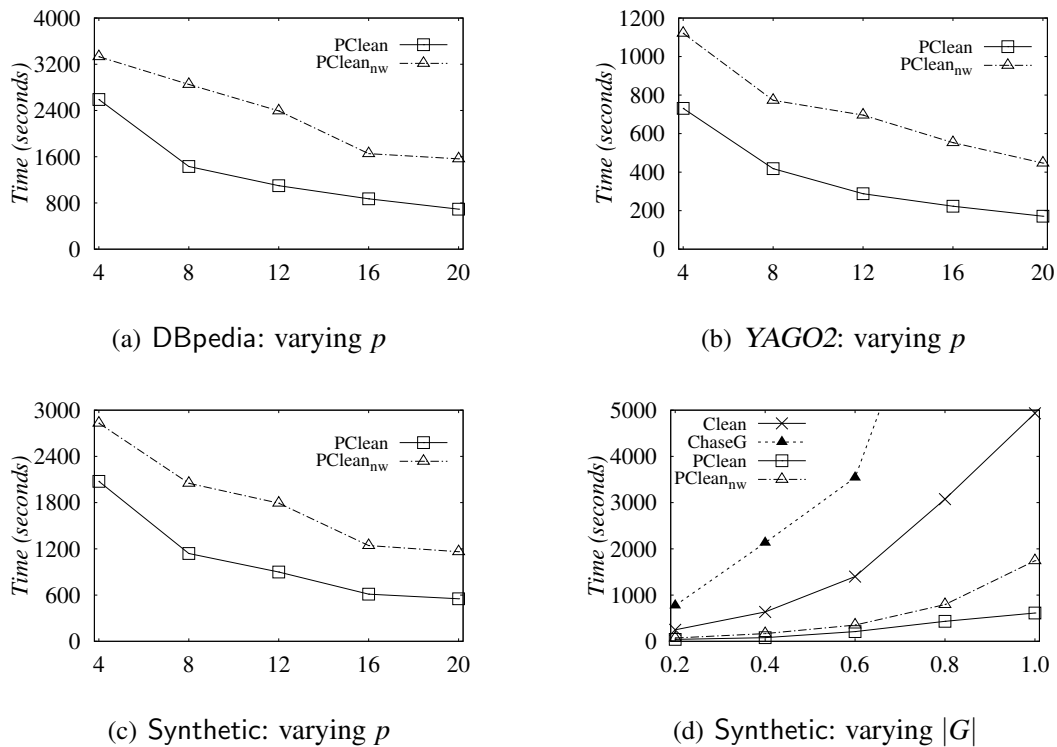(c) Synthetic: varying *p*

(d) Synthetic: varying $|G|$

Figure 5.11: Parallel scalability of PClean

literals, as expected. Moreover, they are more sensitive to the number *k* of pattern
nodes. (b) Clean substantially outperforms ChaseG, by 3.42 times on average. This
justifies the need for developing Clean, instead of using the chase. (c) PClean performs
the best in all cases, and is on average 2.15 times faster than PClean$_{nw}$.

Impact of $|\Gamma|$. We varied $|\Gamma|$ from 30*K* to 150*K* on DBpedia. As shown in Fig. 5.10(c),
(a) the larger $|\Gamma|$ is, the longer is taken by all the algorithms. (b) PClean outperforms
the others, consistent with Figures 5.10(a) and 5.10(b), and takes only 721*s* when $|\Gamma|$
is 120*K*. (c) PClean is less sensitive to $|\Gamma|$, due to its parallelization of the procedures
for partial match identification and completion in Clean.

Parallel scalability. We evaluated the parallel scalability of the parallel algorithms by
varying the number *p* of processors from 4 to 20. As shown in Figure 5.11(a) over
DBpedia (resp. 5.11(b), 5.11(c) over *YAGO2*, Synthetic), (a) PClean scales well with
p: the improvements is 3.74 (resp. 4.27, 3.76) times when *p* increases from 4 to 20; this
verifies the parallel scalability of PClean. (b) PClean outperforms PClean$_{nw}$ by 1.77
times on average. (c) PClean$_{nw}$ is less sensitive to the increase of *p*, due to unbalanced
workloads across processors. This verifies the effectiveness of our workload partition

strategy.

Impact of $|G|$. Using synthetic graphs G, we varied $|G|$ with scale factors from 0.2 to 1. Figure 5.11(d) shows the following. (a) PClean scales well with $|G|$ and is feasible on large graphs. It takes 430 seconds when G has 40 million nodes and 80 million edges, while Clean takes 3075 seconds, and ChaseG takes 9342 seconds. (b) PClean performs better than PClean$_{nw}$, consistent with Figures 5.11(a) to 5.11(c).

Summary. We find the following. (a) The certain fixes of Clean are of high quality: the F-measure is on average 0.916 with a couple hundred of GEDs and limited ground truth. (b) Cleaning graphs by unifying data repairing and object identification is effective: Clean outperforms repairing and entity resolution taken as independent tasks by 43.4% and 46.6% on average, respectively, in accuracy. (c) Cleaning graphs with certainty is feasible. PClean takes 430 seconds to find certain fixes for graphs with $120M$ nodes and edges using 16 processors, as opposed to $9742s$ by ChaseG. (d) PClean is parallel scalable. Its performance is improved by 3.92 times on average when p is from 4 to 20. (e) Our workload partition strategy is effective, improving the performance by 1.96 times on average.

Chapter 6

Conclusion and Future Work

We summarize the results of this thesis and propose future work in this chapter.

6.1 Summary

This thesis develops a package of techniques for the analysis of big graphs, which aim to improve both the efficiency and quality. Specifically, the main results include the following.

- We have established undoable and doable results for incremental graph computations. We have shown that the incremental problems for RPQ, SCC and KWS are unbounded under unit updates. However, we have proposed alternative characterizations for the effectiveness of incremental graph computations, and shown that RPQ, SCC, KWS and ISO are either localizable or bounded relative to their batch counterparts, by providing incremental algorithms with corresponding performance guarantees. Our experimental results have verified that the incremental algorithms substantially outperform their batch counterparts and scale well with large graphs, justifying the effectiveness of the new standards.

- We have proposed a class of keys for graphs. We have shown that entity matching with keys is NP-complete and hard to parallelize. Despite these, we have provided two parallel scalable algorithms, under MapReduce and vertex-centric asynchronous model, respectively. Our experimental results have verified that entity matching is feasible on big graphs in practice, in which the parallel running time of the parallel scalable algorithms are linearly reduced when the number of available processors increases.

- We have proposed a class of NGDs with arithmetic and comparison expressions

191

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

- We have proposed a method Analogist to clean graphs with certainty, by revising the chase. We have extended the GEDs of [FL17] to express rules for data repairing and object identification, positive and negative. We have settled fundamental problems for graph cleaning. We have developed (parallel scalable) algorithms underlying Analogist. Our experimental results have verified that the method is promising for fixing semantic inconsistencies in graphs.

6.2   Future Work

There is much to be done. We list some of the topics that are targeted for future work, which deserve a full treatment.

Incremental graph computations. As we have seen in Chapter 2, localizable and relatively bounded algorithms allow us to effectively conduct incremental computations on big graphs. Now the question is whether there are other alternative characterizations for the effectiveness of incremental graph algorithms? Therefore, one topic for future work is to classify graph queries commonly used in practice, characterize their incremental computations, and identify performance guarantees for their incremental algorithms when possible. Another topic is to identify practical conditions under which unbounded incremental problems become bounded or relatively bounded. We are also investigating (relative) bounded incremental algorithms under access constraints [CFHH15], which are a combination of cardinality constraints and indices. As indicated in [CFHH15], some instances of an unbounded incremental problem become "bounded" under access constraints. This issue also needs a full treatment.

Discovering keys and NGDs. To make practical use of keys and NGDs in identifying duplicate entities and detecting numeric inconsistencies, effective techniques have to been in place to find nontrivial and interesting keys and NGDs from real-life graphs. The problem is more challenging than discovery of relational FDs [HKPT99] and CFDs [HKPT99], since keys and NGDs both are combinations of topological con-

straints and attribute dependencies, not to mention the recursion and numeric expressions involved. Moreover, the validation analysis of the dependencies discovered is NP-complete and coNP-complete for keys (Chapter 3) and NGDs (Chapter 4), respectively, compared to low PTIME for the FD and CFD counterparts. It is also harder than conventional graph pattern mining as it has to deal with disconnected patterns.

Cleaning graphs. The work of Chapter 5 is still preliminary. One topic for future work is to clean graphs by combining rule-based and machine learning approaches, *i.e.,* combining the benefits of the both, similar to the method of [PSC$^+$15] for cleaning relational data. Another topic is to develop an algorithm that, given a graph G and a set Σ of GEDs, computes a minimum block Γ of "facts", *i.e.,* candidate ground truth, to be validated by experts or crowd-sourcing, such that all errors in G can be fixed by $\mathsf{Chase}(G, \Sigma, \Gamma)$. This is also challenging since the coverage problem for graph cleaning is already $P_{||}^{\mathsf{NP}}$-complete.

Parallel scalability. As remarked in Chapter 5, it is not always the case that there exist parallel scalable algorithms to answer graph queries in a big dataset G, such that given more processors, it is warranted to take less time. For instance, it has been shown that the computational and communication costs of distributed graph simulation are functions of the size $|G|$ of G, which do not necessarily get smaller when more processors are added [FWW14a]. In fact, a number of published distributed algorithms are not parallel scalable. Worse still, there is not even a generally accepted notion of parallel scalability yet. It is nontrivial to characterize this for shared-nothing architectures, when both computation complexity and communication costs are taken into account. We are formalizing the notion of parallel scalability for shared-nothing architectures, in response time and communication cost. That is, we want to decide whether it is feasible to answer our queries on big data if we are able to get more processors. We are also investigating systematic methods for developing parallel scalable algorithms with performance guarantees, and studying formal methods to prove the impossibility of parallel scalability for various query classes.

Bibliography

[ABC⁺11]   Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.

[ABH02]   Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, 2002.

[Aca05]   Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.

[ACP10]   Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. Constraints in RDF. In *SDKB*, 2010.

[AHV95]   Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AM17]   Yasser Altowim and Sharad Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, pages 909–920, 2017.

[AMR⁺98]   Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[Ant96]   Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *TCS*, 155(2):291–319, 1996.

[AP93]   Foto N. Afrati and Christos H. Papadimitriou. The parallel complexity of simple logic programs. *J. ACM*, 40(4):891–916, 1993.

[AR06]   Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[ARS09]   Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE*, 2009.

[Bai17]   Baidu. Personal communication, 2017.

[BBFL08]   Leopoldo E. Bertossi, Loreto Bravo, Enrico Franconi, and Andrei Lopatenko. Complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Inf. Syst.*, 33(4-5):407–434, 2008.

[BCN08]   Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, 2008.

[BDF⁺01]   Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for XML. In *WWW*, 2001.

[BFFR05]   Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[BG06]   Indrajit Bhattacharya and Lise Getoor. Entity resolution in graphs. *Mining graph data*, page 311, 2006.

[BG07]   Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.

[BGMG⁺07]   O. Benjelloun, H. Garcia-Molina, Heng Gong, H. Kawai, T.E. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS*, 2007.

[BGMS13]   Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. *TODS*, 38(3):14, 2013.

[BH91]   Samuel R. Buss and Louise Hay. On truth-table reducibility to SAT. *Inf. Comput.*, 91(1):86–102, 1991.

[Bha15]   Pramod Kumar Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Saarland University, 2015.

[BHBE10]   Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, pages 285–296, 2010.

[BHN$^+$02]   Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[BL99]   Robert D. Blumofe and Charles E. Leiserson. Scheduling multi-threaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[BS10]   Peter Buneman and Gianmaria Silvello. A Rule-Based Citation System for Structured and Evolving Datasets. *IEEE Data Eng. Bull.*, 33(3):33–41, 2010.

[BW13]   Paul Burkhardt and Chris Waring. An NSA big graph experiment. Technical Report NSA-RD-2013-056002v1, U.S. National Security Agency, 2013.

[BWR$^+$11]   Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *SOCC*, 2011.

[CFG$^+$07]   Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[CFHH15]   Yang Cao, Wenfei Fan, Jinpeng Huai, and Ruizhe Huang. Making pattern queries bounded in big graphs. In *ICDE*, 2015.

[CFP$^+$14]   Diego Calvanese, Wolfgang Fischl, Reinhard Pichler, Emanuel Sallinger, and Mantas Simkus. Capturing relational schemas and functional dependencies in RDFS. In *AAAI*, 2014.

[CFSV04]   Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[CGL$^+$96]   Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[CGST86]   William Cook, Albertus MH Gerards, Alexander Schrijver, and Eva Tardos. Sensitivity theorems in integer linear programming. *Mathematical Programming*, 34(3):251–264, 1986.

[CGWJ16]   Yang Chen, Sean Louis Goldberg, Daisy Zhe Wang, and Soumitra Sid-
            dharth Johri. Ontological pathfinding. In *SIGMOD*, 2016.

[Chr12]   Peter Christen. A survey of indexing techniques for scalable record
            linkage and deduplication. *TKDE*, 24, 2012.

[CIK16]   Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data dedu-
            plication. *PVLDB*, 9(11):864–875, 2016.

[CP12]   Alvaro Cortés-Calabuig and Jan Paredaens. Semantics of constraints
            in RDFS. In *AMW*, 2012.

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

[DEGI10]   Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano.
            Dynamic graph algorithms. In *Algorithms and theory of computation
            handbook*. Chapman & Hall/CRC, 2010.

[DGH+14]   Xin Luna Dong, Evgeniy Gabrilovich, Geremy Heitz, Wiko Horn,
            Kevin Murphy, Shaohua Sun, and Wei Zhang. From data fusion to
            knowledge fusion. *PVLDB*, 2014.

[DHM05]   Xin Dong, Alon Halevy, and Jayant Madhavan. Reference reconcilia-
            tion in complex information spaces. In *SIGMOD*, 2005.

[DMG+14]   Xin Luna Dong, K Murphy, E Gabrilovich, G Heitz, W Horn, N Lao,
            Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault:
            A web-scale approach to probabilistic knowledge fusion. In *KDD*,
            2014.

[FFG14]   Grace Fan, Wenfei Fan, and Floris Geerts. Detecting errors in numeric
            attributes. In *WAIM*, 2014.

[FFP10]   Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Querying and
            repairing inconsistent numerical databases. *TODS*, 35(2), 2010.

[FFTD15]   Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs.
            PVLDB, 8(12), 2015.

[FG12]   Wenfei Fan and Floris Geerts. *Foundations of Data Quality Manage-
            ment*. Morgan & Claypool Publishers, 2012.

[FGJ⁺11]   Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

[FGJK08]   Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(1), 2008.

[FGMM10]   Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. Detecting inconsistencies in distributed data. In *ICDE*, 2010.

[FHK07]   Fedor V. Fomin, Pinar Heggernes, and Dieter Kratsch. Exact algorithms for graph homomorphisms. *Theory Comput. Syst.*, 41(2):381–393, 2007.

[FHT17]   Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[FL17]   Wenfei Fan and Ping Lu. Dependencies for graphs. In *PODS*, 2017.

[FLL⁺17]   Wenfei Fan, Xueli Liu, Ping Lu, Yinghui Wu, Jingbo Xu, Luara Chen, and Demai Ni. Discovering graph functional dependencies. Manuscript, 2017.

[FLM⁺11]   W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[FLM⁺12]   Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2):213–238, 2012.

[FLTY12]   Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. Incremental detection of inconsistencies in distributed data. In *ICDE*, 2012.

[FPL⁺01]   Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR*, 2001.

[FWW13]   Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.

[FWW14a]   Wenfei Fan, Xin Wang, and Yinghui Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 7(12), 2014.

[FWW14b]   Wenfei Fan, Xin Wang, and Yinghui Wu. Querying big graphs within bounded resources. In *SIGMOD*, 2014.

[FWX16]   Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In *SIGMOD*, 2016.

[GBDS14]   Ivana Grujic, Sanja Bogdanovic-Dinic, and Leonid Stoimenov. Collecting and analyzing data from e-government facebook pages. In *ICT Innovations*, 2014.

[GFMPdlF11] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD workshop*, 2011.

[GG14]   Eric L. Goodman and Dirk Grunwald. Using vertex-centric programming platforms to implement SPARQL queries on large graphs. In *IA3*, 2014.

[GJ79]   Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GJM96]   Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.

[GJS76]   Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[GKK+09]   Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. Sequential dependencies. *PVLDB*, 21(1), 2009.

[GM12]   Lise Getoor and Ashwin Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12), 2012.

[GMS93]   Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[GTHS13]   Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[Guh14]   R. V. Guha. Communicating and resolving entity references, 2014. *http://arxiv.org/abs/1406.6973*.

[GXH+12]   Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *IMC*, 2012.

[HAR11]   Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[HCD+16]   Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *SoCC*, 2016.

[HHK95]   Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[HK97]   Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP*, 1997.

[HKM+12]   Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3, 2012.

[HKPT99]   Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[HLL13]   Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.

[HNST12]   M. Herschel, F. Naumann, S. Szott, and M. Taubert. Scalable iterative graph duplicate detection. *TKDE*, 24, 2012.

[HSW01]   Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.

[HWYY07]   Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[HZZ14]   Binbin He, Lei Zou, and Dongyan Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In *SWIM*, 2014.

[JLG16]   Galileo Mark S. Namata Jr., Ben London, and Lise Getoor. Collective graph identification. *TKDD*, 10(3):25:1–25:36, 2016.

[Jon80]   James P. Jones. Undecidable Diophantine equations. *Bull. Amer. Math. Soc.*, 3(2):859–862, 1980.

[KC12]   Mijung Kim and K. Selçuk Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowl. Eng.*, 72:285–303, 2012.

[KIJ+15]   Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, 2015.

[KL14]   Nitish Korula and Silvio Lattanzi. An efficient reconciliation algorithm for social networks. *PVLDB*, 7(5), 2014.

[KLCL13]   Song-Hyon Kim, Kyong-Ha Lee, Hyebong Choi, and Yoon-Joon Lee. Parallel processing of multiple graph queries using MapReduce. In *DBKDA*, 2013.

[KPC+05]   Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[KR98]   Harumi A. Kuno and Elke A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation. *TKDE*, 10(5):768–792, 1998.

[KRS90]   Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *TCS*, 71(1):95–132, 1990.

[KSSV09]   Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *ICDE*, 2009.

[KTR12a]   Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 2012.

[KTR12b]   Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for MapReduce-based entity resolution. In *ICDE*, 2012.

[KWA+14]   Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. Test-driven evaluation of linked data quality. In *WWW*, 2014.

[Lac13]   Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27, 2013.

[LGK+12]   Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[LHKL12]   Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[LKDL12]   Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012.

[LMC11]   Ni Lao, Tom Mitchell, and William W. Cohen. Random walk inference and learning in a large scale knowledge base. In *EMNLP*, 2011.

[LMS08]   Georg Lausen, Michael Meier, and Michael Schmidt. SPARQLing constraints for RDF. In *EDBT*, 2008.

[LQLC15]   Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable sub-graph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[MAS14]   Pankaj Malhotra, Puneet Agarwal, and Gautam Shroff. Graph-parallel entity resolution using LSH & IMM. In *EDBT/ICDT Workshops*, 2014.

[Mat93]   Yuri Matiyasevich. *Hilbert's 10th Problem*. The MIT Press, 1993.

[Met]   Metis. *http://glaros.dtc.umn.edu/gkhome/*.

[Mur]   Amelia Murray. Fake natwest twitter account targets customers to steal bank details. *http://www.telegraph.co.uk/money/consumer-affairs/fake-natwest-twitter-account- targets-customers-to-steal-bank-det*.

[MW95]   Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SICOMP*, 24(6), 1995.

[NCO04]   Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the Web? The evolution of the Web from a search engine per-spective. In *WWW*, 2004.

[Pap94]   Christos H Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PKS+10]   Nicoleta Preda, Gjergji Kasneci, Fabian M. Suchanek, Thomas Neu-mann, Wenjun Yuan, and Gerhard Weikum. Active knowledge: dy-namically enriching RDF knowledge bases by web services. In *SIG-MOD*, 2010.

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

[PSC+15]   Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. Combining quantitative and logical data clean-ing. *PVLDB*, 9(4):300–311, 2015.

[PSS13]   Nathalie Pernelle, Fatiha Saïs, and Danai Symeonidou. An automatic key discovery approach for data linking. *J. Web Sem.*, 23, 2013.

[PT89]   William Pugh and Tim Teitelbaum. Incremental computation via func-tion caching. In *POPL*, 1989.

[PZ82]   Christos Papadimitriou and Stathis Zachos. Two remarks on the power of counting. *Theoretical Computer Science*, pages 269–275, 1982.

[QYC$^+$14]   Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *SIGMOD*, 2014.

[RDG11]   Vibhor Rastogi, Nilesh Dalvi, and Minos Garofalakis. Large-scale collective entity matching. *PVLDB*, 2011.

[RR96a]   G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[RR96b]   G. Ramalingam and Thomas W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[RSSS98]   Kenneth A. Rossa, Divesh Srivastava, Peter J. Stuckeyc, and S. Sudarshan. Foundations of aggregation constraints. *TCS*, 193(1-2):149–179, 1998.

[RvRH$^+$14] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. PGX.ISO: Parallel and efficient in-memory engine for subgraph isomorphism. In *GRADES*, 2014.

[RZ04]   Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, 2004.

[Rza14]   Paweł Rzażewski. Exact algorithm for graph homomorphism and locally injective graph homomorphism. *Inf. Process. Lett.*, 114(7):387–391, 2014.

[Sah07]   Diptikalyan Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.

[SC11]   Shaoxu Song and Lei Chen. Differential dependencies: Reasoning and discovery. *TODS*, 36(3):16, 2011.

[SCYC14]   Shaoxu Song, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 2014.

[SKW07]   Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[SNA]   SNAP. *http://snap.stanford.edu/data/index.html*.

[SNS09]   A. Stotz, R. Nagi, and M. Sudit. Incremental graph matching for situation awareness. *FUSION*, 2009.

[Spa05]   Holger Spakowski. *Completeness for parallel access to NP and counting class separations*. PhD thesis, University of Düsseldorf, Germany, 2005.

[SPSL13]   Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 2013.

[SSW09]   Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. SOFIE: a self-organizing framework for information extraction. In *WWW*, 2009.

[SU02]   Marcus Schaefer and Christopher Umans. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT news*, 33(3):32–49, 2002.

[SWW$^+$12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.

[Tar72]   Robert Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.

[TR81]   Tim Teitelbaum and Thomas W. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[VCSM14]   Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[Wag90]   Klaus W Wagner. Bounded query classes. *SICOMP*, 19(5):833–846, 1990.

[WMW17]   Xiaolan Wang, Alexandra Meliou, and Eugene Wu. QFix: Diagnosing errors through query histories. In *SIGMOD*, pages 1369–1384, 2017.

[WP14]   Dominik Wienand and Heiko Paulheim. Detecting incorrect numerical
              data in dbpedia. In *ESWC*, 2014.

[WZ13]   David P. Woodruff and Qin Zhang. When distributed computation is
              communication expensive. In *DISC*, 2013.

[YEN$^+$11]   Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad
              Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, pages 279–
              289, 2011.

[YH11]   Yang Yu and Jeff Heflin. Extending functional dependency to detect
              abnormal data in RDF graphs. In *ISWC*, 2011.

[YQC10]   Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational
              databases: A survey. *IEEE Data Eng. Bull.*, 33(1), 2010.

[ZGM98]   Yue Zhuge and Hector Garcia-Molina. Graph structured views and
              their incremental maintenance. In *ICDE*, 1998.