GRAPE: Parallel Graph Query Engine

Jingbo Xu

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

The need for graph computations is evident in a multitude of use cases. To support computations on large-scale graphs, several parallel systems have been developed. However, existing graph systems require users to recast algorithms into new models, which makes parallel graph computations as a privilege to experienced users only. Moreover, real world applications often require much more complex graph processing workflows than previously evaluated. In response to these challenges, the thesis presents GRAPE, a distributed graph computation system, shipped with various applications for social network analysis, social media marketing and functional dependencies on graphs.

Firstly, the thesis presents the foundation of GRAPE. The principled approach of GRAPE is based on partial evaluation and incremental computation. Sequential graph algorithms can be plugged into GRAPE with minor changes, and get parallelized as a whole. The termination and correctness are guaranteed under a monotonic condition.

Secondly, as an application on GRAPE, the thesis proposes graph-pattern association rules (GPARs) for social media marketing. GPARs help users discover regularities between entities in social graphs and identify potential customers by exploring social influence. The thesis studies the problem of discovering top-k diversified GPARs and the problem of identifying potential customers with GPARs. Although both are NP-hard, parallel scalable algorithms on GRAPE are developed, which guarantee a polynomial speedup over sequential algorithms with the increase of processors.

Thirdly, the thesis proposes quantified graph patterns (QGPs), an extension of graph patterns by supporting simple counting quantifiers on edges. QGPs naturally express universal and existential quantification, numeric and ratio aggregates, as well as negation. The thesis proves that the matching problem of QGPs remains NP-complete in the absence of negation, and is DP-complete for general QGPs. In addition, the thesis introduces quantified graph association rules defined with QGPs, to identify potential customers in social media marketing.

Finally, to address the issue of data consistency, the thesis proposes a class of functional dependencies for graphs, referred to as GFDs. GFDs capture both attribute-value dependencies and topological structures of entities. The satisfiability and implication problems for GFDs are studied and proved to be coNP-complete and NP-complete, respectively. The thesis also proves that the validation problem for GFDs is coNP-complete. The parallel algorithms developed on GRAPE verify that GFDs provide an effective approach to detecting inconsistencies in knowledge and social graphs.

Lay Summary

The need for graph computations is evident in transportation network analysis, knowledge extraction, Web mining, social network analysis and social media marketing, among other things. Graph computations are, however, costly in real-life graphs. For instance, the social graph of Facebook has billions of nodes and trillions of edges. In such a graph, it is already expensive to compute shortest distances from a single source, not to mention graph pattern matching, which is intractable in nature.

To support graph computations in large-scale graphs, several parallel systems have been developed, *e.g.,* Pregel, GraphLab, Giraph++, GraphX. These systems, however, do not allow us to reuse existing sequential graph algorithms, which have been studied for decades and are well optimized. To use Pregel, for instance, one has to "think like a vertex" and recast existing algorithms into a vertex-centric model; similarly when programming with other systems, *e.g.,* Blogel, which adopts vertex-centric programming by treating blocks as vertices. The recasting is nontrivial for people who are not very familiar with the parallel models. This makes parallel graph computations a privilege of experienced users only.

Is it possible to make parallel graph computations accessible to users who only know conventional graph algorithms covered in undergraduate textbooks? Can we have a system such that given a graph computation problem, we can "plug in" its existing sequential algorithms as a whole, without recasting or "thinking in parallel", and the system automatically parallelizes the computation across multiple processors? Moreover, can the system guarantee that the parallelization terminates and converges at correct answers as long as the sequential algorithms plugged in are correct? Furthermore, can the system inherit optimization techniques well developed for sequential graph algorithms, such as indexing and compression? Better yet, despite the ease of programming, can the system achieve performance comparable to the state-of-the-art parallel graph system?

These questions motivate us to develop GRAPE, a parallel GRAPh query Engine. GRAPE has the unique ability to parallelize existing sequential graph algorithms as a whole. Sequential graph algorithms can be plugged into GRAPE with minor changes, and get parallelized. The termination and correctness are guaranteed under a monotonic condition.

As an application on GRAPE, the thesis proposes graph pattern association rules (GPARs) for social media marketing. GPARs help users discover regularities between

entities in social graphs and identify potential customers by exploring social influence.

To make the pattern matching on GRAPE more expressive, the thesis also proposes quantified graph patterns (QGPs), an extension of graph patterns by supporting simple counting quantifiers on edges while does not introduce extra complexity for its matching problem. Moreover, the thesis introduces quantified graph association rules defined with QGPs, to identify potential customers in social media marketing.

Finally, to address the issue of data consistency, the thesis proposes a class of functional dependencies for graphs, referred to as GFDs. GFDs capture both attribute-value dependencies and topological structures of entities. The algorithms on GRAPE verify that GFDs provide an effective approach to detecting inconsistencies in knowledge and social graphs.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my principal supervisor, Professor Wenfei Fan, for all his support, guidance, trust, encouragement, and inspiration. He led me into the area of database study and taught me about every perspective of the principles of research. His talent, self-discipline, and hard work inspired me to complete this PhD. Without his supervision and unfailing support, this dissertation would not have been possible.

I am very grateful to Professor Xudong Liu, my supervisor at Beihang University, who gave me the opportunity to enrol in this PhD programme. He also gave me great support regarding my research in Edinburgh and was always available to help in many ways. I would also like to express my gratitude to Professor Leonid Libkin, my second supervisor at the University of Edinburgh. I learned a great deal about complexity and logic from his solid work.

Special gratitude goes to Dr Yinghui Wu and Dr Wenyuan Yu, who helped me with their constructive advice and supportive encouragement when we cooperated on many projects. I am very grateful to my collaborators and colleagues, including Dr Xin Wang, Dr Yang Cao, Chao Tian, Xueli Liu, and all the other members of the database group for their extended support. I really enjoyed working with them.

I would like to express my heartfelt thanks to Sifeng Gu, my best friend in Edinburgh. His enthusiasm for research and his self-motivation encouraged me to work hard. He changed my life in Edinburgh and affected me in many ways. Thanks go to Jing Yu and Yi Xiao, who helped me during a difficult time before I enrolled in this programme, and to Mingming Zhang, Dr Wen Zhang, Dr Zide Meng, and Shuang Ma for the supporting with their own experience in doing PhD. I also acknowledge my friends in Edinburgh, including Nan Su, Shuming Zhang, PinLiang Xiong, Hanlu Fang, Feiyang Deng, Yixuan Wang, Junyu Huang, Angus Liong, and Mingxin Wang. I enjoyed the time spent with them in Edinburgh.

Finally, I would like to thank my parents and Zongcheng. Their support and encouragement were ultimately what made this thesis possible.

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.

(*Jingbo Xu*)

Table of Contents

1 Introduction **1**

1.1 Motivations . . . 1

1.2 Graph Systems: State of The Art . . . 4

1.3 Outline of Thesis . . . 7

1.4 Contributions . . . 7

1.5 Publication List . . . 11

2 Framework and Foundation of GRAPE **13**

2.1 Preliminaries . . . 15

2.2 Programming with GRAPE . . . 17

2.2.1 The Parallel Model of GRAPE . . . 17

2.2.2 PEval: Partial Evaluation . . . 18

2.2.3 IncEval: Incremental Evaluation . . . 20

2.2.4 Assemble Partial Results . . . 23

2.2.5 GRAPE API . . . 23

2.3 Foundation of GRAPE . . . 25

2.3.1 Correctness of Parallel Model . . . 25

2.3.2 The Power of GRAPE . . . 28

2.4 Graph Computations in GRAPE . . . 32

2.4.1 Graph Pattern Matching . . . 32

2.4.2 Graph Connectivity . . . 34

2.4.3 Collaborative Filtering . . . 35

2.5 Implementation of GRAPE . . . 38

2.6 Experimental Study . . . 42

2.7 Related Work . . . 55

2.8 Summary . . . 57

3 Association Rules Discovery on GRAPE **58**

3.1 Association via Graph Patterns 62

3.1.1 Graphs, Patterns, and Pattern Matching 62

3.1.2 Graph Pattern Association Rules 64

3.2 Support and Confidence 66

3.3 Diversified Rule Discovery 70

3.3.1 The Diversified Mining Problem 70

3.3.2 Discovery Algorithm 72

3.4 Identifying Customers 80

3.4.1 The Entity Identification Problem 80

3.4.2 Optimization Strategies 82

3.5 Experimental Study 85

3.6 Related Work 92

3.7 Summary 94

4 Extending Pattern Matching on GRAPE with Quantifiers **95**

4.1 Quantified Graph Patterns 98

4.1.1 Conventional Graph Pattern Matching 98

4.1.2 Quantified Graph Patterns 98

4.2 The Complexity of Quantified Matching 104

4.3 Algorithms for Quantified Matching 108

4.3.1 Quantified Graph Pattern Matching 109

4.3.2 Incremental Quantified Matching 113

4.4 Parallel Quantified Matching 117

4.4.1 Parallel Scalability 117

4.4.2 Parallel Scalable Algorithm 117

4.5 Quantified Association Rules 125

4.6 Experimental Study 129

4.7 Related Work 136

4.8 Summary 138

5 Functional Dependencies on Graphs **139**

5.1 Preliminaries 142

5.2 GFDs: Syntax and Semantics 145

5.3 Reasoning about GFDs 149

5.3.1 The Satisfiability Problem for GFDs 149

5.3.2 The Implication Problem for GFDs . . . 152
5.4 Inconsistency Detection . . . 155
5.4.1 GFD Validation and Error Detection . . . 155
5.4.2 Parallel Scalability . . . 156
5.5 Parallel Algorithms . . . 159
5.5.1 Parallel Algorithm for Replicated Graphs . . . 159
5.5.2 Algorithm for Fragmented Graphs . . . 163
5.6 Experimental Study . . . 166
5.7 Related Work . . . 173
5.8 Summary . . . 177

6 Conclusion and Future Work **178**
6.1 Conclusion . . . 178
6.2 Future Work . . . 179

Bibliography **181**

List of Figures

2.1 GRAPE workflow 17
2.2 Algorithm PEval for SSSP 21
2.3 Algorithm IncEval for SSSP 22
2.4 GRAPE interface 38
2.5 GRAPE architecture 39
2.6 Performance evaluation of SSSP 44
2.7 Performance evaluation of CC 46
2.8 Performance evaluation of Sim 47
2.9 Performance evaluation of SubIso 48
2.10 Performance evaluation of CF 49
2.11 Scalability on synthetic graphs 50
2.12 Incremental steps and optimization 51
2.13 Giraph vertex program for SSSP 52
2.14 Blogel block program for SSSP 53

3.1 Associations as graph patterns 60
3.2 Labeled social graphs 63
3.3 Diversified GPARs 71
3.4 Algorithm DMine 74
3.5 Parallel scalability of DMine 86
3.6 Effectiveness of DMine 88
3.7 Performance evaluation of Match 90

4.1 Quantified graph patterns 97
4.2 Examples of social graph 99
4.3 Negative QGPs 101
4.4 Generic search procedure Match 108
4.5 Algorithm QMatch 110

4.6 Algorithm PQMatch 121
4.7 QGARs 125
4.8 Response time and scalability for quentified match 130
4.9 Parallel scalability of PQMatch 131
4.10 Impact of negative edges and aggregate on PQMatch 132
4.11 Real-world QGARs 134

5.1 Graphs with dependencies 141
5.2 Graph patterns 143
5.3 Graph patterns in GFDs 150
5.4 Algorithm repVal 160
5.5 Parallel scalability and communication 167
5.6 Workload complexity 169
5.7 Scalability: Varying $|G|$ (synthetic) 170
5.8 Real-life GFDs 171

List of Tables

2.1 Notations in Chapter 2 16

3.1 Notations in Chapter 3 69
3.2 Running example for DMine, round 1 76
3.3 Running example for DMine, round 2 76
3.4 Prediction precision 89

4.1 Notations in Chapter 4 103
4.2 Running example for QMatch, quantifier check 111
4.3 Running example for QMatch, next verificaiton 112
4.4 Computation of IncQMatch 115
4.5 Vertex requests in DPar 121
4.6 Cost estimation in DPar 122

5.1 Notations in Chapter 5 144
5.2 Running time and accuracy 172

Chapter 1

Introduction

With the emergence of big data, the graph model has drawn great attention both in academia and industry in the past decade. Graphs provide strong modelling for complex applications and rich querying. Graph systems outperform relational database for the cases that require costly join operations since graph stores and processes connections as first citizens. In light of these, many systems and frameworks are designed and developed for graph computations. However, to handle graphs with billions or even trillions of nodes and edges is non-trivial. Particularly, querying and analytics on very large graph efficiently is a challenging task.

In this dissertation, we propose a parallel graph computation engine, named GRAPE, with a variety of applications used in real-life graph data, and study the related problems. In this chapter, we present the motivations for this thesis, review existing graph systems, describe the main results of the work and give an outline of this thesis.

1.1 Motivations

Graphs make an important source of big data and have proved to be prevalently used in social marketing, knowledge discovery, transportation/mobile network analysis, machine learning, and other disciplines like chemistry and bioinformatics.

The social graph is one of the most important kinds of graph data. Social graphs depict personal relations of internet users, and other things such as posts, pictures, and comments. The graph-shape is natural to organise and express the rich information generated by users and the interactions between them. Analysing and deriving new insights from the social graphs is attractive to big web companies. They research on a wide range of use cases, including collaborative filtering to give accurate recommenda-

tions for users sharing common interests, friends of friends to connect more users and expand one's social circle, as well as fraud detection to find deeply hidden criminals by analysis users' abnormal interactions.

The knowledge base is another kind of big graph. It uses graphs to store and manage a massive of complex structured and unstructured information in computer systems. Data is organised under an ontology schema aimed to support computational tasks. Many knowledge base projects are established, *e.g.,* DBpedia [dbp], Freebase [fre], and Yagos [SKW07]. In knowledge bases, information is extracted as RDF triples and added as properties of the corresponding URI. The knowledge base is fundamental to the semantic web and the Internet of things. It helps computers retrieve and understand semantic information in specific domains. For instance, Google uses a knowledge graph to enhance its search results by gathering information from a variety of sources about a topic [goo].

No matter where the graphs come from, on the abstract level, graphs could be represented as a set of vertices and a set of edges. The vertices usually represent the entities, while the edges represent the relationships. These analytics, explorations, searches and computations on the graph could be unified and termed queries on graphs. We formalise the problem: Consider a class $\mathcal{Q}$ of graph queries, such as graph connectivity, (weak or strong connected component), graph traversal, (breadth-first search or depth-first search), graph pattern matching (via graph simulation or subgraph isomorphism) and graph search. Given a query $Q \in \mathcal{Q}$ and a data graph G, the problem of querying graphs is to compute the answer $Q(G)$ to Q in G.

There are several parallel graph systems developed for solving graph query, *e.g.,* GraphLab [LGK$^+$10], Giraph [Ave11], Giraph++ [TBC$^+$13], GraphX [GXD$^+$14], Blogel [YCLN14] and Trinity [SWL12]. Although they enjoy great popularity in the industry and the community, some issues are recognised, *e.g.,* the inefficiency IO cost for MapReduce, excessive message passing and lack of global optimisation in the vertex-centric systems, and the need to rewrite existing algorithms in the new model. We can bear these, but graph computations have been studied for decades, and many sequential algorithms already exist. Can we reuse them in the parallel setting with minor revision and guarantee the termination and correctness, without drastic degradation in performance or functionality compared with other parallel graph systems?

Moreover, there are some well-known graph computations such as reachability query, page rank and triangle counting. However, with the rich profile information in the social network, one may want to evaluate some more complex queries beyond

these.

One of the applications is association rules considering the social relationships. The need for studying associations between entities in social graphs is evident, especially in social media marketing. Social media marketing is predicted to trump traditional marketing. Indeed, "Consumers are 92% more likely to trust their peers over advertising when it comes to purchasing decisions" [Lit], "60% of users said Twitter plays an important role in their shopping" [Smi13], and "the peer influence from ones friends causes more than 50% increases in odds of buying products" [BU12]. Can we provide an application based on GRAPE to identify potential customers with the help of association rules on the social graph data?

Inspecting the associations in social networks raises another problem. Associations in the real world are usually too complicated to be expressed in a single graph pattern. More expressive patterns are needed, notably ones with counting quantifiers. However, adding counting quantifiers to graph patterns poses several questions: How to define the quantified graph pattern, to strike a balance between expressive power and complexity? Can we efficiently conduct graph pattern matchings with quantifiers? How to employ these expressive patterns in the emerging applications?

Another issue of real life graphs is data quality. It is widely recognised that real-life data is dirty: "more than 25% of critical data in the worlds top companies is flawed" [dat] and dirty data may lead to strategical mistakes. For relational data, a variety of dependencies have been studied, such as conditional functional dependencies (CFDs) [FGJK08] and denial constraints [BCD03]. Employing the dependencies, a batch of techniques have been developed to detect errors in relational data and repair the data. However, the study on dependencies on graphs is still in its infancy. There lacks a comprehensive study on the quality on graphs while it is indeed needed. For instance, to build a knowledge base with high quality, effective methods must be developed to catch the inconsistencies.

In summary, real-life scale graphs introduce new challenges to query evaluation, data mining and data cleaning, among other things. They demand a departure from theory to systems and applications and call for new techniques to query big graphs, identify associations among entities and improve data quality. In this thesis, we aim to (1) propose a parallel graph system with new query evaluation approaches (2) study its application for identifying association rules with graphs (3) extend graph pattern matching to make it more expressive in social marketing, and (4) study the data consistency problem on graphs.

1.2 Graph Systems: State of The Art

GraphLab and PowerGraph. In 2010, CMU revealed GraphLab [LGK+10], which is designed to target the general Machine Learning (ML) problems. By inspecting the common patterns in ML problems, the authors found that the different convergence speed between vertexes limits the parallel performance. Based on this, they proposed an asynchronised execution model for iterative graph computation. GraphLab is the one that gives the concept of vertex-centric computation, which leads to a fine-grained graph algorithm expression, and has been inherited in many subsequent systems. GraphLab also offers a set of elaborately designed concurrency control models to ensure the consistency in the asynchronised model.

GraphLab has attracted significant interest, both in academia and industry. As a consequence, the research team has extended it to focus on large power-law graphs in distributed settings: specifically, PowerGraph [GLG+12]. In the PowerGraph abstraction, a vertex programme could read states from its neighbours directly to archive shared memory illusion. PowerGraph also introduces a Gather-Apply-Scatter model to decompose the vertex programme into three phases: In the gather phase, a vertex access states from its neighbours with incoming edges aggregate them; In the apply phase, the vertex program applied the aggregated messages and processed the computation logic; during the last scatter phase, the computation result is scatted along its outgoing edges.

PowerGraph is designed to process large scale graph from the real world. One observation is that some particular vertices are converging slowly due to their extremely high degree. PowerGraph introduced a new partitioning model name vertex-cut to address this issue. Different from edge-cut, vertex-cut split and mirror vertices rather than edges. By cutting a small fraction of the very high degree vertices, users of PowerGraph can quickly shatter a graph and achieve a good performance.

Many systems are following the principle or implementation of GraphLab. These systems include GraphChi [KBG12], a single-machine fitted graph processing system with a particular access design; PowerLyra [CSCC15], which is based on GraphLab but supports hybrid-cut of partitioning strategy; and PowerSwitch [XCG+15], supports fast and seamless switches between sync and async modes by dynamically estimating the cost.

Pregel and Giraph. Google has revealed its MapReduce [DG08] and Google File System [GGL03] computing abstractions, which have proved to be a good practice for big

data. So was Pregel [MAB+10]. Pregel is a distributed graph processing system proposed by Google. Programmes on a graph are divided into a sequence of iterations, in which a vertex sends or receives messages, processes them and changes its status. The computation terminates when each of the vertices have voted to halt.

Pregel is designed for efficient, scalable and fault-tolerant implementations on clusters. It runs in a Bulk Synchronous Parallel(BSP) [Val90] model, which synchronises the status of vertices between each iteration and has made the reasoning of algorithms easier than Graphlab.

The vertex-centric approach is working here. Based on the success of MapRecude, users are invited to focus on a local action, rather than thinking in a whole scope. Pregel takes the responsibility for processing the independent actions on each vertex to lift a whole computation.

Google has claimed that Pregel is used in many products. However, Google did not make it open-source. Apache Giraph [Ave11] is a community implementation of Pregel. Besides Pregel, Giraph also incorporates several other features such as master computation and out-of-core computation. With a steady and active community, Giraph is used by many network companies. For example, Facebook has claimed that it uses Giraph to process social relationships at a scale of trillion edges [CEK+15]. To the best of our knowledge, this is the largest size reported in a real application of graph computation. Facebook explains that Giraph is their first choice to for processing big data, since its Java implementation and based on MapReduce, which is in the pipeline of the existing Hadoop infrastructure widely used in the company.

Neo4j. Focusing on the graph data storage and management, Neo4j is an open-source No-SQL graph database implemented in Java and Scala. [Mil13] It provides full characteristics of databases such as ACID transaction compliance, cluster support and runtime failover. It models the data as a property graph, which contains connected entities with various attributes and can be labelled as different roles in a domain. Neo4j shipped with a declarative language, Cypher. Cypher allows users to state what they want to select, update, delete and insert in the graph without being required to describe how to do this. Neo4j has a rich ecosystem with connectors to other big data analytical frameworks, such as Apache Spark, Docker and Cassandra. These connectors support exporting of selected graph data to analytics platforms and writing back to the Neo4j as persistent data, which makes it a full-fledged system.

Blogel and Giraph++. While the vertex-centric framework introduces straightforward

programming logic, it is easy to observe the overheads of communication and the limitations on random access for the vertices. Some systems extend the vertex-centric model and adopt a block based model. These include Blogel and Giraph++. They partition vertices into multiple disjoint subgraphs. Within each subgraph, the vertex data propagation can bypass the network interface and access the memory to improve the performance.

Giraph++ [TBC^{+}13] is claimed to be a new graph-centric model, and is implemented based on Giraph. To overcome the limitations of the slow propagation and short-sightedness for the neighbourhood in the vertex-centric model, it opens the partition structure to the users and allows messages within a fragment to flow freely.

Blogel [YCLN14] is another block-centric graph system. The authors proposed a model aggregating vertices in the same connected component into large vertex, namely block, to resolve the large diameter and high-density problems in real-world graphs. Blogel supports three types of computing modes: the vertex mode, block mode and vertex-block mix mode. An application can select from these modes for its computing phrases. In some graphs, Blogel achieves orders of magnitude performance improvements over other vertex-centric systems.

Other graph systems There are several other graph systems designed to solve graph computations from different perspectives. Microsoft Trinity [SWL12] provides fast random data access by employing a memory cloud and providing a unified address space. Users explore graphs through Trinity API as if the data is stored in the memory of a single machine. The detail storage and partition management is transparent for users. GraphX [XGFS13] is a graph processing component in Apache Spark. It introduces some small dataflow operators to materialise graph views and express existing graph APIs. By recasting graph-specific optimisations as general-purpose data storage optimisations and query evaluation optimisations, it benefits from the existing techniques which have been studied for decades in the database community.

1.3 Outline of Thesis

The remainder of the thesis is organised as follows.

Chapter 2 proposes the framework and design of GRAPE. It introduces the principled approach of GRAPE, which is based on partial evaluation and incremental evaluations. It studies the correctness and termination problems. In addition, it gives the programming interfaces and some details in the implementation.

Chapter 3 presents an application on GRAPE, namely association rules with graph patterns. Top-k diversified discovery problem and potential customer identify problem are studied.

Chapter 4 revisits the graph pattern matching, and formally defines graph pattern matching with quantifiers. It also discusses the complexity of quantified graph matching. Parallel scalable algorithms for efficiently identifying potential customers are developed and verified.

Chapter 5 introduces a class of functional dependencies for graphs. It gives formal definition and discusses satisfiability and implication problems. Algorithms for catching violations on GRAPE with parallel scalability are developed and verified in experiments.

Chapter 6 concludes this thesis.

1.4 Contributions

We summarise the contributions of this work as follows:

Contributions of Chapter 2. We propose GRAPE, from foundation to implementation, to parallelize sequential graph algorithms.

(1) We introduce the parallel model of GRAPE, by combining partial and (bounded) incremental evaluation (Section 2.2). We also present the programming model of GRAPE. We show how to plug in *existing* sequential algorithms for GRAPE to parallelize the entire algorithms, in contrast to parallelization of instructions or operators [RMM15, MMS14].

(2) We prove two fundamental results (Section 2.3): (a) Assurance Theorem guarantees GRAPE to terminate with correct answers under a monotonic condition when its input sequential algorithms are correct; and (b) Simulation Theorem shows that MapReduce [DG08], BSP (Bulk Synchronous Parallel) [Val90] and PRAM (Parallel Random

Access Machine) [Val91] can be optimally simulated by GRAPE. Hence algorithms for existing graph systems can be migrated to GRAPE.

(3) We show that a variety of graph computations can be readily parallelized in GRAPE (Section 2.4). These include graph traversal (shortest path queries SSSP), pattern matching (via graph simulation Sim and subgraph isomorphism SubIso), connected components (CC), and collaborative filtering (CF in machine learning). We show how GRAPE easily parallelizes their sequential algorithms with minor revisions.

(4) We outline an implementation of GRAPE (Section 2.5). We show how GRAPE supports parallelization, message passing, fault tolerance and consistency. We also show how easily GRAPE implements optimization such as indexing, compression and dynamic grouping, which are not supported by the state-of-the-art vertex-centric and block-centric systems.

(5) We experimentally evaluate GRAPE (Section 2.6), compared with (a) Giraph, a open-source version of Pregel, (b) GraphLab, an asynchronous vertex-centric system, and (c) Blogel, the fastest block-centric system we are aware of. Over real-life graphs, we find that in addition to the ease of programming, GRAPE achieves comparable performance to the state-of-the-art systems.

Contributions of chapter 3. We propose GPARs, and provide effective algorithms for discovering and applying GPARs.

(1) We introduce graph-pattern association rules (GPARs) for social media marketing (Section 3.1). GPARs differ from conventional rules for itemsets in both syntax and semantics. A GPAR defines its antecedent as a graph pattern, which specifies associations between entities in a social graph, and explores social links, influence and recommendations. It enforces conditions via both value bindings and topological constraints by subgraph isomorphism.

(2) We define topological support and confidence metrics for GPARs (Section 3.2). Conventional support for itemsets is no longer anti-monotonic for GPARs. We define support in terms of distinct "potential customers" by revising a measure proposed by [BN08]. We propose a confidence measure for GPARs by revising Bayes Factor [LTP07] to incorporate the local closed world assumption [GTHS13, Don14]. This allows us to cope with (incomplete) social graphs, and to identify interesting GPARs

with correlated antecedent and consequent.

(3) We study a new mining problem, referred to as the *diversified mining problem* and denoted by DMP. It is a bi-criteria optimization problem to discover top-k GPARs. While useful, DMP is NP-hard. Nonetheless, we develop a parallel approximation algorithm with a *constant accuracy bound*. We also provide optimization methods to filter redundant or non-promising rules as early as possible.

(4) We also study how to identify potential customers by applying GPARs, referred to as *the entity identification problem* and denoted by EIP. Given a social graph G and *a set* Σ of GPARs pertaining to an event $p(x,y)$, we identify potential customers x of y in G with confidence above a given bound η, by using GPARs in Σ. We show that it is NP-hard even to decide whether such x exists.

Despite this, we develop a *parallel scalable* algorithm for EIP such that its response time is in $O(t(|G|,|\Sigma|)/n)$, a polynomial reduction in the running time $t(|G|,|\Sigma|)$ of *sequential algorithms*, by using n processors. Hence given a big graph, we can identify potential customers in it by increasing n.

(5) Using real-life and synthetic graphs, we experimentally verify the scalability and effectiveness of our algorithms (Section 3.5). We show that, despite their complexity, applying and discovering GPARs are feasible in practice via parallelization.

Contributions of Chapter 4. We extend graph pattern matching with quantifiers.

(1) We propose QGPs (Section 4.1). Using simple counting quantifiers, QGPs uniformly support numeric and ratio aggregates, universal and existential quantification, and negation. We formalize *quantified matching*, *i.e.,* graph pattern matching with QGPs, by revising the traditional semantics of pattern matching to incorporate counting quantifiers.

(2) We establish the complexity of quantified matching (Section 4.2). We show that despite their increased expressiveness, QGPs do not make our lives much harder: quantified matching is NP-complete in the absence of negation, the same as subgraph isomorphism; and it is DP-complete otherwise.

(3) We provide a quantified matching algorithm (Section 4.3). The algorithm unifies conventional pattern matching and quantifier verification in a generic search process, and handles negation by novel incremental evaluation IncQMatch. As opposed to con-

ventional incremental settings, IncQMatch acts in response to changes in patterns, not in graphs, and is *optimal* by performing only necessary verification.

(4) We develop parallel algorithms for quantified matching (Section 4.4). We identify a practical condition under which quantified matching is *parallel scalable*, *i.e., guaranteeing* provable reduction in sequential running time with the increase of processors. Under the condition, we develop graph partition and QGP matching algorithms, both parallel scalable, by exploring inter and intra-fragment parallelism.

(5) As an application of QGPs, we introduce quantified graph association rules (QGARs; Section 4.5). QGARs help us identify potential customers in social graphs, and (positive and negative) correlations in knowledge graphs. We propose support and confidence metrics for QGARs, a departure from their conventional counterparts. We also show that the (parallel) quantified matching algorithms can be readily extended to identify interesting entities with QGARs.

(6) Using real-life and synthetic graphs, we experimentally verify the effectiveness of QGPs and the scalability of our algorithms (Section 4.6). We find that quantified matching is feasible and parallel scalable. And QGARs are able to capture behaviour patterns in social and knowledge graphs that cannot be expressed with conventional graph patterns.

Contributions of Chapter 5. We study functional dependencies for graphs, from their fundamental problems to applications.

(1) We propose a class of functional dependencies for graphs, referred to as GFDs (Section 5.2). As opposed to relational FDs, a GFD specifies two constraints: (a) a topological constraint in terms of a graph pattern (Section 5.1), to identify entities on which the dependency is defined, and (b) an extension of CFDs to specify the dependencies of the attribute values of the entities. We show that GFDs subsume FDs and CFDs as special cases, and capture inconsistencies between attributes of the same entity and across different entities.

(2) We settle two classical problems for reasoning about GFDs. For a set Σ of GFDs, we study (a) its satisfiability, to decide whether there exists a non-empty graph that satisfies all the GFDs in Σ, and (b) its implication, to decide whether a GFD is entailed by Σ. We show that the satisfiability and implication problems for GFDs are coNP-complete and

NP-complete, respectively. The results tell us that reasoning about GFDs is no harder than their relational counterparts such as CFDs, which are also intractable [FGJK08].

(3) As one of applications of GFDs, we study the validation problem, to detect errors in graphs by using GFDs as data quality rules (Section 5.4). We show that it is coNP-complete to decide whether a graph contains no violation of a set of GFDs. Despite the intractability, we develop algorithms that are *parallel scalable*, *i.e.,* they guarantee to take less time when more processors are used. They are 2-approximation algorithms for a bi-criteria optimization problem, to balance workload and minimize communication costs (Section 5.4). These make it feasible to detect errors in large-scale graphs.

(4) Using real-life and synthetic graphs, we experimentally verify the effectiveness and efficiency of our GFD techniques (Section 5.6). We find that the inconsistency detection with GFDs is feasible in real-life graphs. And GFDs catch a variety of inconsistencies in real-life graphs, validating the need for combining topological constraints and value dependencies.

1.5 Publication List

During the course of the PhD study, as a co-author, I have published the following publications as a co-author that are relevant to this thesis.

- [FXW$^+$17a] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang. GRAPE: Parallelizing Sequential Graph Computations. In the 43rd Proceedings of the VLDB Endowment (PVLDB), Demo, 2017. The Best Demo Award recipient.

- [FXW$^+$17b] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, Chao Tian. Parallelizing Sequential Graph Computations. ACM SIG Conference on Management of Data (SIGMOD), 2017. The Best Paper Award recipient.

- [FWX16b] Wenfei Fan, Yinghui Wu, Jingbo Xu. Functional Dependencies for Graphs. ACM SIG Conference on Management of Data (SIGMOD), 2016.

- [FWX16a] Wenfei Fan, Yinghui Wu, Jingbo Xu. Adding Counting Quantifiers to Graph Patterns. ACM SIG Conference on Management of Data (SIGMOD), 2016.

- [FWWX15] Wenfei Fan, Xin Wang, Yinghui Wu, Jingbo Xu. Association Rules with Graph Patterns. In the 41st Proceedings of the VLDB Endowment (PVLDB), 2015.

Remark. It is worth mentioning that the (partial) results of this thesis appeared in the above publications: (1) The results in Chapter 2 appeared in SIGMOD 2017 [FXW$^+$17b], I took part in the design of the framework and algorithms, implemented the framework and conducted the experiments. (2) The results in Chapter 3 appeared in VLDB 2015 [FWWX15], I jointly developed the discovery algorithms, refined the model, and carried out the experiments. (3) The results in Chapter 4 are taken from SIGMOD 2016 [FWX16a], I participated in the development of quantified patterns and the quantified matching algorithms, and conducted the experiments. (4) The results in Chapter 5 have been previously published in SIGMOD 2016 [FWX16b]. I was one of the developers of the algorithms under the two graph models, and experimentally verified their efficiency and effectiveness of the algorithms.

Chapter 2

Framework and Foundation of GRAPE

We have witnessed several graph systems emerging in the recent years, which attracted considerable interest in the large network companies. However, these systems required users to recast the existing algorithms into new models, which is often too restrictive for users who may not qualify algorithm design. This suggests a new approach to design the distributed graph computation system.

In this chapter, we propose the prototype of GRAPE. Based on the partial evaluation and incremental evaluation, we show that sequential algorithms could be easily plugged into GRAPE with minor changes and get parallelized. We also study the correctness conditions and simulation theorems of GRAPE. At last, we give an outline of the implementation and experimentally verify that the parallelized sequential algorithm could achieve comparable performance without substantial degradation.

Several parallel graph systems have been developed for graph computations, *e.g.,* Pregel [MAB+10], GraphLab [LBG+12], Giraph++ [TBC+13] and Blogel [YCLN14]. These systems, however, require users to recast graph algorithms into their models. While graphs have been studied for decades and a number of sequential algorithms are already in place, to use Pregel, for instance, one has to "think like a vertex" and recast the existing algorithms into a vertex-centric model; similarly when programming with other systems. The recasting is non-trivial for people who are not very familiar with the parallel models. This makes these systems a privilege for experienced users only.

Is it possible to have a system such that we can "plug" sequential graph algorithms into it as a whole (subject to minor changes), and it parallelizes the computation across multiple processors, without drastic degradation in performance or functionality of existing systems?

To answer this question, we develop GRAPE, a parallel GRAPh Engine, for graph computations such as traversal, pattern matching, connectivity and collaborative filtering. It differs from prior graph systems in the following.

(1) Ease of programming. GRAPE supports a simple programming model. For a class $\mathcal{Q}$ of graph queries, users only need to provide three existing sequential (incremental) algorithms for $\mathcal{Q}$ with minor additions. There is *no need* to revise the logic of the existing algorithms, and it substantially reduces the efforts to "think in parallel". This makes parallel graph computations accessible to users who know conventional graph algorithms covered in undergraduate textbooks.

(2) Semi-automated parallelization. GRAPE parallelizes the sequential algorithms based on a combination of partial evaluation and incremental computation. It guarantees to terminate with correct answers under a monotonic condition, if the three sequential algorithms provided are correct.

(3) Graph-level optimization. GRAPE inherits all optimization strategies available for sequential algorithms and graphs, *e.g.,* indexing, compression and partitioning. These strategies are hard to implement for vertex programs.

(4) Scale-up. The ease of programming does not imply performance degradation. GRAPE could be easily scale up to hundreds of processors, which is total transparent to users.

We present its underlying principles in the following sections.

2.1 Preliminaries

We first review the basic notations.

Graphs. We consider graphs $G = (V, E, L)$, directed or undirected, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; (3) each node v in V (resp. edge $e \in E$) carries $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

Graph $G' = (V', E', L')$ is called a *subgraph of* G if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$ (resp. each edge $e \in E'$), $L'(v) = L(v)$ (resp. $L'(e) = L(e)$).

Subgraph G' is said to be *induced by* V' if E' consists of all the edges in G whose endpoints are both in V'.

Partition strategy. Given a number m, a strategy P partitions graph G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$, and F_i resides at processor P_i. Denote by

- $F_i.I$ the set of nodes $v \in V_i$ such that there is an edge (v', v) *incoming* from a node v' in F_j $(i \neq j)$;
- $F_i.O$ the set of nodes v' such that there exists an edge (v, v') in E, $v \in V_i$ and v' is in some F_j $(i \neq j)$; and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$; $\mathcal{F}.O = \mathcal{F}.I$.

The *fragmentation graph* G_P of G via P is an index such that given each node v in $\mathcal{F}.O$ (or $\mathcal{F}.I$), $G_\mathsf{P}(v)$ retrieves a set of $(i \mapsto j)$ if $v \in F_i.O$ and $v \in F_j.I$ with $i \neq j$. As will be seen shortly, G_P helps us deduce the directions of messages.

The notations of this chapter are summarized in Table 2.1.

Symbols	Notations
$\mathcal{Q}$, Q	a class of graph queries, query $Q \in \mathcal{Q}$
G	graph, directed or undirected
P_0, P_i	P_0: coordinator; P_i: workers ($i \in [1,n]$)
P	graph partition strategy
G_{P}	the fragmentation graph of G via P
$\mathcal{F}$	fragmentation $(F_1, \ldots, F_n)$
M_i	messages designated to worker P_i

Table 2.1: Notations in Chapter 2

2.2 Programming with GRAPE

We start with the parallel model of GRAPE, and then show how to program with GRAPE. Following BSP [Val90], GRAPE employs a *coordinator* P_0 and a set of m *workers* $P_1,\ldots,P_m$.

2.2.1 The Parallel Model of GRAPE

GRAPE supports data-partitioned parallelism. Given a partition strategy P and sequential PEval, IncEval and Assemble for a class $\mathcal{Q}$ of graph queries, GRAPE parallelizes the computations as follows. It first partitions G into $(F_1,\ldots,F_m)$ with P, and distributes F_i's across m shared-nothing *virtual workers* $(P_1,\ldots,P_m)$. It maps m virtual workers to n physical workers. When $n < m$, multiple virtual workers mapped to the same worker share memory. It also constructs fragmentation graph G_{P}. Note that G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ posed on G.

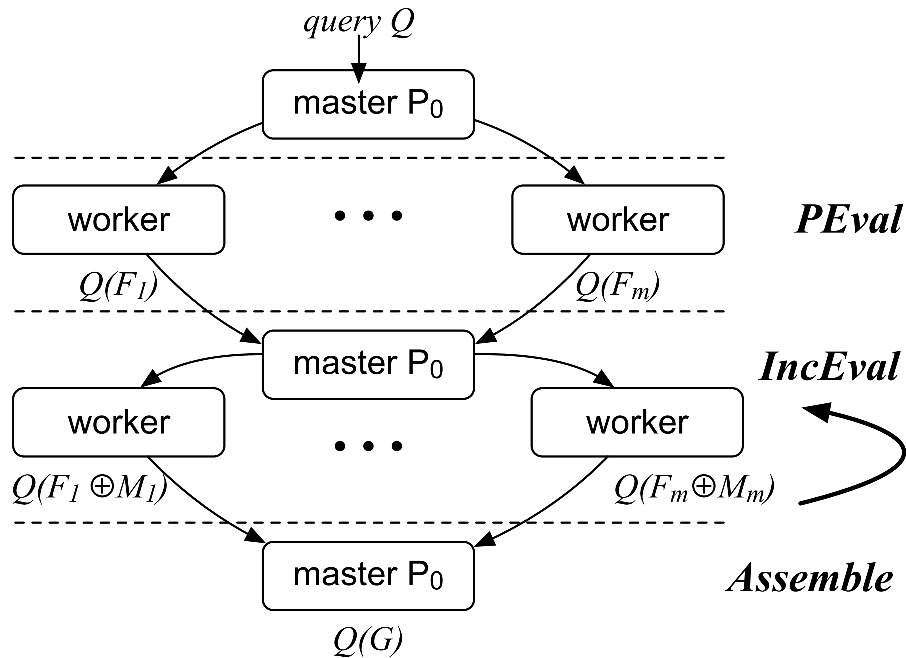


Figure 2.1: GRAPE workflow

Parallel model. Given $Q \in \mathcal{Q}$, GRAPE computes $Q(G)$ in the partitioned G as shown in Fig. 2.1. Upon receiving Q at coordinator P_0, GRAPE posts the same Q to all the workers. It adopts synchronous message passing following BSP [Val90]. Its parallel computation consists of three phases.

(1) Partial evaluation (**PEval**). In the first superstep, upon receiving Q, each worker P_i computes partial results $Q(F_i)$ locally at F_i using PEval, in parallel ($i \in [1,m]$). It also identifies and initializes a set of update parameters for each F_i that records the status of its border nodes. At the end of the process, it generates a message from the update parameters at each P_i and sends it to coordinator P_0 (see Section 2.2.2).

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Each superstep has two steps, one at P_0 and the other at the workers.

(2.a) Coordinator. Coordinator P_0 checks whether for all $i \in [1,m]$, P_i is inactive, *i.e.,* P_i is done with its local computation and there is no pending message designated for P_i. If so, GRAPE invokes Assemble and terminates (see below). Otherwise, P_0 routes messages from the last superstep to workers (Section 2.2.2), and triggers the next superstep.

(2.b) Workers. Upon receiving message M_i, worker P_i *incrementally* computes $Q(F_i \oplus M_i)$ with IncEval, by treating M_i as updates, in parallel for all $i \in [1,m]$. It automatically finds the changes to the update parameters in each F_i, and sends the changes as a message to P_0 (see Section 2.2.3).

GRAPE supports data-partitioned parallelism by *partial evaluation* on local fragments, in parallel by all workers. Its *incremental step* (2.b) speeds up iterative graph computations by reusing the partial results from the last superstep.

(3) Termination (**Assemble**). The coordinator P_0 decides to terminate if there is no change to any update parameters (see (2.a) above). If so, P_0 pulls partial results from all workers, and computes $Q(G)$ by Assemble. It returns $Q(G)$.

We now introduce the programming model of GRAPE. For a class $\mathcal{Q}$ of graph queries, one only needs to provide three core functions PEval, IncEval and Assemble referred to as *a* PIE *program.* These are conventional sequential algorithms, and can be picked from Library API of GRAPE. We next elaborate a PIE program.

2.2.2 PEval: Partial Evaluation

PEval takes a query $Q \in \mathcal{Q}$ and a fragment F_i of G as input, and computes partial answers $Q(F_i)$ at worker P_i in parallel for all $i \in [1,m]$. It may be any existing sequential algorithm $\mathcal{T}$ for $\mathcal{Q}$, extended with the following:

- *partial result* kept in a designated variable; and
- *message specification* as its interface to IncEval.

Communication between workers is conducted via messages, defined in terms of *update parameters* as follows.

(1) Message preamble. PEval (a) declares *status variables* $\vec{x}$, and (b) specifies a set C_i of nodes and edges relative to $F_i.I$ or $F_i.O$. The status variables associated with C_i are denoted by $C_i.\bar{x}$, referred to as the *update parameters* of F_i.

Intuitively, variables in $C_i.\bar{x}$ are the candidates to be updated by incremental steps. In other words, messages M_i to worker P_i are *updates* to the values of variables in $C_i.\bar{x}$.

More specifically, C_i is specified by an integer d and S, where S is either $F_i.I$ or $F_i.O$. That is, C_i is the set of nodes and edges within d-hops of nodes in S.

If $d = 0$, C_i is $F_i.I$ or $F_i.O$. Otherwise, C_i may include nodes and edges from other fragments F_j of G (see an example in Section 2.4).

The variables are declared and initialized in PEval. At the end of PEval, it sends the values of $C_i.\bar{x}$ to coordinator P_0.

(2) Message segment. PEval may specify function *aggregateMsg*, to resolve conflicts when multiple messages from different workers attempt to assign different values to the same update parameter (variable). When such a strategy is not provided, GRAPE picks a default exception handler.

(3) Message grouping. GRAPE deduces updates to $C_i.\vec{x}$ for $i \in [1,m]$, and treats them as messages exchanged among workers. More specifically, at coordinator P_0, GRAPE identifies and maintains $C_i.\bar{x}$ for each worker P_i. Upon receiving messages from P_i's, GRAPE works as follows.

(a) Identifying C_i. It deduces C_i for $i \in [1,m]$ by referencing fragmentation graph G_{P}, and C_i remains unchanged in the entire process. It maintains update parameters $C_i.\bar{x}$ for F_i.

(b) Composing M_i. For messages from each P_i, GRAPE (i) identifies variables in $C_i.\bar{x}$ with *changed values*; (ii) deduces their designations P_j by referencing G_{P}; if P is edge-cut, the variable tagged with a node v in $F_i.O$ will be sent to worker P_j if v is in $F_j.I$ (*i.e.,* if $i \mapsto j$ is in $G_{\mathsf{P}}(v)$); similarly for v in $F_i.I$; if P is vertex-cut, it identifies nodes shared by F_i and F_j ($i \neq j$); and (iii) it combines all changed variables values designated to P_j into a single message M_j, and sends M_j to worker P_j in the next superstep for all $j \in [1,m]$.

If a variable x is assigned a set S of values from different workers, function *aggregateMsg* is applied to S to resolve the conflicts, and its result is taken as the value of

x.

These are automatically conducted by GRAPE, which minimizes communication costs by passing only *updated* variable values. To reduce the workload at the coordinator, alternatively each worker may maintain a copy of G_P and deduce the designation of its messages in parallel.

Example 1: We show how GRAPE parallelizes SSSP. Consider a directed graph $G = (V, E, L)$ in which for each edge e, $L(e)$ is a positive number. The length of a path $(v_0, \ldots, v_k)$ in G is the sum of $L(v_{i-1}, v_i)$ for $i \in [1, k]$. For a pair (s, v) of nodes, denote by $\mathsf{dist}(s, v)$ the *shortest distance* from s to v, *i.e.,* the length of a shortest path from s to v. Given graph G and a node s in V, GRAPE computes $\mathsf{dist}(s, v)$ for all $v \in V$. It adopts edge-cut partition [BLV14]. It deduces $F_i.O$ by referencing G_P and stores $F_i.O$ at each fragment F_i.

As shown in Fig. 2.2, PEval (lines 1-14) is *verbally identical* to Dijsktra's sequential algorithm [FT87]. The *only changes* are message preamble and segment (underlined). It declares an integer variable $\mathsf{dist}(s, v)$ for each node v, initially ∞ (except $\mathsf{dist}(s, s)$ = 0). It specifies min as *aggregateMsg* to resolve conflicts: if there are multiple values for the same $\mathsf{dist}(s, v)$, the smallest value is taken by the linear order on integers. The update parameters are $C_i.\bar{x} = \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$.

At the end of its process, PEval sends $C_i.\bar{x}$ to coordinator P_0. At P_0, GRAPE maintains $\mathsf{dist}(s, v)$ for all $v \in \mathcal{F}.O = \mathcal{F}.I$. Upon receiving messages from all workers, it takes the smallest value for each $\mathsf{dist}(s, v)$. It finds those variables with smaller values, deduces their destinations by referencing G_P, groups them into message M_j, and sends M_j to P_j. □

2.2.3 IncEval: Incremental Evaluation

Given query Q, fragment F_i, partial results $Q(F_i)$ and message M_i (updates to $C_i.\bar{x}$), IncEval computes $Q(F_i \oplus M_i)$ incrementally, making maximum reuse of the computation of $Q(F_i)$ in the last round. Each time after IncEval is executed, GRAPE treats $F_i \oplus M_i$ and $Q(F_i \oplus M_i)$ as F_i and $Q(F_i)$, respectively, for the next round of incremental computation.

IncEval can take any existing sequential incremental algorithm $\mathcal{T}_\Delta$ for Q. It shares the message preamble of PEval. At the end of the process, it identifies *changed values* to $C_i.\bar{x}$ at each F_i, and sends the changes as messages to P_0. At P_0, GRAPE composes

---

Input: $F_i(V_i, E_i, L_i)$, source vertex s
Output: $Q(F_i)$ consisting of current $\mathsf{dist}(s,v)$ for all $v \in V_i$

Message preamble: (designated) /*candidate set C_i is $F_i.O$*/
for each node $v \in V_i$, an integer variable $\mathsf{dist}(s,v)$

```
/*sequential algorithm for SSSP (pseudo-code)*/
1.  initialize priority queue Que;
2.  dist(s,s) := 0;
3.  for each v in V_i do
4.    if v! = s then
5.    dist(s,v) := ∞;
6.  Que.addOrAdjust(s, dist(s,s));
7.  while Que is not empty do
8.    u := Que.pop() // pop vertex with minimal distance
9.    for each child v of u do // only v that is still in Q
10.     alt := dist(s,u) + L_i(u,v);
11.     if alt < dist(s,v) then
12.       dist(s,v) := alt;
13.       Que.addOrAdjust(v, dist(s,v));
14. Q(F_i) := {dist(s,v) | v ∈ V_i}
```

Message segment: $M_i := \{\mathsf{dist}(s,v) \mid v \in F_i.O\}$;
aggregateMsg = $\min(\mathsf{dist}(s,v))$;

Figure 2.2: Algorithm PEval for SSSP

messages as described in 3(b) above.

Boundedness. Graph computations are typically iterative. GRAPE reduces the costs of iterative computations by promoting *bounded incremental algorithms* for IncEval.

Consider an incremental algorithm $\mathcal{T}_\Delta$ for $\mathcal{Q}$. Given G, $Q \in \mathcal{Q}$, $Q(G)$ and updates M to G, it computes ΔO such that $Q(G \oplus M) = Q(G) \oplus \Delta O$, where ΔO denotes changes to the old output $O(G)$. It is said to be *bounded* if its cost can be expressed as a function in the size of $|\mathsf{CHANGED}| = |\Delta M| + |\Delta O|$, *i.e.,* the size of changes in the input and output [RR96b, FWW13].

Intuitively, $|\mathsf{CHANGED}|$ represents the updating costs inherent to the incremental

problem for Q itself. For a bounded IncEval, its cost is determined by $|\mathsf{CHANGED}|$, not by the size $|F_i|$ of entire F_i, no matter how big $|F_i|$ is.

Input: $F_i(V_i, E_i, L_i)$, partial result $Q(F_i)$, message M_i
Output: $Q(F_i \oplus M_i)$

1. initialize priority queue Que;
2. **for each** $\mathsf{dist}(s, v)$ in M **do**
3. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
4. **while** Que is not empty **do**
5. u := Que.pop() /* pop vertex with minimum distance*/
6. **for each** children v of u **do**
7. alt := $\mathsf{dist}(s, u)$ + $L_i(u, v)$;
8. **if** $alt < \mathsf{dist}(s, v)$ **then**
9. $\mathsf{dist}(s, v) := alt$;
10. Que.addOrAdjust(v, $\mathsf{dist}(s, v)$);
11. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

Message segment: $M_i = \{\mathsf{dist}(s, v) \mid v \in F_i.O,\ \mathsf{dist}(s, v)$ decreased$\}$;

Figure 2.3: Algorithm IncEval for SSSP

Example 2: Continuing with Example 1, we give IncEval in Fig. 2.3. It is the sequential incremental algorithm for SSSP in [RR96b], in response to changed $\mathsf{dist}(s, v)$ for v in $F_i.I$ (here M_i includes changes to $\mathsf{dist}(s, v)$ for $v \in F_i.I$ deduced from G_P).

Using a queue Que, it starts with M_i, propagates the changes to affected area, and updates the distances (see [RR96b]). The partial result is now the revised distances (line 11).

At the end of the process, IncEval sends to coordinator P_0 updated values of those status variables in $C_i.\bar{x}$, as in PEval. It applies *aggregateMsg* min to resolve conflicts.

The only changes to the algorithm of [RR96b] are underlined in Fig. 2.3. Following [RR96b], one can show that IncEval is *bounded*: its cost is determined by the sizes of "updates" $|M_i|$ and the changes to the output. This reduces the cost of iterative computation of SSSP (the **while** and **for** loops). □

2.2.4 Assemble Partial Results

Function Assemble takes partial results $Q(F_i \oplus M_i)$ and fragmentation graph G_P as input, and combines $Q(F_i \oplus M_i)$ to get $Q(G)$. It is triggered when no more changes can be made to update parameters $C_i.\bar{x}$ for any $i \in [1,m]$.

Example 3: Continuing with Example 2, Assemble (not shown) for SSSP takes $Q(G)$ = $\bigcup_{i \in [1,n]} Q(F_i)$, the union of the shortest distance for each node in each F_i.

The GRAPE process terminates with correct $Q(G)$. The updates to $C_i.\bar{x}$ are "monotonic": the value of dist(s,v) for each node v decreases or remains unchanged. There are finitely many such variables. Furthermore, dist(s,v) is the shortest distance from s to v, as warranted by the correctness of the sequential algorithms [FT87, RR96b] (PEval and IncEval). □

Putting these together, one can see that a PIE program parallelizes a graph query class $\mathcal{Q}$ provided with a sequential algorithm $\mathcal{T}$ (PEval) and a sequential incremental algorithm $\mathcal{T}_\Delta$ (IncEval) for $\mathcal{Q}$. Assemble is typically a straightforward sequential algorithm. A large number of sequential (incremental) algorithms are already in place for various $\mathcal{Q}$. Moreover, there have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [Har04, FHT17]. Thus GRAPE makes parallel graph computations accessible to a large group of end users.

In contrast to existing graph systems, GRAPE plugs in $\mathcal{T}$ and ΔT as a whole, and confines communication specification to the message segment of PEval. Users do not have to think "like a vertex" [MAB$^+$10, GLG$^+$12, TBC$^+$13, YCLN14] when programming. As opposed to vertex-centric and block-centric systems, GRAPE runs sequential algorithms on entire fragments. Moreover, IncEval employs incremental evaluation to reduce cost, which is a unique feature of GRAPE. Note that IncEval speeds up iterative computations by minimizing unnecessary recomputation of $Q(F_i)$, *no matter whether it is bounded or not.*

2.2.5 GRAPE API

GRAPE provides a declarative programming interface for users to plug in the sequential algorithms as UDFs (user-defined functions). Upon receiving (sequential) algorithms, GRAPE registers them as stored procedures in its API library, and maps them to a query class $\mathcal{Q}$.

In addition, GRAPE can simulate MapReduce. More specifically, GRAPE supports

two types of messages:

- *designated* messages from one worker to another; and
- *key-value* pairs (*key*, *val*), to simulate MapReduce.

The messages generated by PEval and IncEval are marked *key-value* or *designated*. The messages we have seen so far are designated, and GRAPE automatically identifies their destinations at coordinator P_0, as described in Section 2.2.2.

If the messages are marked *key-value*, GRAPE automatically recognizes the key and value segments by parsing the message declaration in PEval and IncEval. Following MapReduce, it groups the messages by keys at coordinator P_0, and distributes them across m workers, to balance the workload.

2.3 Foundation of GRAPE

Below we present the correctness guarantees of the parallel model of GRAPE, and demonstrate the power of GRAPE.

2.3.1 Correctness of Parallel Model

Intuitively, GRAPE supports a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ over m fragments defined as:

$$\begin{aligned} R_i^0 &= \mathsf{PEval}(Q, F_i^0[\bar{x}_i]), \\ R_i^{r+1} &= \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i), \end{aligned}$$

where $i \in [1,m]$, r indicates a superstep, R_i^r denotes partial results in step r at worker P_i, $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ is fragment F_i at the end of superstep r carrying update parameters $C_i.\bar{x}_i$, and M_i indicates changes to $C_i.\bar{x}_i$. The computation proceeds until it reaches r_0 when $R_i^{r_0} = R_i^{r_0+1}$. At this point, $\mathsf{Assemble}(G_{\mathsf{P}}, R_1^{r_0}, \ldots, R_m^{r_0})$ is computed and returned.

We next prove a correctness guarantee for the simple model with *designated messages*. We start with notations.

(1) We say that GRAPE with PEval, IncEval and P *terminates* if for all queries $Q \in \mathcal{Q}$ and all graphs G, there exists r_0 such that at superstep r_0, $R_i^{r_0} = R_i^{r_0+1}$ for all $i \in [1,m]$.

(2) Denote by $G[\bar{x}]$ a graph G with update parameters $\bar{x}$. We say that PEval is *correct for* $\mathcal{Q}$ if for all $Q \in \mathcal{Q}$ and graphs G, $\mathsf{PEval}(Q, G[\bar{x}])$ returns $Q(G[\bar{x}])$. Similarly, IncEval is *correct for* $\mathcal{Q}$ if $\mathsf{IncEval}(Q, G[\bar{x}], M, Q(G[\bar{x}]))$ returns $Q(G[\bar{x} \oplus M])$, where $\bar{x} \oplus M$ denotes $\bar{x}$ updated by M.

We say that Assemble is *correct for* $\mathcal{Q}$ *w.r.t.* P if when GRAPE with PEval, IncEval and P terminates at superstep r_0, $\mathsf{Assemble}(Q(F_1[\bar{x}_1^{r_0}]), \ldots, Q(F_m[\bar{x}_m^{r_0}])) = Q(G)$, where $\bar{x}_i^{r_0}$ denotes the values of parameters $C_i.\bar{x}_i$ at round r_0.

(3) We say that PEval and IncEval satisfy the *monotonic condition w.r.t.* P if for all variables $x \in C_i.\bar{x}$, $i \in [1,m]$ (a) the values of x are computed from values in the active domain of G, and (b) there exists a partial order p_x on the values of x such that IncEval updates x in the order of p_x.

Intuitively, condition (a) says that x draws values from a finite domain, and condition (b) says that x is updated "monotonically" following p_x. These ensure that GRAPE parallelization with PEval, IncEval and P terminate.

For instance, $\mathsf{dist}(s,v)$ in Example 1 can only be changed in the decreasing order (*i.e.,* min for *aggregateMsg*).

Theorem 1 [Assurance Theorem]: *Consider sequential algorithms* PEval, IncEval, Assemble *for a graph query class* $\mathcal{Q}$, *and a partition strategy* P. *If (a)* PEval *and* IncEval *satisfy the monotonic condition* w.r.t. P, *and (b)* PEval, IncEval *and* Assemble *are correct for* $\mathcal{Q}$ w.r.t. P, *then* GRAPE *with* PEval, IncEval, Assemble *and* P *guarantees to terminate and correctly compute* $Q(G)$ *for all* $Q \in \mathcal{Q}$ *and graphs* G. □

Proof: By the correctness of Assemble, PEval and IncEval, we only need to show the following: for any query Q and graph G,

(1) there exists a natural number $r_{(Q,G)}$ for Q and G such that GRAPE terminates at superstep $r_{(Q,G)}$, with deterministic values $\bar{x}_i^{r_{(Q,G)}}$ for all update parameters in all fragments F_i of G (for $i \in [1,m]$); and

(2) IncEval computes partial answers $Q(F_i[\bar{x}_i^{r_{(Q,G)}}])$ on all fragments $F_i (i \in [1,m])$ of G.

Intuitively, (1) ensures that given Q and G, GRAPE always terminates in the same state, and (2) guarantees that partial answers $Q(F_i[\bar{x}_i^{r_{(Q,G)}}])$ are correctly computed for all fragments F_i $(i \in [1,m])$ of G. If these hold, GRAPE is guaranteed to return $Q(G)$ by the correctness of Assemble.

(1) We first show that GRAPE terminates. Assume by contradiction that there exist Q and G such that GRAPE does not terminate. Consider the values of update parameters in the fragments of G during the run. Since at least one update parameter has to be updated in a superstep of incremental computation (except the last step), and the total number of distinct values to update parameters is bounded by Q and G by the monotonic condition (a) given. Hence there must exist supersteps p and q such that for each $i \in [1,m]$, $\bar{x}_i^p = \bar{x}_i^q$, *i.e.,* the values to all the parameters changed at supersteps p and q are the same. This contradicts the monotonic condition (b) that requires IncEval to update parameters following a partial order on their values. Thus for all Q and G, GRAPE must terminate.

To verify that the values to $C_i.\bar{x}$ when GRAPE terminates are deterministic for Q and G, we show the following: the values to $C_i.\bar{x}$ are updated deterministically at each superstep r in the run of GRAPE, by induction on r. (a) When $r = 1$, *i.e.,* in the first superstep by PEval, the parameters are initialized deterministically by the definition

of PEval. (b) Assume that when $r \leq k$, the parameters in fragments of G for Q are changed deterministically at step r. Consider step $r = k+1$. Since $\bar{x}_i^k$'s $(i \in [1,m])$ are deterministic, IncEval generates M_i to each F_i deterministically, *i.e.*, $\bar{x}_i^{k+1} = \bar{x}_i^k \oplus M_i$ are updated deterministically. That is, values to $\bar{x}_i^{k+1}$ are also deterministic, independent of the order of fragments on which IncEval terminates at each superstep. Therefore, GRAPE always terminates for Q and G with the same final state for the update parameters.

(2) We prove that for any Q and G, at any superstep r of the run of GRAPE for Q and G with PEval, IncEval and Assemble, partial answers $Q(F_i[\bar{x}_i^r])$ $(i \in [1,m])$ are computed on all fragments F_i of G. We show this by induction on r.

(a) When $r = 1$. By the correctness of PEval, partial answers $Q(F_i[\bar{x}_i^1])$ are computed by PEval on fragments F_i of G.

(b) Assume that when $r = k$, GRAPE computes partial answers $Q(F_i[\bar{x}_i^k])$ on fragments F_i of G. Consider $r = k+1$. By the correctness of IncEval, GRAPE also correctly computes $Q(F_i[\bar{x}_i^k \oplus M]) = Q(F_i[\bar{x}_i^{k+1}])$ on each fragment F_i of G. Therefore, GRAPE computes partial answers on fragments of G at each superstep in the run for Q and G. □

Remark. Since PEval and IncEval can be any graph algorithms and the halting problem for Turing machine is undecidable, the monotonic condition is one of the sufficient conditions for termination and correctness. As demonstrated by the variety of algorithms in the dissertation, most of common graph computations satisfy this condition. Moreover, by the formulation of the fixpoint computation, any existing conditions for contracting fixpoint computation apply for the termination of GRAPE parallelization. It should be remarked that no other systems provide similar sufficient conditions for termination and correctness, to the best of my knowledge.

When the monotonicity is not guaranteed, the programmers may have to warrant the correctness and termination themselves, like the state-of-the-art graph query engines.

When it is required to terminate within a fixed number c of rounds, such as PageRank and SGD, the results are often defined as the results of the computation within c bounds. We do not enforce such a default bound by GRAPE, and if users opt to do so, it is up to them to justify the semantics and correctness.

2.3.2 The Power of GRAPE

GRAPE can readily switch to other parallel models.

Following [Val91], we say that a parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there exists a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$. The cost includes computational and communication cost. For GRAPE, these are measured by the running time of PEval, IncEval and Assemble on all the processors, and by the total size of the messages passed among all the processors in the entire process.

We show that GRAPE optimally simulates popular parallel models MapReduce [DG08], BSP [Val90] and PRAM [Val91].

Theorem 2 [Simulation Theorem]: *(1) all BSP algorithms with n workers in t supersteps can be optimally simulated on* GRAPE *with n workers in t supersteps, without extra cost in each superstep;*

(2) all MapReduce programs using n processors can be optimally simulated by GRAPE *using n processors; and*

(3) all CREW PRAM algorithms using $O(P)$ total memory, $O(P)$ processors and t time can be run in GRAPE *in $O(t)$ supersteps using $O(P)$ processors with $O(P)$ memory.* □

As a consequence, all algorithms developed for graph systems based on MapReduce and/or BSP can be readily migrated to GRAPE without much extra cost, including Pregel [MAB+10], GraphX [GXD+14], Giraph++ [TBC+13] and Blogel [YCLN14].

We next give the proof outline.

Bulk-Synchronous parallel model. A BSP algorithm proceeds in supersteps. Each superstep consists of an input phase, a local computation phase and an output phase. The workers are synchronized between supersteps. The cost of a superstep is expressed as $w+gh+l$, where (a) w is the maximum number of operations by any worker within the superstep; (b) h is the maximum amount of messages sent/received by any workers; (c) g is the communication throughput ratio, or bandwidth inefficiency; and (d) l is the communication latency or synchronization periodicity. We define the throughput and latency for GRAPE similarly.

For Theorem 2(1), each worker of BSP is simulated by a worker in GRAPE. PEval is defined to perform the same local computation in the first superstep of BSP, IncEval

simulates the actions of each worker in the later supersteps of the BSP algorithm, and Assemble collects and combines the partial results. Message routing and synchronization control adopt the same strategy of BSP, via designated messages, where the coordinator acts as the synchronization router. One can verify that the simulation does not incur extra cost.

In particular, Pregel [MAB+10] assigns a virtual worker to each node (single-vertex fragments). GRAPE reduces its excessive messages, supports graph-level optimization, and employs incremental steps to speed up iterative computation.

MapReduce. A MapReduce program is specified by a Map function and a Reduce function [DG08]. Its computation is a sequence of map, shuffle and reduce steps that operate on a set of key-value pairs. Its cost is measured in terms of (a) N: the *number of rounds* of map-shuffle-reduce conducted in the process, (b) S_i: the *communication cost* of round i, as the sum of the sizes of input and output for all reducers, and (c) H_i: the *computational cost* of round i, as the sum of the time taken by each mapper and reducer in round i.

For Theorem 2(2), GRAPE uses PEval to perform the map phase of the first map-shuffle-reduce round, and two supersteps (IncEval) to simulate each later round, one for map and the other for reduce, via key-value messages (see Section 2.2.5). We provide a compilation function that given Map and Reduce functions, constructs (a) PEval as the Map function, (b) IncEval by invoking Map for odd supersteps and Reduce for even supersteps, and (c) Assemble by simply taking a union of partial results. By induction on the round N of MapReduce, one can verify that the transformed GRAPE process has running time $O(C)$, where C is the parallel running time of the MapReduce computation.

There exist more efficient compilation algorithms by combining multiple MapReduce tasks into a single GRAPE superstep. Moreover, GRAPE employs (bounded) IncEval to reduce MapReduce cost for iterative graph computations.

Parallel Random Access Machine. PRAM consists of a number of processors sharing memory, and any processor can access any memory cell in unit time. The computation is synchronous. In one unit time, each processor can read one memory location, execute a single operation and write into one memory location. PRAM is further classified for access policies of shared memory, *e.g.,* CREW PRAM indicates concurrent read and exclusive write (see [Val91] for details).

It is known that a CREW PRAM algorithm using t time with $O(P)$ total mem-

ory and $O(P)$ processors can be simulated by a MapReduce algorithm in $O(t)$ rounds using at most $O(P)$ reducers and memory [KSV10]. By Theorem 2(2), each MapReduce algorithm in r rounds can be simulated by GRAPE in $2r$ supersteps. From these Theorem 2(3) follows.

We next give the detailed proof.

Proof: (1) Since BSP and GRAPE have the same amount of physical workers, each worker of BSP is simulated by a worker in GRAPE. Initially the graph is distributed in the same way as that in BSP algorithm $\mathcal{A}$. PEval is defined to do the same as the local computation during the first superstep of $\mathcal{A}$, and it generates messages that are identical to the ones in $\mathcal{A}$. From the second superstep, IncEval conducts the actions of each worker when executing BSP algorithm. Message routing and synchronization control adopt the same strategy as in $\mathcal{A}$. Obviously the computation on each worker in GRAPE is the same as its counterpart in BSP, and all messages sent or received by each pair of workers within each superstep are also identical, which lead to an optimal simulation.

(2) We use two supersteps in GRAPE to simulate one map-shuffle-reduce round of a MapReduce algorithm, including a map phase and a reduce phase, in the key-value message mode (see Section 2.2.5) for messages. More specifically, for a MapReduce algorithm $\mathcal{A}$ that has R rounds, we implement each round $r \in [1,R]$ of $\mathcal{A}$ in GRAPE as follows.

(a) Round $r = 1$: Initially input data is distributed among the worker by using the same strategy as in $\mathcal{A}$, such that each worker is assigned the same data as that of the mapper it simulates. We define PEval to be the same as the mapping function μ in round 1, *i.e.,* it performs the same computation as specified in μ and generates an intermediate multiset of key-value pairs. Moreover, the key-value pairs are treated as messages and sent to the coordinator P_0. Then P_0 groups all the messages ($\langle key; value \rangle$ pairs) with the same *key* and sends them to a worker that simulates the corresponding reducer dealing with *key* in $\mathcal{A}$. This process simulates one shuffle step of $\mathcal{A}$. After that, each worker that receives a message (list) $L_k = \langle k; v_{i_j} \ldots \rangle$ simulates a reducer, *i.e.,* we let function IncEval in this superstep do the same as the reducer function ρ in round 1. Note that IncEval uses the messages received only, ignoring the local data. The outputs of IncEval are also treated as messages and delivered to P_0. Upon receiving these, P_0 routes them based on the distribution of key-value pairs to mappers in the next round

of $\mathcal{A}$, so that each worker gets the same key-value pairs as that of the mapper it will simulate in the next round.

(b) Round $r > 1$: The simulations for latter rounds are similar to those of the first round, except that PEval is no longer used. More specifically, the action of mapping function μ in round r is simulated by IncEval instead of PEval as in case (a). Hence, function IncEval is carefully designed to model the computation of the functions μ and ρ in different rounds of $\mathcal{A}$. IncEval operates on newly received messages alone, to simulate MapReduce. When $\mathcal{A}$ terminates, P_0 stops routing the messages produced in the last superstep and returns result in the same way as $\mathcal{A}$, possibly using Assemble.

It is easy to verify that the computational and communication cost of the GRAPE algorithm is the same as $\mathcal{A}$. Indeed, every worker simulates a mapper/reducer and conducts the same computation, and all the messages generated are identical to the key-value pairs transmitted in the shuffle network of $\mathcal{A}$. Thus, this makes an optimal simulation.

(3) A proof has been given above.

Taken together, GRAPE can easily switch to different modes, and does not imply degradation of computational power. □

2.4 Graph Computations in GRAPE

We have seen how GRAPE parallelizes graph traversal SSSP (Section 2.2). We next show how GRAPE parallelizes existing sequential algorithms for a variety of graph computations. We take pattern matching, connectivity and collaborative filtering as examples (Sections 2.4.1–2.4.3, respectively).

2.4.1 Graph Pattern Matching

We start with graph pattern matching commonly used in, *e.g.,* social media marketing and knowledge base expansion.

A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$, in which (a) V_Q is a set of *query nodes*, (b) E_Q is a set of *query edges*, and (c) each node u in V_Q carries a label $L_Q(u)$.

We study two semantics of graph pattern matching.

Graph simulation. A graph G *matches* a pattern Q via *simulation* if there is a binary relation $R \subseteq V_Q \times V$ such that

(a) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u,v) \in R$, referred to as a *match* of u; and

(b) for each pair $(u,v) \in R$, (a) $L_Q(u) = L(v)$, and (b) for each query edge (u,u') in E_q, there exists an edge (v,v') in graph G such that $(u',v') \in R$.

Graph pattern matching via graph simulation is as follows.

- Input: A directed graph G and a pattern Q.
- Output: The unique maximum relation $Q(G)$.

It is known that if G matches Q, then there exists a *unique maximum* relation [HHK95], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Moreover, $Q(G)$ can be computed in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [HHK95, FLM+10].

We show how GRAPE parallelizes graph simulation. Like SSSP, it adopts an edge-cut partition strategy.

(1) PEval. GRAPE takes the sequential simulation algorithm of [HHK95] as PEval to compute $Q(F_i)$ in parallel. Its message preamble declares a Boolean status variable $x_{(u,v)}$ for each query node u in V_Q and each node v in F_i, indicating whether v matches u, initialized true. It takes $F_i.I$ as candidate set C_i. For each node $u \in V_Q$, PEval computes a set $\mathsf{sim}(u)$ of candidate matches v in F_i, and iteratively removes from $\mathsf{sim}(u)$ those

nodes that violate the simulation condition (see [HHK95] for details). At the end of the process, PEval sends $C_i.\bar{x} = \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I\}$ to coordinator P_0.

At coordinator P_0, GRAPE maintains $x_{(u,v)}$ for all $v \in \mathcal{F}.I$. Upon receiving messages from all workers, it changes $x_{(u,v)}$ to false if it is false in *one of* the messages. This is specified by min as *aggregateMsg*, taking the order false prec true. GRAPE identifies those variables that become false, deduces their destinations by referencing G_P and $\mathcal{F}.I = \mathcal{F}.O$, groups them into messages M_j, and sends M_j to P_j.

(2) IncEval is the sequential incremental graph simulation algorithm of [FWW13] in response to edge deletions. If $x_{(u,v)}$ is changed to false by message M_i, it is treated as deletion of "cross edges" to $v \in F_i.O$. It starts with changed status variables in M_i, propagates the changes to affected area, and removes from sim matches that become invalid (see [FWW13] for details). The partial result is now the revised sim relation. At the end of the process, IncEval sends to coordinator P_0 updated values of those status variables in $C_i.\bar{x}$, as in PEval.

IncEval is *semi-bounded* [FWW13]: its cost is decided by the sizes of "updates" $|M_i|$ and changes to the affected area necessarily checked by all incremental algorithms for Sim, not by $|F_i|$.

(3) Assemble simply takes $Q(G) = \bigcup_{i\in[1,n]} Q(F_i)$, the union of all partial matches (sim at each F_i).

(4) The correctness is warranted by Theorem 1: the sequential algorithms [HHK95, FWW13] (PEval and IncEval) are correct, and the "monotonic" updates to $C_i.\bar{x}$: $x_{(u,v)}$ is initially true for each border node v, and is changed at most once to false.

Subgraph isomorphism. We next parallelize subgraph isomorphism, under which a *match* of pattern Q in graph G is a subgraph of G that is isomorphic to Q. Graph pattern matching via subgraph isomorphism is to compute the set $Q(G)$ of all matches of Q in G. It is intractable: it is NP-complete to decide whether $Q(G)$ is nonempty.

GRAPE parallelizes VF2, the sequential algorithm of [CFSV04] for subgraph isomorphism. It adopts a default edge-cut graph partition strategy P. It has *two* supersteps, one for PEval and the other for IncEval, outlined as follows.

(1) PEval identifies update parameter $C_i.\bar{x}$. It declares a status variable x_{id} with each node and edge, to store its id. It specifies the d_Q-neighbor $N_{d_Q}(v)$ of each node $v \in F_i.I$, where d_Q is the diameter of pattern Q, *i.e.,* the length of the shortest path between any two nodes in Q, and $N_d(v)$ is the subgraph of G induced by the nodes within d hops of v.

At P_0, $C_i.\bar{x}$ is identified for each fragment F_i (this can be done in parallel by workers as remarked in Section 2.2.2). Message M_i is composed and sent to P_i, including all nodes and edges in $C_i.\bar{x}$ that are from fragments F_j with $j \neq i$. The values of variables in $C_i.\bar{x}$ (the ids) will not be changed, and thus no partial order is defined on their values.

(2) IncEval is VF2. It computes $Q(F_i \oplus M_i)$ at each worker P_i in parallel, on fragment F_i extended with d_Q-neighbor of each node in $F_i.I$. IncEval sends no messages since the values of variables in $C_i.\bar{x}$ remain unchanged. As a result, IncEval is executed once, and hence two supersteps suffice.

(3) Assemble simply takes the union of all partial matches computed by IncEval from all workers.

(4) The correctness of the process is assured by VF2 and the locality of subgraph isomorphism: a pair (v, v') of nodes in G is in a match of Q only if v is in the d_Q-neighbor of v'.

2.4.2 Graph Connectivity

We next study graph connectivity. We parallelize sequential algorithms for computing connected components (CC).

Consider an undirected graph G. A subgraph G_s of G is a *connected component* of G if (a) it is connected, *i.e.,* for any pair (v, v') of nodes in G_s, there exists a path between v to v', and (b) it is maximum, *i.e.,* adding any node to G_s makes the induced subgraph no longer connected.

- Input: An undirected graph $G = (V, E, L)$.
- Output: All connected components of G.

It is known that CC is in $O(|G|)$ time [BJG08].

GRAPE partitions G by edge-cut. It picks a sequential CC algorithm as PEval. At each fragment F_i, PEval computes its local connected components and creates their ids. The component ids of the border nodes are exchanged with neighboring fragments. The (changed) ids are then used to incrementally update local components in each fragment by IncEval, which simulates a "merging" of two components whenever possible, until no more changes can be made.

(1) PEval declares an integer status variable v.cid for each node v in fragment F_i, initialized as its node id.

PEval uses a standard sequential traversal (*e.g.,* DFS) to compute the local connected components of F_i and determines v.cid for each $v \in F_i$. For each local component C, (a) PEval creates a "root" node v_c carrying the minimum node id in C as v_c.cid, and (b) links all the nodes in C to v_c, and sets their cid as v_c.cid. These can be completed in one pass of the edges of F_i via DFS. At the end of process, PEval sends $\{v.\mathsf{cid} \mid v \in F_i.I\}$ to coordinator P_0.

At P_0, GRAPE maintains v.cid for each all $v \in \mathcal{F}.I$. It updates v.cid by taking the smallest cid if multiple cids are received, by taking min as *aggregateMsg* in the message segment of PEval. It groups the nodes with updated cids into messages M_j, and sends M_j to P_j by referencing $G_{\mathcal{P}}$.

(2) IncEval incrementally updates the cids of the nodes in F_i upon receiving M_i. The message M_i sent to P_i consists of v.cid with updated (smaller) values. For each v in M_i, IncEval (a) finds the root v_c of v, and (b) for v_c and all the nodes linked to it, directly changes their cids to v.cid.

The incremental computation of IncEval is *bounded*: it takes $O(|M_i|)$ time to identify the root nodes, and $O(|\mathsf{AFF}|)$ time to update cids by following the direct link from the root nodes, where AFF consists of only those nodes with their cid *changed*. Hence, it avoids redundant local traversal, and makes the complexity of IncEval independent of $|F_i|$.

(3) Assemble merges all the nodes having the same cids in a bucket as a single connected component, and returns all the connected components as a set of buckets.

(4) Correctness. The process terminates as the cids of the nodes are monotonically decreasing, until no changes can be made. Moreover, it correctly merges two local connected components by propagating the smaller component id.

2.4.3 Collaborative Filtering

As an example of machine learning, we consider collaborative filtering (CF) [KBV+09], a method commonly used for inferring user-product rates in social recommendation. It takes as input a bipartite graph G that includes users U and products P, and a set of weighted edges $E \subseteq U \times P$. (1) Each user $u \in U$ (resp. product $p \in P$) carries (unknown) latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in E carries a weight $r(e)$, estimated as $u.f^T * p.f$ (possibly $\emptyset$ *i.e.,* "unknown") that encodes a rating from user u to product p. The *training set* E_T refers to edge set $\{e \mid r(e) \neq \emptyset, e \in E\}$, *i.e.,* all the known ratings. The CF problem is as follows.

- Input: Directed bipartite graph G, training set E_T.
- Output: The missing factor vectors $u.f$ and $p.f$ that minimizes an error function $\varepsilon(f, E_T)$, estimated as $\min \sum_{((u,p) \in E_T)} (r(u,p) - u.f^T p.f) + \lambda(\|u.f\|^2 + \|p.f\|^2)$.

That is, CF predicts all the unknown ratings by learning the factor vectors that "best fit" E_T. A common practice to approach CF is to use stochastic gradient descent (SGD) algorithm [KBV+09], which iteratively (1) predicts error $\varepsilon(u,p) = r(u,p) - u.f^T * p.f$, for each $e = (u,p) \in E_T$, and (2) updates $u.f$ and $p.f$ accordingly towards minimizing $\varepsilon(f, E_T)$.

GRAPE parallelizes CF by edge-cut partitioning E_T (as a bipartite graph). It adopts SGD [KBV+09] as PEval and an incremental algorithm ISGD [VJG14] as IncEval, using coordinator P_0 to synchronize the shared factor vectors $u.f$ and $p.f$.

(1) PEval. It declares a status variable $v.x = (v.f, t)$ for each node v, where $v.f$ is the factor vector of v (initially $\emptyset$), and t bookkeeps a timestamp at which $v.f$ is lastly updated. The candidate set is $C_i = F_i.O$. PEval is essentially the sequential SGD algorithm of [KBV+09]. It processes a "mini-batch" of training examples independently of others, to compute the prediction error $\varepsilon(u,p)$, and update local factor vectors f in the opposite direction of the gradient as:

$$u.f^t = u.f^{t-1} + \gamma(\varepsilon(u,p) * v.f^{t-1} - \lambda * u.f^{t-1}); \quad (2.1)$$

$$p.f^t = p.f^{t-1} + \gamma(\varepsilon(u,p) * u.f^{t-1} - \lambda * p.f^{t-1}). \quad (2.2)$$

At the end of its process, PEval sends messages M_i that contains updated $v.x$ for $v \in C_i$ to coordinator P_0.

At P_0, GRAPE maintains $v.x = (v.f, t)$ for all $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated values $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* it takes max as *aggregateMsg* on timestamps. GRAPE then groups the updated vectors into messages M_j, and sends M_j to P_j as usual.

(2) IncEval is algorithm ISGD of [VJG14]. Upon receiving M_i at worker P_i, it computes $F_i \oplus M_i$ by treating M_j as updates to factor vectors of nodes in $F_i.I$, and only modifies affected factor vectors as in PEval based solely on the new observations. It sends the updated vectors in C_i as in PEval.

(3) Assemble simply takes the union of all the factor vectors of nodes from the workers (to be used for recommendation).

(4) Correctness. The convergence condition in a sequential SGD algorithm [KBV+09, VJG14] is specified either as a predetermined maximum number of supersteps (as

in GraphLab), or when $\varepsilon(f, E_T)$ is smaller than a threshold. In either case, GRAPE correctly infers CF models guaranteed by the correctness of SGD and ISGD, and by monotonic updates with the latest changes as in sequential SGD algorithms.

2.5 Implementation of GRAPE

We next outline an implementation of GRAPE.

Architecture overview. GRAPE adopts a four-tier architecture depicted in Fig. 2.5, described as follows.

(1) Its top layer is a user interface. As shown in Fig. 2.4, GRAPE supports interactions with (a) developers who specify and register sequential PEval, IncEval and Assemble as a PIE program for a class $\mathcal{Q}$ of graph queries (the plug panel); and (b) end users who plug-in PIE programs from API library, pick a graph G, enter queries $Q \in \mathcal{Q}$, and "play" (the play panel). GRAPE parallelizes the PIE program, computes $Q(G)$ and displays $Q(G)$ in result and analytics consoles.

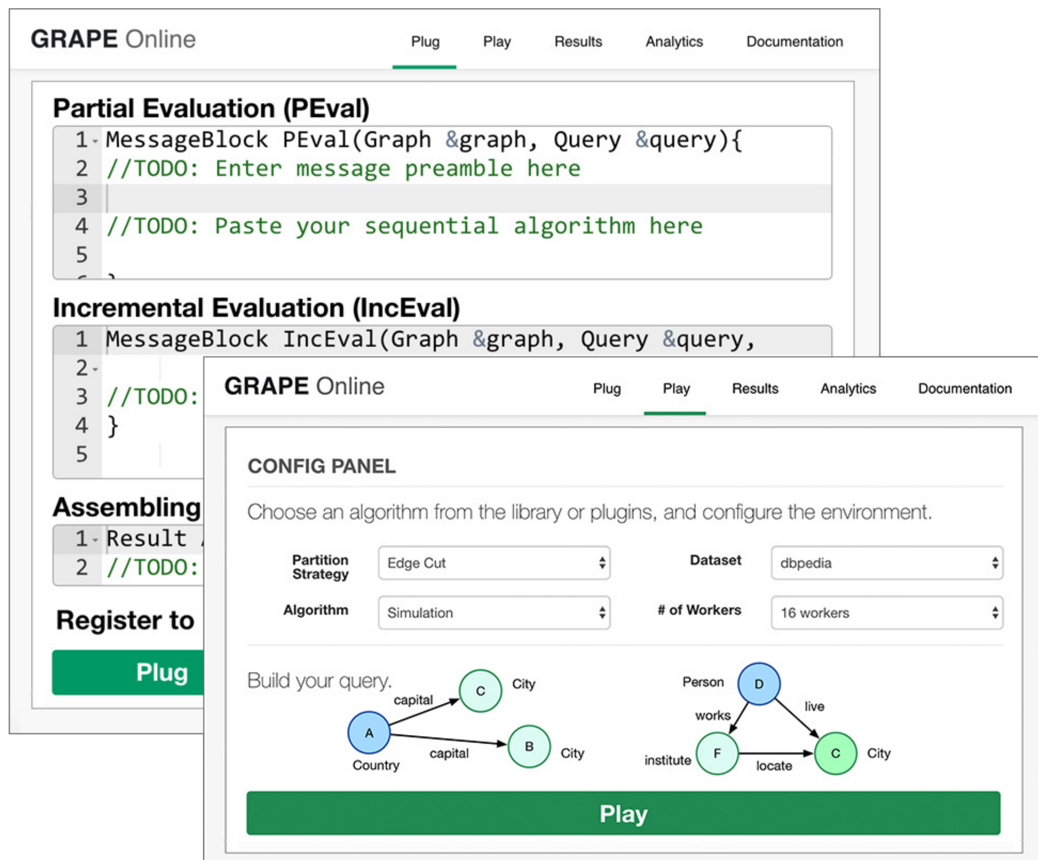


Figure 2.4: GRAPE interface

(2) At the core of the system is a parallel query engine. It manages sequential algorithms registered in GRAPE API, makes parallel evaluation plans for PIE programs, and executes the plans for query answering (see Section 2.2.1). It also enforces con-

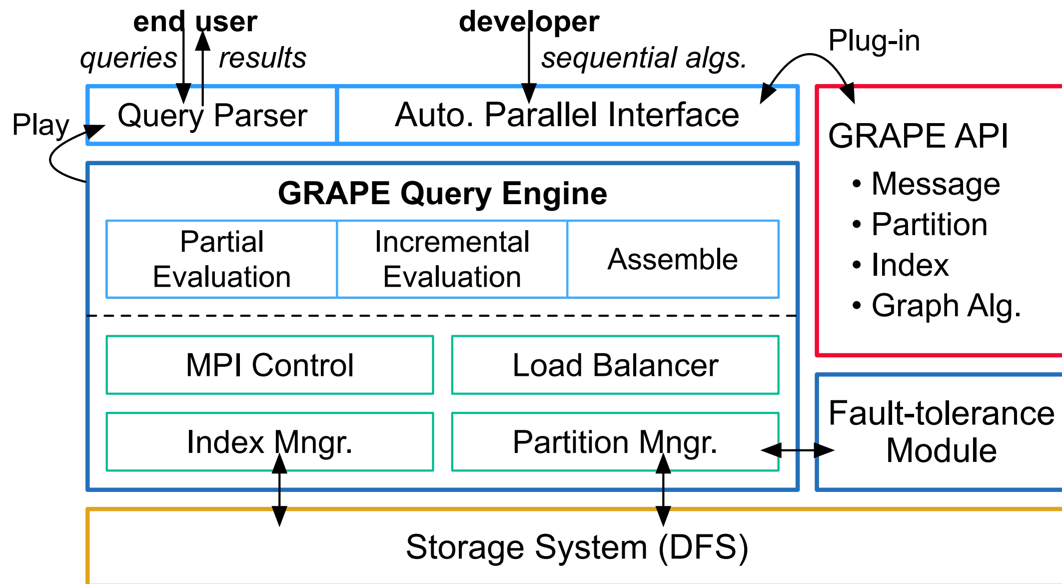


Figure 2.5: GRAPE architecture

sistency control and fault tolerance (see below).

(3) Underlying the query engine are (a) an *MPI Controller* (message passing interface) for communications between coordinator and workers, (b) an *Index Manager* for loading indices, (c) a *Partition Manager* to partition graphs, and (d) a *Load Balancer* to balance workload (see below).

(4) The storage layer manages graph data in DFS (distributed file system). It is accessible to the query engine, Index Manager, Partition Manager and Load Balancer.

Message passing. The MPI Controller of GRAPE makes use of a standard MPI for parallel and distributed programs. It currently adopts MPICH [GLDS96], which is also the basis of other systems such as GraphLab [LBG+12] and Blogel [YCLN14]. It generates messages and coordinates messages in synchronization steps using standard MPI primitives. It supports both designated messages and key-value pairs (see Section 2.2).

Graph partition. The Graph Partitioner supports a variety of built-in partition algorithms. Users may pick (a) METIS, a fast heuristic algorithm for sparse graphs [KK95], (b) vertex cut and edge cut partitions [GLG+12] for graphs with small vertex cut-set and edge cut-set, respectively, (c) 1-D and 2-D partitions [BDR13], which distribute vertex and adjacent matrix to the workers, respectively, emphasizing on maximizing the parallelism of graph traversal, and (d) a fast streaming-style partition strat-

egy [SK12] that assigns edges to high degree nodes to reduce cross edges. New data partition strategies can also be plugged into GRAPE.

Graph-level optimization. In contrast to prior graph systems, GRAPE supports data-partitioned parallelism by parallelizing the runs of sequential algorithms. Hence all optimization strategies developed for sequential (batch and incremental) algorithms can be readily plugged into GRAPE, to speed up PEval and IncEval over graph fragments. As examples, below we outline some optimization strategies.

(1) Indexing. Any indexing structure effective for sequential algorithm can be computed offline and directly used to optimize PEval, IncEval and Assemble, without recasting. GRAPE supports indices including (1) 2-hop index [CHKZ03] for reachability queries; and (2) neighborhood-index [KWAY13] for candidate filtering in graph pattern matching. Moreover, new indices can be "plugged" into GRAPE API library.

(2) Compression. GRAPE adopts *query preserving compression* [FLWW12] at the fragment level. Given a query class $\mathcal{Q}$ and a fragment F_i, each worker P_i computes a smaller F_i^c offline via a compression algorithm, such that for any query Q in $\mathcal{Q}$, $Q(F_i)$ can be computed from F_i^c without decompressing F_i^c, *regardless* of what sequential PEval and IncEval are used. As shown in [FLWW12], this compression scheme is effective for graph pattern matching and graph traversal, among others.

(3) Dynamic grouping. GRAPE dynamically group a set of border nodes by adding a "dummy" node, and sends messages from the dummy nodes in batches, instead of one by one. This effectively reduces the amount of message communication in each synchronization step.

(4) Load balancing. GRAPE groups computation tasks into work units, and estimates the cost at each virtual worker P_i in terms of the fragment size $|F_i|$ at P_i, the number of border nodes in F_i, and the complexity of computation $\mathcal{Q}$. Its Load Balancer computes an assignment of the work units to physical workers, to minimize both computational cost and communication cost (recall from Section 2.2 that GRAPE employs m virtual workers and n physical workers, and $m > n$). The bi-criteria objective makes it easy to deal with skewed graphs, when a small fraction of nodes are adjacent to a large fraction of the edges in G, as found in social graphs.

To the best of our knowledge, these optimization strategies are not supported by the state-of-the-art vertex-centric and block-centric systems. Indexing and query-preserving compression for sequential algorithms do not carry over to vertex programs, and block-centric programming essentially treats blocks as vertices rather than graphs. Moreover,

dynamic grouping does not help vertex-level synchronization.

Fault tolerance. GRAPE employs an arbitrator mechanism to recover from both worker failures and coordinator failures (*a.k.a.* single-point failures). It reserves a worker P_a as arbitrator, and a worker S'_c as a standby coordinator. It keeps sending heart-beat signals to all workers and the coordinator. In case of failure, (a) if a worker fails to respond, the arbitrator transfers its computation tasks to another worker; and (b) if the coordinator fails, it activates the standby coordinator S'_c to continue computation.

Consistency. Multiple workers may update copies of the same status variable. To cope with this, (a) GRAPE allows users to specify a conflict resolution policy as function *aggregateMsg* in PEval (Section 2.2.2), *e.g.,* min for SSSP and CC (Section 2.4), based on a partial order on the domain of status variables, *e.g.,* linear order on integers. Based on the policy, inconsistencies are resolved in each synchronization step of PEval and IncEval processes. Moreover, Theorem 1 guarantees the consistency when the policy satisfies the monotonic condition. (b) GRAPE also supports default exception handlers when users opt not to specify *aggregateMsg*. In addition, GRAPE allows users to specify generic consistency control strategies and register them in GRAPE API library.

We are also implementing a lightweight transaction controller, to support not only queries but also updates such as insertions and deletions of nodes and edges. When the load is light, it adopts non-destructive updates of functional databases [Tri89]. Otherwise, it switches to multi-version concurrency control [BG81] that keeps track of timestamps and versions, as also adopted by existing distributed systems.

2.6 Experimental Study

We next empirically evaluate the performance of GRAPE, for its (1) efficiency and communication cost using real-life graphs, (2) scalability with larger synthetic graphs, (3) effectiveness of incremental steps, (4) compatibility with optimization techniques developed for sequential graph algorithms, and (5) ease of programming. To focus on the main idea, we compared GRAPE with prior graph systems by plugging existing sequential algorithms into a preliminary implementation of GRAPE, without optimization.

Experimental setting. We start with graphs and queries.

Datasets. We used three real-life graphs of different types, including (1) liveJournal [liv], a social network with 4.8 million entities and 68 million relationships, with 100 labels and 18293 connected components; (2) DBpedia [dbp], a knowledge base with 28 million entities of 200 types and 33.4 million edges of 160 types; and (3) traffic [tra], a US road network with 23 million nodes (locations) and 58 million edges.

To evaluate collaborative filtering (CF), we used another real-life dataset movieLens [mov], which has 10 million movie ratings (as weighted edges) between a set of 71567 users and 10681 movies; these make a bipartite graph G for CF.

Queries. We randomly generated the following queries. (a) We sampled 10 source nodes in each graph, and constructed an SSSP query for each node. (b) We generated 20 pattern queries for Sim and SubIso, controlled by $|Q| = (|V_Q|, |E_Q|)$, the number of nodes and edges, respectively, using labels drawn from the graphs (see Section 2.4).

Algorithms. We implemented the core functions PEval, IncEval and Assemble given in Sections 2.2 and 2.4 for these query classes, registered in the API library of GRAPE. We used METIS [KK95] as the default graph partition strategy. We adopted basic sequential algorithms, and only used optimized Sim to demonstrate how GRAPE inherits optimization strategies developed for sequential algorithms (Exp-3).

We also implemented algorithms for the queries for Giraph, GraphLab and Blogel. We used "default" code provided by the systems when available, and made our best efforts to develop "optimal" algorithms otherwise; the code is available at [GRA] for interested reader. As GraphLab supports both synchronized and asynchronized models [LBG+12], we implemented synchronized algorithms for both GraphLab and Giraph for the ease of comparison. We expect the observed relative performance trends to hold on other similar graph systems.

We deployed the systems on Aliyun ECS n2.large instances [ali], each powered by an Intel Xeon processor with 2.5GHz and 16G memory. We used up to 24 instances. We used ECS since its average inter-connection speed is close to real-life large-scale distributed systems. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency and Communication. We first evaluated the efficiency and communication of GRAPE over real-life graphs by varying the number n of processors used, from 4 to 24. We compared its performance with Giraph, GraphLab and Blogel. For SSSP and CC, we experimented with all three real-life datasets. For Sim and SubIso, we evaluated the queries over liveJournal and DBpedia, since these queries are meaningful on labeled graphs only, while traffic does not carry labels.

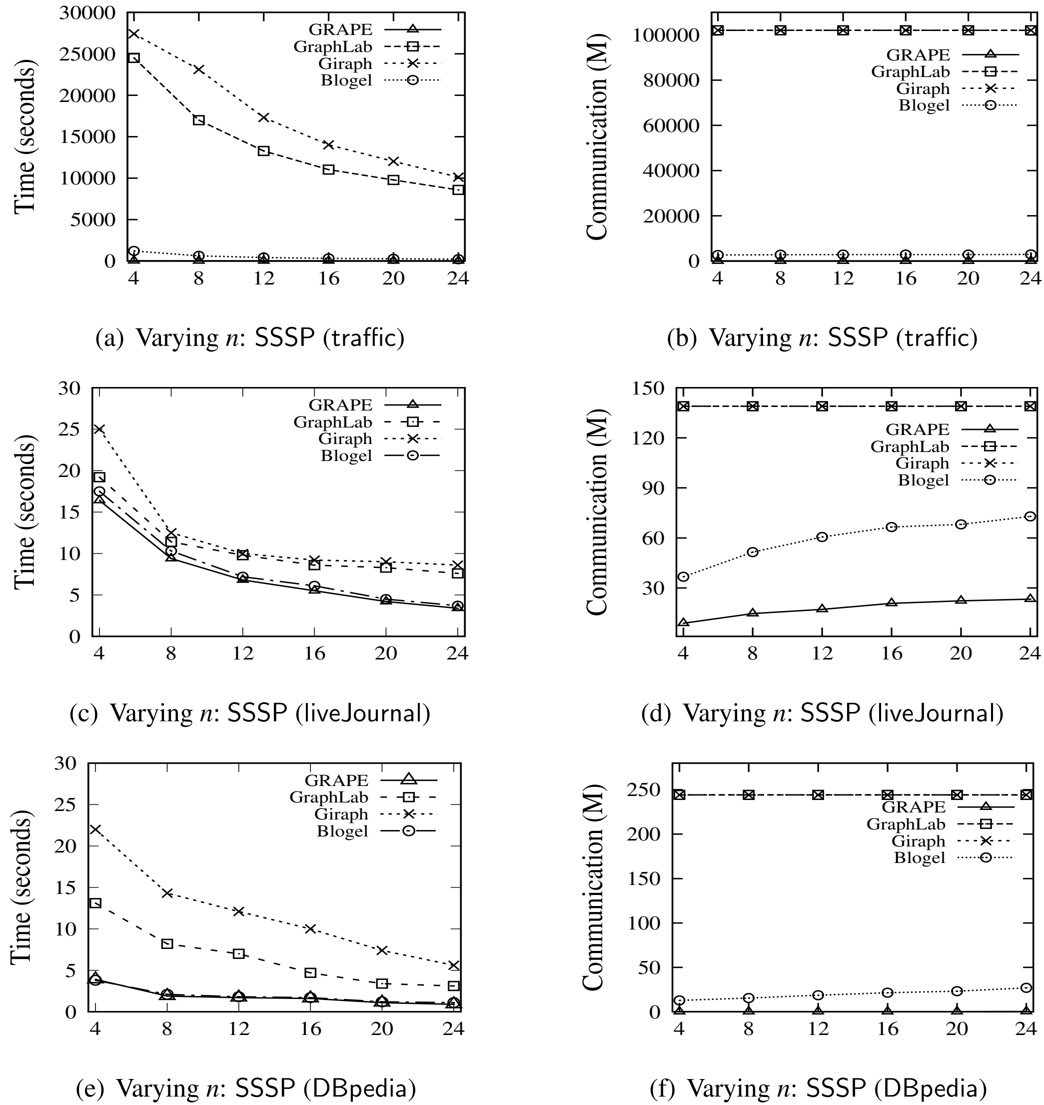
(1) SSSP. Figures 2.6(a)-2.6(e) report the performance of the systems for SSSP over traffic, liveJournal and DBpedia, respectively. From the results we can see the following.

(a) GRAPE outperforms Giraph, GraphLab and Blogel by 964, 818 and 22 times, respectively, over traffic with 24 processors (Fig 2.6(a)). In the same setting, it is 2.5, 2.2 and 1.1 times faster over liveJournal (Fig. 2.6(c)), and 6.2, 3.4 and 1.2 times faster over DBpedia (Fig. 2.6(e)). By simply parallelizing sequential algorithms without further optimization, GRAPE is comparable to the state-of-the-art systems in response time.

Note that the improvement of GRAPE over Giraph and GraphLab on traffic is much larger than on liveJournal and DBpedia. This is because vertex-centric algorithms take more supersteps to converge on graphs with large diameters, *e.g.,* traffic. Giraph takes 10752 supersteps over traffic, while 18 over liveJournal; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and is more robust; it takes 18 supersteps on traffic and 10 on liveJournal.

(b) All systems take less time when n increases, and GRAPE scales well with n. The speedup of GRAPE compared to Giraph and GraphLab becomes larger when more processors are used; *e.g.,* GRAPE is 818 times faster than Giraph with 4 processors, and is 964 times faster with 24 processors. On average, GRAPE is 4 times faster for n from 4 to 24, while it is 3 times for Giraph, 3.2 times for GraphLab and 5 times for Blogel. These verify the parallel scalability of GRAPE.

(c) GRAPE ships on average $9*10^{-6}\%$, 6.4% and 0.05% of the data shipped by Giraph,



(a) Varying n: SSSP (traffic)
(b) Varying n: SSSP (traffic)
(c) Varying n: SSSP (liveJournal)
(d) Varying n: SSSP (liveJournal)
(e) Varying n: SSSP (DBpedia)
(f) Varying n: SSSP (DBpedia)

Figure 2.6: Performance evaluation of SSSP

$9*10^{-6}$%, 6.4% and 0.05% of GraphLab, and $3.5*10^{-4}$%, 24% and 0.94% of Blogel, over traffic, liveJournal and DBpedia, respectively.

Figures 2.6(b)-2.6(f) show that both GRAPE and Blogel incurs communication costs that are orders of magnitudes smaller than those of GraphLab and Giraph (whose curves coincide). For instance, GRAPE ships 0.07% of the data shipped by GraphLab (same for Giraph) on DBpedia. This is because vertex-centric programming incurs a large number of inter-vertex messages. Both block-centric programs (Blogel) and PIE programs (GRAPE) effectively reduce unnecessary messages, and trigger inter-block messages only when necessary. We also observe that GRAPE ships 30% and 0.9% of the data shipped by Blogel over liveJournal and DBpedia, respectively. This is because GRAPE ships only updated values under monotonic condition. The improvement over

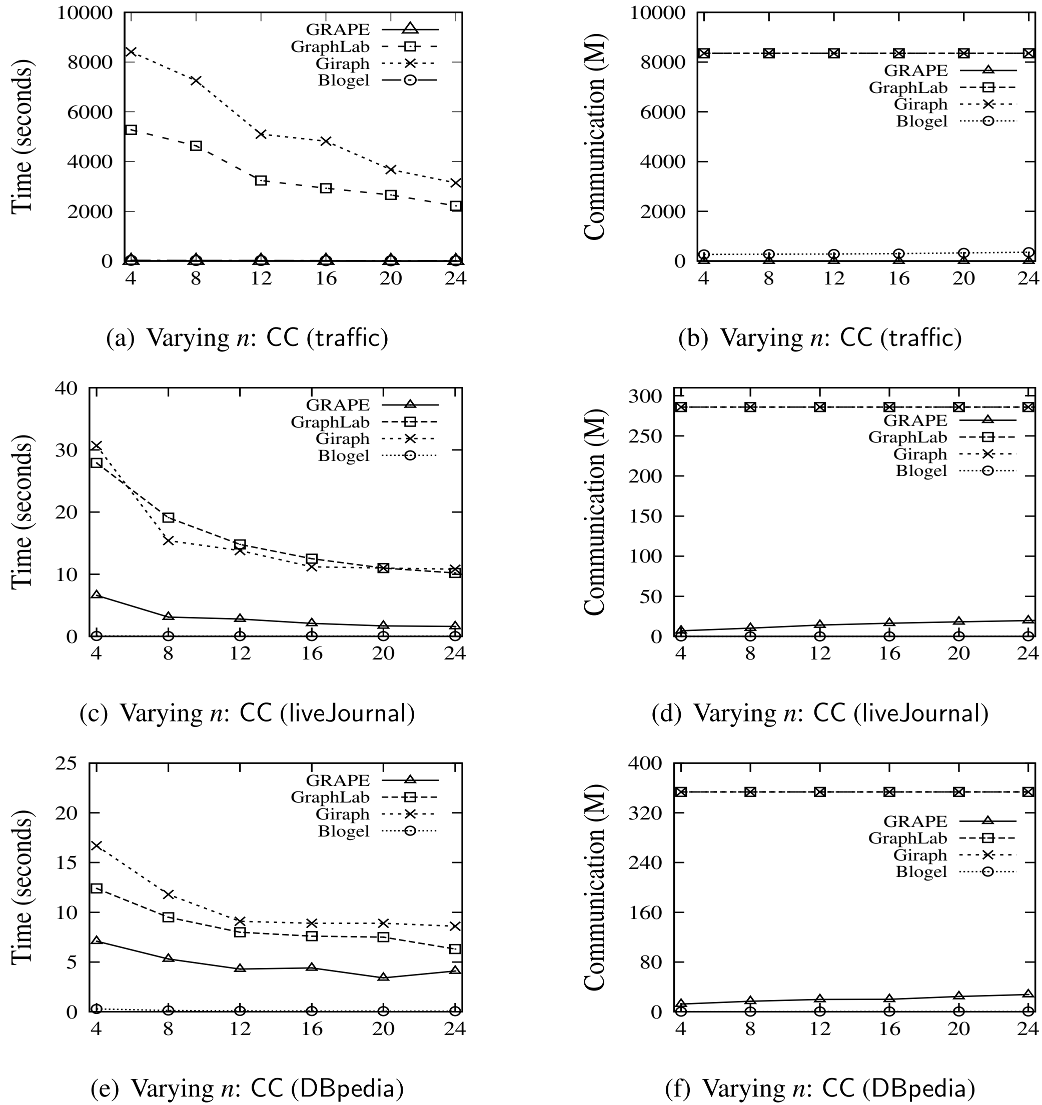
Blogel on traffic is not substantial because the road network has a small average node degree, and hence imposes a smaller bound (worst-case data shipment) on the improvement of GRAPE over Blogel.

In particular, GRAPE significantly reduces supersteps. It takes on average 12 supersteps, while Giraph, GraphLab and Blogel take 10752, 10752 and 1673 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs, and triggers cross-fragment communication only when necessary; moreover, IncEval ships only *changes* to status variable, which are updated monotonically (Theorem 1). In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages as required by recasted programs.

(2) CC. Figures 2.7(a)-2.7(c) report the performance for CC detection, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 24$, GRAPE is on average 4.4 and 4.0 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE, *e.g.,* 0.05s vs. 1.6s over liveJournal when $n = 24$. This is because Blogel embeds the computation of CCs in its graph partition phase as precomputation, while the partitioning cost (on average 16.2 seconds) is *not* included in the response time of Blogel. In other words, without precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel that is run over graphs already partitioned into connected components. (c) GRAPE incurs only 4.8% of communication cost of both Giraph and GraphLab on average, and is comparable to that of the near "optimal" case of Blogel.

Figures 2.7(b)-2.7(f) demonstrate similar improvement of GRAPE over GraphLab and Giraph for CC, *e.g.,* on average GRAPE ships 5.4% of the data shipped by Giraph and GraphLab. Blogel is slightly better than GRAPE. As remarked in Section 2.6 for Exp-1(2), this is because Blogel precomputes CCs of graphs when partitioning and loading the graphs, and thus already recognizes connected components by using an internal partition strategy. While a fair comparison should include the time for precomputing CCs in the evaluation time of CC by Blogel, we cannot identify the communication cost saved by its preprocessing. Thus, the reported communication cost of Blogel is almost 0 in all cases. Nonetheless, GRAPE incurs communication cost comparable to the near "optimal" case reported by Blogel, when Blogel operates on a graph that is already partitioned as CCs.

(3) Sim. Fixing $|Q| = (8, 15)$, *i.e.,* patterns Q with 8 nodes and 15 edges, we evaluated matching via graph simulation over liveJournal and DBpedia. As shown in Fig-



(a) Varying n: CC (traffic)

(b) Varying n: CC (traffic)

(c) Varying n: CC (liveJournal)

(d) Varying n: CC (liveJournal)

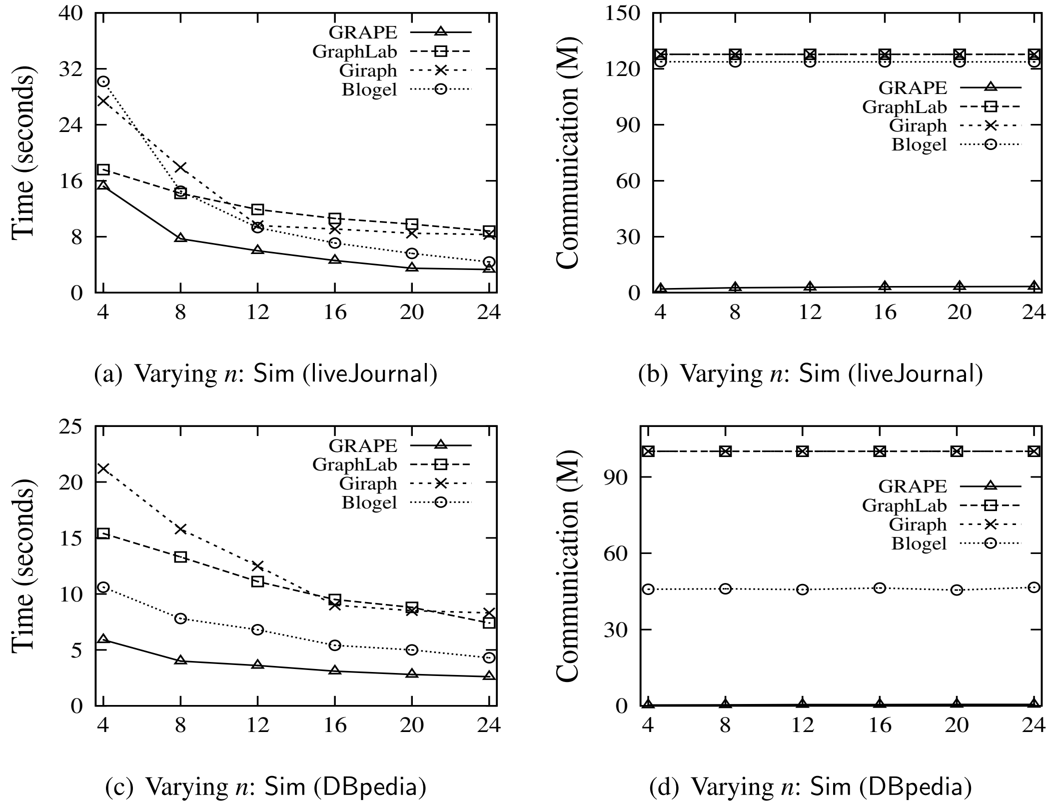
(e) Varying n: CC (DBpedia)

(f) Varying n: CC (DBpedia)

Figure 2.7: Performance evaluation of CC

ures 2.8(a)-2.8(c), (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 2.5, 2.7 and 1.3 times faster over liveJournal, and 3.2, 2.8 and 1.7 times faster over DBpedia on average, respectively, when $n = 24$. (b) GRAPE scales better with the number n of processors than Giraph and GraphLab, and is comparable to Blogel in parallel scalability. (c) GRAPE ships 2.2%, 2.2% and 2.3% (liveJournal), and 0.45%, 0.45% and 0.9% (DBpedia) of the data shipped by Giraph, GraphLab and Blogel on average, respectively, when $n = 24$.

Figures 2.8(b) and 2.8(d) report the communication cost for graph simulation over liveJournal and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average 1.3%, 1.3% and 1.6% of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that here the communication cost of Blogel is much



(a) Varying n: Sim (liveJournal)

(b) Varying n: Sim (liveJournal)

(c) Varying n: Sim (DBpedia)

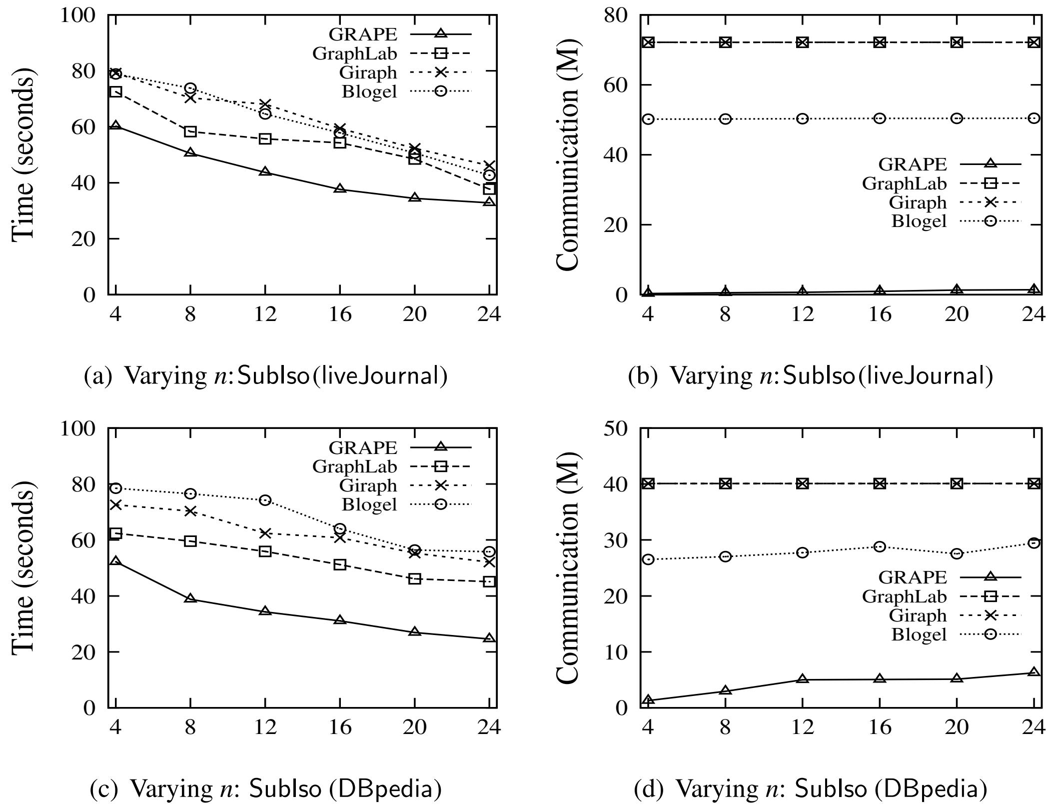
(d) Varying n: Sim (DBpedia)

Figure 2.8: Performance evaluation of Sim

higher than that of GRAPE, even though Blogel adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel does not help much on more complex queries. GRAPE works better than vertex-centric and block-centric systems on complex queries, by employing incremental IncEval to reduce excessive messages.

In particular, GRAPE takes at most 6 supersteps to terminate, while Giraph, GraphLab and Blogel take 7, 8 and 10 supersteps, respectively. This again empirically validates Theorem 1, which allows us to monotonically update status variables.

(4) SubIso. Fixing $|Q|=(6,10)$, we evaluated subgraph isomorphism. As shown in Figures 2.9(a)-2.9(c) over liveJournal and DBpedia, respectively, (a) GRAPE is on average 1.86, 1.49 and 1.98 times faster than Giraph, GraphLab and Blogel, respectively, when $n = 24$. (b) GRAPE does well over all queries tested. It takes 2 supersteps and 38.9 seconds on average, while Giraph, GraphLab and Blogel take 62.4, 54.3 and 64.5 seconds and 4, 4 and 6 supersteps, respectively. (c) GRAPE scales well with the number n of processors. (d) GRAPE incurs on average 5.9%, 5.9% and 8.4% of the communication cost of Giraph, GraphLab and Blogel, respectively, when $n = 24$.



(a) Varying n: SubIso(liveJournal)

(b) Varying n: SubIso(liveJournal)

(c) Varying n: SubIso (DBpedia)

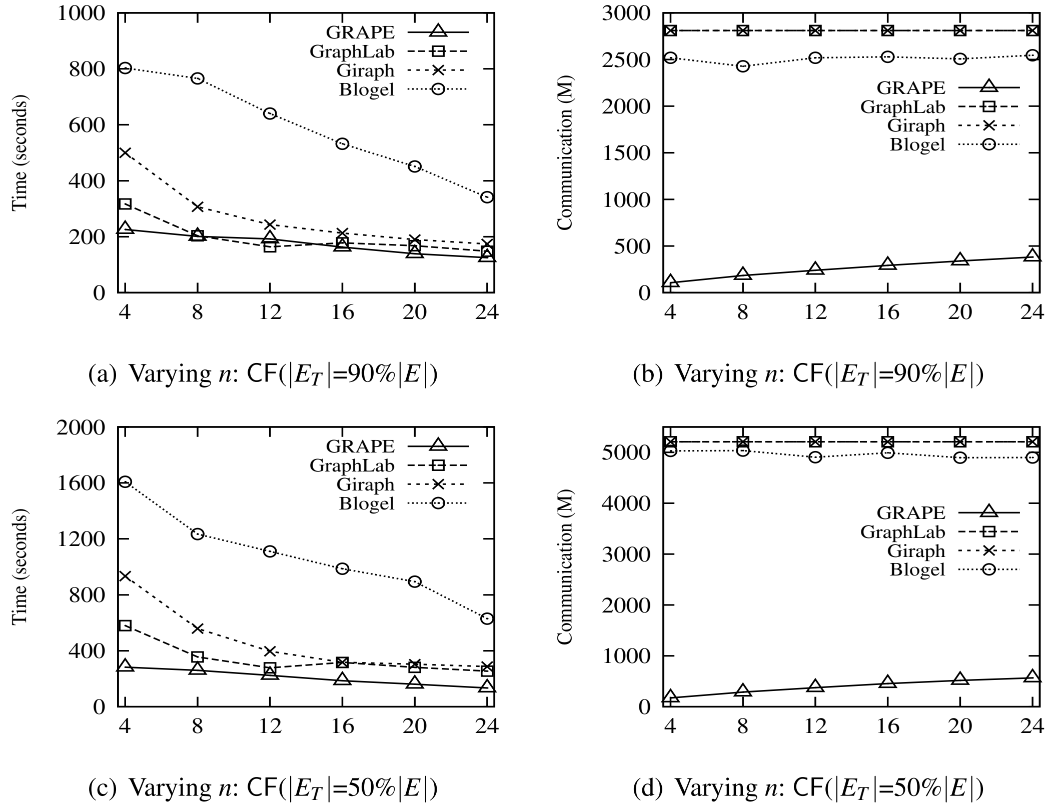
(d) Varying n: SubIso (DBpedia)

Figure 2.9: Performance evaluation of SubIso

Figures 2.9(b) and 2.9(d) report the results for SubIso over liveJournal and DBpedia, respectively. The results are consistent with Sim queries. On average, GRAPE ships 4.7%, 4.7%, and 6.5% of the data shipped by Giraph, GraphLab and Blogel, respectively. Due to the locality of subgraph isomorphism, matches to a pattern are confined in connected blocks. Hence Blogel takes advantage of its CC preserving graph partition, and does better than the case for Sim. Nevertheless, GRAPE only ships 6.5% of the data shipped by Blogel on average, and outperforms Blogel.

*(5) Collaborative filtering (*CF*).* For CF, we used movieLens [mov] with two training sets, compared with the built-in SGD-based CF in Giraph and GraphLab, and CF implemented for Blogel. We calibrated the termination condition of all the systems as the convergence point when the root-mean-square error of predicted ratings is less than a threshold.

We first tested training set $|E_T|$ = 90% $|E|$. Note that CF favors "vertex-centric" programming since each user or product node only needs to exchanges data with their neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, as shown in Fig. 2.10(a), GRAPE is on average 1.6, 1.1 and 3.4 times faster than Giraph,



(a) Varying n: CF($|E_T|$=90%$|E|$)

(b) Varying n: CF($|E_T|$=90%$|E|$)

(c) Varying n: CF($|E_T|$=50%$|E|$)

(d) Varying n: CF($|E_T|$=50%$|E|$)

Figure 2.10: Performance evaluation of CF

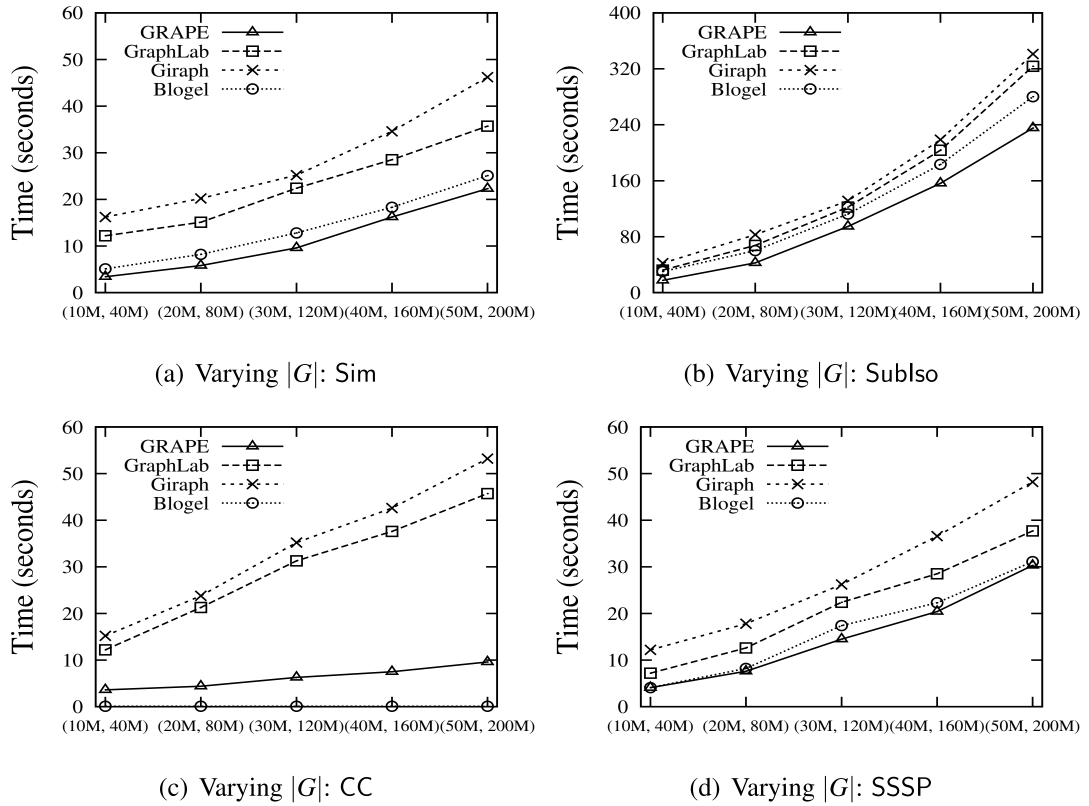
GraphLab and Blogel, respectively, when the number n of processors varies from 4 to 24. It scales well with n. In addition, it ships on average 9.2%, 9.2% and 10.2% of data shipped by Giraph, GraphLab and Blogel, respectively.

We also tested smaller training set ($|E_T|$ = 50% $|E|$). Figure 2.10(c) shows that GRAPE outperforms Blogel and Giraph, and is comparable with GraphLab. It ships at most 11.6% of data shipped by Giraph, GraphLab and Blogel.

Figures 2.10(b) and 2.10(d) report the results for CF over movieLens, with 90% and 50% training set E_T, respectively. On average, GRAPE ships 9.2%, 9.2%, and 10.3% of the data shipped by Giraph, GraphLab and Blogel, respectively, for $|E_T|$ = 90% $|E|$; and 7.6%, 7.6%, and 8.0% for $|E_T|$ = 50% $|E|$. This verifies that GRAPE is effective in reducing the communication cost of CF even for algorithms that favor vertex-centric programming. It also shows that GRAPE remains effective when the amount of training data varies.

Exp-2. Scalability.

We also evaluated the scalability of GRAPE over larger synthetic graphs. We developed a generator to produce graphs $G = (V, E, L)$ with L drawn from an alphabet $\mathcal{L}$



(a) Varying $|G|$: Sim

(b) Varying $|G|$: SubIso

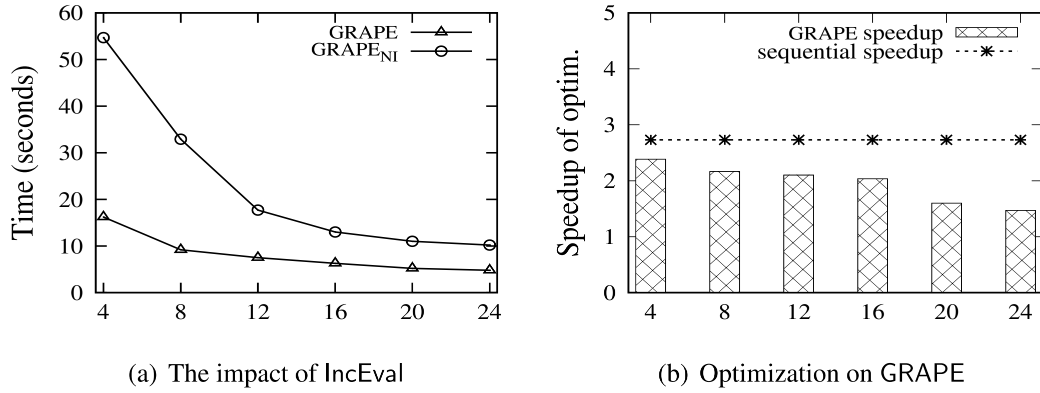
(c) Varying $|G|$: CC

(d) Varying $|G|$: SSSP

Figure 2.11: Scalability on synthetic graphs

of 50 labels. It is controlled by the numbers of nodes $|V|$ and edges $|E|$, up to $50M$ and $200M$, respectively.

Fixing n = 24, we varied $|G|$ from $(10M, 40M)$ to $(50M, 200M)$. As reported in Fig. 2.11, we tested SSSP, CC, Sim and SubIso; as the "true" behavior of CF is better characterized by real-world data, we omit the performance of CF over the synthetic data. The results are consistent with Fig. 5.6 over real-life graphs. (a) All systems take longer when G gets larger, as expected. (b) GRAPE scales reasonably well with the increase of $|G|$. With $|G|$ increased by 5 times, the running time of GRAPE increases by 7 times, 2.7 times, 6 times and 12 times, for SSSP (with linear time sequential algorithm), CC (linear time), Sim (quadratic time) and SubIso (exponential time), respectively. (c) GRAPE consistently outperforms Giraph and GraphLab for all queries, by 2.1 and 1.5 times for SSSP, 5.3 and 4.6 times for CC, 3 and 2.4 times for Sim, and 1.7 and 1.4 times for SubIso. The gap for SSSP is smaller than it on traffic, due to the special features of traffic mentioned earlier. GRAPE is 1.1 times faster than Blogel for SSSP, 1.3 for Sim, and 1.3 for SubIso. Blogel does better than GRAPE on CC for the reasons given above.



(a) The impact of IncEval

(b) Optimization on GRAPE

Figure 2.12: Incremental steps and optimization

Exp-3: Incremental computation. We evaluated the effectiveness of incremental IncEval. We implemented a batch version of GRAPE for Sim queries, denoted as GRAPE_{NI}, which uses PEval to perform iterative computations and handle the messages, instead of IncEval. It mimics the case when no incremental computation is used. As shown in Fig. 2.12 over liveJournal, (1) GRAPE outperforms GRAPE_{NI} by 2.1 times with 24 processors; and (2) the gap is larger when less workers are employed, *e.g.,* 3.4 times when 4 processors are used. This is because the less workers are used, the larger fragments reside at each worker, and as a consequence, heavier computation costs are incurred at each superstep. This verifies that incremental steps effectively reduces redundant local computations in iterative graph computations. The results on DBpedia are consistent and are not shown.

Exp-4. Compatibility. We also evaluated the compatibility of optimization strategies developed for sequential graph algorithms with GRAPE parallelization.

For a query class $\mathcal{Q}$, a sequential algorithm $\mathcal{A}$ and its optimized version $\mathcal{A}^*$ for $\mathcal{Q}$, denote the speedup of the optimization as $\frac{T(\mathcal{A})}{T(\mathcal{A}^*)}$. Denote the running time of GRAPE parallelization of $\mathcal{A}$ (resp. $\mathcal{A}^*$) as $T_p(\mathcal{A})$ (resp. $T_p(\mathcal{A}^*)$) for a given number n of workers. Ideally, $\frac{T(\mathcal{A})}{T(\mathcal{A}^*)}$ should be close to $\frac{T_p(\mathcal{A})}{T_p(\mathcal{A}^*)}$, *i.e.,* GRAPE *preserves* the speedup from the optimization. That is, the impact of the optimization is not "dampened out" by parallelization overhead such as synchronization and message passing.

We make a case for graph simulation. We evaluated two sequential algorithms, one from [HHK95], and the other is an optimized version that employs indices to reduce candidates [FLM+10]. Using Sim queries over liveJournal, we found that the average speedup of sequential algorithms is 2.7. Varying n from 4 to 24, we report the speedup of the parallelized algorithms of GRAPE in Fig. 2.12(b). The result on DBpedia are

consistent (not shown). The results suggest that the speedup is close to its sequential counterpart. Such optimization cannot be easily encoded in vertex programs of Giraph and GraphLab and the V-mode and B-mode programs of Blogel.

Exp-5: Ease of programming. We also inspected the usability of GRAPE. Taking SSSP as an example, we examined (a) vertex-centric programs for Giraph (similarly for GraphLab), and (b) block-centric programs for Blogel. Parts of the Giraph and Blogel algorithms are shown in Figures 2.13 and 2.14, respectively. We adopt the Giraph code taken from [MAB+10], and use the Blogel code from its developers.

Comparing these programs with their GRAPE counterpart (Figures 2.2 and 2.3), we find the following.

```
class ShortestPathVertex (Vertex<int, int, int>) {
  void Compute (MessageIterator msgs) {
    int mindist = IsSource(vertex_id) ? 0 : INF;
    while (msgs not empty)
      mindist = min(mindist, msgs.GetValue());
    if (mindist < GetValue())
      MutableValue() = mindist;
    OutEdgeIterator iter = GetOutEdgeIterator();
    while (iter not empty)
      SendMessageTo(iter.Target(), GetValue() + iter.GetValue());
    VoteToHalt(); }}
```

Figure 2.13: Giraph vertex program for SSSP

(1) The vertex program for Giraph requires substantial changes to its corresponding sequential algorithm. As shown in Fig. 2.13, the logic flow of a Giraph program for SSSP is quite different from that of a sequential SSSP algorithm. Writing such programs requires users to have prior knowledge about the query classes and the design principle of the vertex-centric model. Moreover, it is challenging to integrate graph-level optimization, *e.g.,* incremental evaluation, into the vertex programming model. In contrast, the logic flow of PIE algorithms (GRAPE) remains the same as those sequential algorithms adapted for PEval and IncEval.

```
 void VCompute(Messages) { /*V-mode computing*/
1.  if ( step == 1) {
2.      ··· /* initialize source distance, vote to halt otherwise */}
3.  else { for (msg : Messages) {
4.      ··· /* update local distance with minimum one in Messages*/ }}
 void BCompute(Messages, Container) {/*B-mode computing*/
1.  for (vertex : Container) {
2.      if (vertex.isactive()) { heap.add(vertex); }}
3.  while (heap.size > 0) { /*recasted Dijkstra's algorithm*/
4.      u = heap.peek(); edges = u.value().edges; split = u.value().split;
5.      for (edge : edges[0...split]) {
6.          ··· /* invokes V-mode computing for each in-block node*/}
7.      for (edge : edges[split...edge.size()]){
8.          ··· /* out-block msg passing */ } } }
9.  voteToHalt(); }
```

Figure 2.14: Blogel block program for SSSP

Similar to Giraph, GraphLab code for SSSP (not shown) requires users to recast the sequential SSSP algorithm into vertex programs. For example, a sequential operation in an SSSP algorithm that "collects the distances from the neighbors of a node and updates the distances" is broken down to two core functions as follows: (a) the "Apply" function updates the local distance at each vertex; and (b) the "Scatter" function propagates the updated value to the neighbors of a node. In contrast, a GRAPE program keeps the integrity of this operation for all the nodes within a fragment.

(2) While Blogel supports block-centric computation, it also requires recasting of sequential algorithms, as shown in Fig. 2.14. Indeed, Blogel programming extends vertex-centric algorithms (*e.g.,* Giraph) by treating each block as a "virtual vertex", while still retaining the same message passing strategies for blocks as in the vertex-centric algorithms. Hence, its logic flow is along the same lines as Giraph algorithms, and requires recasting of sequential algorithms.

Remark. The partitioning strategy does not affect the correctness: the Theorem 1

holds regardless any partitioning strategy, *i.e.,* GRAPE guarantees the termination and correctness if the sequential algorithms provided are correct and monotonic; there is no requirement on the partitioning strategy.

When it comes to efficient, the choice of different partitioning strategies has impact on some algorithms, especially on algorithms with the data locality, such as subgraph isomorphism. In this Chapter, we use pre-fetch to ensure the locality for the SubIso algorithm. In other Chapters, one may find some more complicated and carefully designed partitioning strategies employed to maximize the benefits from the locality with a property named parallel scalability.

Nonetheless, for most algorithms, like SSSP and CC, an arbitrary partitioning strategy would warrant good performance, *e.g.,* edge-cut, as long as it minimizes the number of crossing edge as usual. All the experiments in Chapter 2 used this setting. Since other platforms are based on the vertex-centric model, they do not support fragment-based partitioning. We employed their default partitioning for a fair comparison. *i.e.,* 2D-partition for GraphLab and Giraph, Voronoi partitioner for Blogel.

Summary. We find the following. (1) By plugging in sequential algorithms, GRAPE performs comparably to state-of-the-art systems. Over real-life graphs and using from 4 to 24 processors, GRAPE is on average 323, 274 and 7.9 times faster than Giraph, GraphLab and Blogel for SSSP, 2.7, 2.6 and 1.7 for Sim, 1.7, 1.4 and 1.7 for SubIso, and 1.9, 1.4 and 3.8 for CF, respectively. For CC, it is 3.9 and 3.8 times faster than Giraph and GraphLab, respectively, and is comparable to the "optimal" case of Blogel. The results on synthetic graphs are consistent. (2) Better still, GRAPE ships on average 5.6%, 5.6% and 10% of the data shipped by Giraph, GraphLab and Blogel for SSSP, 1.3%, 1.3% and 1.6% for Sim, 4.7%, 4.7% and 6.5% for SubIso, and 8.1%, 8.1% and 8.7% for CF, respectively, in the same setting. For CC, it incurs 7.3% and 7.3% of data shipment of Giraph and GraphLab, and is comparable with "optimized" Blogel. (3) Incremental steps effectively reduce iterative recomputation. For Sim, it improves the response time by 2.6 times on average. (4) GRAPE inherits the benefit of optimized sequential algorithms. For Sim, it is on average 2 times faster by using the algorithm of [FLM$^+$10] instead of [HHK95].

2.7 Related Work

The related work of this chapter is categorized as follows.

Parallel models and systems. Several parallel models have been studied for graphs, *e.g.,* PRAM [Val91], BSP [Val90] and MapReduce [DG08]. PRAM abstracts parallel RAM access over shared memory. BSP models parallel computations in supersteps (including local computation, communication and a synchronization barrier) to synchronize communication among workers. Pregel [MAB+10] (Giraph [Ave11]) implements BSP with vertex-centric programming, where a superstep executes a user-defined function at each vertex in parallel. GraphLab [LBG+12] revises BSP to pass messages asynchronously. Block-centric models [TBC+13, YCLN14] extend vertex-centric programming to blocks, to exchange messages among blocks.

Popular graph systems also include GraphX [XGFS13], GRACE [WXDG13], GPS [SW13], Trinity [SWL12], etc. GraphX [GXD+14] recasts graph computation in its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations over Spark platform. GRACE [WXDG13] provides an operator-level, iterative programming model to enhance synchronous BSP with asynchronous execution. GPS [SW13] implements Pregel with extended APIs and partition strategies. All these systems require recasting of sequential algorithms.

GRAPE adopts the synchronization mechanism of BSP. As opposed to the prior systems, (a) GRAPE aims to parallelize existing sequential algorithms, by combining partial evaluation and incremental computation. (b) As opposed to MapReduce, it highlights data-partitioned parallelism via graph fragmentation. For iterative computations, it does not need to ship the entire state of the graphs in each round [MAB+10]. (c) The vertex-centric model of Pregel (synchronized) is a special case of GRAPE, when each fragment is limited to a single vertex. The communications of Pregel are via "inter-processor" messages, and a message from a node often has to go through several supersteps to reach another node. GRAPE reduces excessive messages and scheduling cost of Pregel, since communications within the same fragment are local. GRAPE also facilitates graph-level optimizations that are hard to implement in vertex-centric systems; similarly for GraphLab (asynchronized). (d) Closer to GRAPE are block-centric models [TBC+13, YCLN14]. However, the programming interface of [TBC+13] is still vertex-centric, and [YCLN14] is a mix of vertex-centric and block-centric programming (V-compute and B-compute). The B-compute interface is essentially vertex-centric programming, by treating each block as a vertex. Users have to

recast existing sequential algorithms into a new model. In contrast, GRAPE "plugs in" sequential algorithms PEval and IncEval from GRAPE library, and applies them to blocks without recasting. None of the prior systems uses (bounded) incremental steps to speed up iterative computations. No one provides assurance on termination and correctness of parallel graph computations.

Partial evaluation has been studied for certain XML [BCFK06] and graph queries [FWWD14]. There has also been a host of work on incremental graph computation (*e.g.,* [RR96b, FWWD14]). This work makes a first effort to provide a uniform model by combining partial evaluation and incremental computation together, to parallelize sequential graph algorithms as a whole.

Parallelization of graph computations. A number of graph algorithms have been developed in MapReduce, vertex-centric models and others [YCX+14, FWWD14]. In contrast, GRAPE aims to parallelize existing sequential graph algorithms, without revising their logic and work flow. Moreover, parallel algorithms for MapReduce, BSP (vertex-centric or not) and PRAM can be easily migrated to GRAPE (Section 2.3.2).

Prior work on automated parallelization has focused on the instruction or operator level [RMM15, PNK+11] by breaking dependencies via symbolic and automata analyses. There has also been work at data partition level [ZLL+15], to perform multi-level partition ("parallel abstraction") and enable locality-optimized access to adapt to different parallel abstraction.

In contrast, GRAPE aims to parallelize sequential algorithms as a whole. It is to make parallel computation accessible to end users, while [RMM15, PNK+11, ZLL+15] target experienced developers of parallel algorithms. There have also been tools for translating imperative code to MapReduce, *e.g.,* word count [RFRS14]. GRAPE advocates a different approach, by parallelizing the runs of sequential graph algorithms to benefit from data-partitioned parallelism, without translation. This said, the techniques of [RFRS14, RMM15, PNK+11, ZLL+15] are complementary to GRAPE.

Simulation results. Prior work has mostly focused on simulations between variants of PRAM with different memory management strategies, to characterize bounds of slowdown for deterministic or randomized solutions [Har94]. There has also been recent work on simulation of PRAM on MapReduce and BSP [KSV10]. We present optimal deterministic simulation results of MapReduce, BSP and PRAM on GRAPE, adopting the notion of optimal simulations of [Val91].

2.8 Summary

In this chapter, We have given the design and framework of GRAPE. we proposed an approach to parallelizing sequential graph algorithms. For a class of graph queries, users can plug in existing sequential algorithms with minor changes. GRAPE parallelization guarantees to terminate with correct answers under a monotonic condition if the sequential algorithms are correct. Moreover, we prove that graph algorithms for existing parallel graph systems can be migrated to GRAPE, without incurring extra cost. We have verified that GRAPE achieves comparable performance to the state-of-the-art graph systems for various query classes, and that (bounded) IncEval reduces the cost of iterative graph computations.

Chapter 3

Association Rules Discovery on GRAPE

We presented the framework of GRAPE and showed that many classes of graph computations work well on it. However, real life graph applications are usually far more complicated. GRAPE has the flexibility to resolve these problems as well.

In this chapter, we propose an application, association rules with graph patterns, targeting on discovering regularities between entities on the social networks. As will be seen in this chapter, we study the top-k discovery problem and identifying the potential customers problem with association rules with graph patterns. While they are both NP-hard problems, we develop parallel algorithms on GRAPE with accuracy bound.

Association rules have been well studied for discovering regularities between items in relational data, for promotional pricing and product placements [AIS93, YDCL06]. They have a traditional form $X \Rightarrow Y$, where X and Y are disjoint itemsets.

There have been recent interests in studying associations between entities in social graphs. Such associations are useful in social media marketing; indeed, "90% of customers trust peer recommendations versus 14% who trust advertising"[tru], and "60% of users said Twitter plays an important role in their shopping" [Smi13]. Nonetheless, association rules for social graphs are more involved than rules for itemsets.

Example 4: (1) Association rules for social graphs are defined on graphs rather on itemsets. Below is an example.

- *If (a) x and x' are friends living in the same city c, (b) there are at least 3 French restaurants in c that x and x' both like, and if (c) x' visits a newly opened French restaurant y in c, then x may also visit y.*

The antecedent of the rule can be represented as a graph pattern Q_1 (with solid edges) shown in Fig. 3.1(a), and the consequent is indicated by a dotted edge visit(x, y). A succinct presentation of Q_1 associates integer 3 with "French Restaurant" to indicate its 3 copies. As opposed to conventional association rules, Q_1 specifies conditions as topological constraints: edges between customers (the friend relation), customers and restaurants (like, visit), city and restaurants (in), and between city and customers (live_in). In a social graph G, for x and y satisfying the antecedent Q_1 via graph pattern matching, we can recommend y to x.

(2) As opposed to rules for itemsets, association rules for social graphs may target social groups with multiple entities:

- *If (a) x, x_1 and x_2 are friends, (b) they all live in* `Ecuador`*, and (c) if x_1, x_2 both like Shakira's album y (a Colombian singer), then x may also like y.*

This rule is depicted in Fig. 3.1(b), in which a graph pattern Q_2 (excluding the dotted edge) specifies conditions for (x, y) as antecedent, and dotted edge like(x, y) indicates its consequent. We can use the rule to identify potential customers x of y, characterized by a social group of three members.

(3) Association rules with graph patterns conveniently extend data dependencies such as conditional functional dependencies (CFDs) [FGJK08] in the context of social networks.

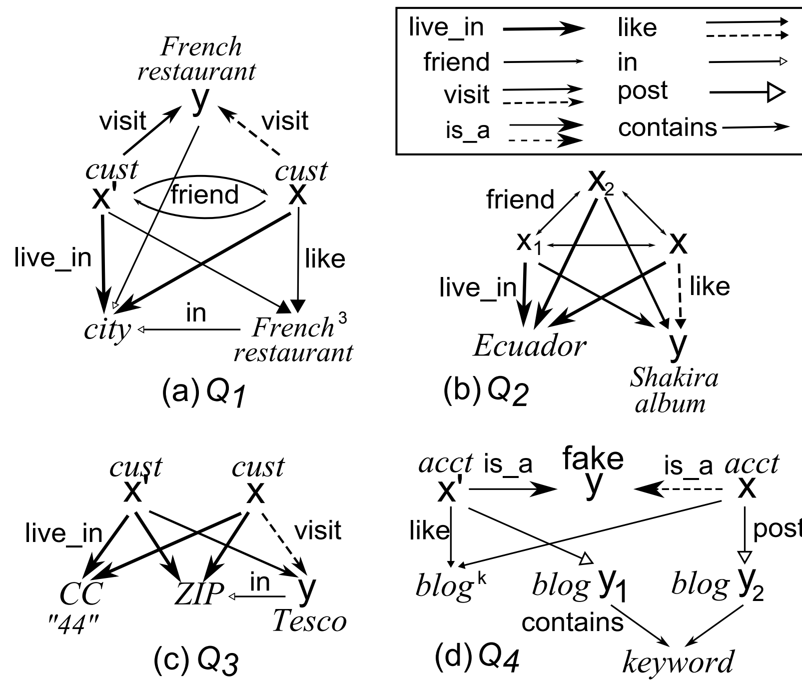


Figure 3.1: Associations as graph patterns

- *If the addresses of x and x' have the same country code "44" and same zip code, and if x' shops at a Tesco store y with the same zip, then x may also shop at y.*

Such a rule (Fig. 3.1(c)) embeds a corresponding CFD in its pattern Q_3, stating that if x and x' live in the UK with the same zip code, then they live on the same street. The rule is valid in the UK where zip code determines street.

(4) The applications of association rules are not limited to marketing activities. They also help us detect scams. As an example, the rule below is used to identify fake accounts [CSYP12].

- *If (a) account x' is confirmed fake, (b) both x and x' like blogs $P_1, \ldots, P_k$, (c) x posts blog y_1, (d) x' posts y_2, and (e) if y_1 and y_2 contain the same particular content (keyword), then x is likely a fake account.*

As depicted in Fig. 3.1(d), its antecedent is given by graph pattern Q_4 (excluding the dotted edge), and its consequent is the dotted edge is_a(x, *fake*). In a social graph G, the rule is to identify suspects for fake accounts, *i.e.,* accounts x that satisfy the structural constraints of pattern Q_4. □

The need for graph-pattern association rules (GPARs) is evident in social media marketing, community structure analysis, social recommendation, knowledge extrac-

tion and link prediction [LZ11]. Such rules, however, depart from association rules for itemsets, and introduce several challenges. (1) Conventional support and confidence metrics no longer work for GPARs. (2) Mining algorithms for traditional rules and frequent graph patterns cannot be used to discover practical diversified GPARs. (3) A major application of GPARs is to identify potential customers in social graphs. This is costly: graph pattern matching by subgraph isomorphism is intractable. Worse still, real-life social graphs are often big, *e.g.,* Facebook has 13.1 billion nodes and 1 trillion links [UKBM11].

3.1 Association via Graph Patterns

In this section we define graph-pattern association rules.

3.1.1 Graphs, Patterns, and Pattern Matching

We first review the notions of graphs and graph patterns.

Graphs. A *graph* is defined as $G = (V,E,L)$, 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 node v to v'; (3) each node v in V (resp. edge e) carries $L(v)$ (resp. $L(e)$), indicating its label or content *e.g.,* cust, French restaurant, 44 (resp. post, like), as found in social networks and property graphs.

Example 5: Two graphs G_1 and G_2 are shown in Fig. 3.2. (1) Graph G_1 depicts a restaurant recommendation network. For instance, cust_1 and cust_2 (labeled cust) live in New York; they share common interests in 3 French restaurants (marked with superscript 3 for simplicity); and they both visit a newly opened French restaurant “Le Bernadin” in New York. (2) Graph G_2 shows activities of social accounts. It contains (a) accounts $\text{acct}_1, \ldots, \text{acct}_4$ (labeled acct), (b) blogs $p_1, \ldots, p_7$; and (c) edges from accounts to blogs. For example, edge $\text{post}(\text{acct}_1, p_1)$ means that account acct_1 posts blog p_1, which contains keyword w_1 “claim a prize”. □

Patterns. A *pattern query* Q is a graph (V_p, E_p, f, C), in which V_p and E_p are the set of pattern nodes and edges, respectively; each node u_p in V_p (resp. edge e_p in E_p) has a label $f(u_p)$ (resp. $f(e_p)$) specifying a search condition, *e.g.,* city, or “44” for *value binding* (Q_3 of Example 4). For succinct representation, a node u_p can be labeled with an integer $C(u_p) = k$, indicating k copies of u_p with the same label and associated links in the common neighborhood.

Graph pattern matching. We first review two notions of subgraphs. (1) A graph $G' = (V',E',L')$ is a *subgraph of* $G = (V,E,L)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and moreover, for each edge $e \in E'$, $L'(e) = L(e)$, and for each $v \in V'$, $L'(v) = L(v)$. (2) We say that G' is *a subgraph induced by* a set V' of nodes if $G' \subseteq G$ and E' consists of all those edges in G whose endpoints are both in V'.

We adopt subgraph isomorphism for pattern matching. A *match* of pattern Q in graph G is a *bijective function* h from the nodes of Q to the nodes of a subgraph G' of G such that (a) for each node $u \in V_p$, $f(u) = L(h(u))$, and (b) (u,u') is an edge in Q if

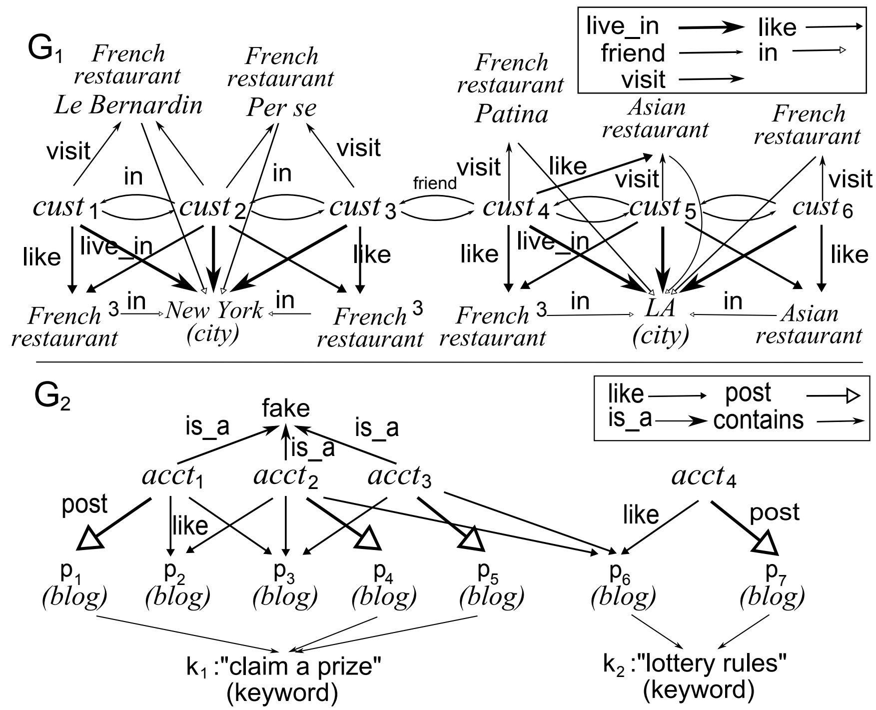


Figure 3.2: Labeled social graphs

and only if $(h(u), h(u'))$ is an edge in G', and $f(u, u') = L(h(u), h(u'))$. We say that G' *matches* Q.

Note that *similarity predicates* can be used instead of equality "=" with no impact on our algorithms.

We denote by $Q(G)$ the set of all matches of Q in G. For each pattern node u, we use $Q(u, G)$ to denote the set of all *matches* of u in $Q(G)$, *i.e.,* $Q(u, G)$ consists of nodes v in G such that there exists a function h under which a subgraph $G' \in Q(G)$ is isomorphic to Q, $v \in G'$ and $h(u) = v$.

Example 6: For Q_1 of Fig. 3.1 and G_1 of Fig. 3.2, a match in $Q_1(G)$ is $x \mapsto \text{cust}_1$, $x' \mapsto \text{cust}_2$, $city \mapsto$ New York, $y \mapsto$ Le Bernardin, and French restaurant^3 to 3 French restaurants. Here $Q_1(x, G_1)$ includes cust_1–cust_3 and cust_5.

□

A pattern $Q' = (V'_p, E'_p, f', C')$ is *subsumed by* another pattern $Q = (V_p, E_p, f, C)$, denoted by $Q' \sqsubseteq Q$, if (V'_p, E'_p) is a subgraph of (V_p, E_p), and functions f' and C' are restrictions of f and C in V, respectively. Observe that if $Q' \sqsubseteq Q$, then for any graph G' that matches Q, there exists a subgraph G'' of G' such that G'' matches Q'.

We will use the following notations. (1) For a pattern Q and a node x in Q, *the radius of Q at x*, denoted by $r(Q, x)$, is the longest distance from x to all nodes in Q

when Q is treated as an *undirected* graph.

(2) Pattern Q is *connected* if for each pair of nodes in Q, there exists an undirected path in Q between them. (3) For a node v_x in a graph G and a positive integer r, $N_r(v_x)$ denotes the set of all nodes in G within radius r of v_x. (4) The *size* $|G|$ of G is $|V|+|E|$, the number of nodes and edges in G. (5) Node v' is a *descendant* of v if there is a directed path from v to v' in G.

3.1.2 Graph Pattern Association Rules

We now define graph-pattern association rules.

GPARs. A *graph-pattern association rule* (GPAR) $R(x,y)$ is defined as $Q(x,y) \Rightarrow q(x,y)$, where $Q(x,y)$ is a graph pattern in which x and y are two *designated nodes*, and $q(x,y)$ is an edge labeled q from x to y, on which the same search conditions as in Q are imposed. We refer to Q and q as the *antecedent* and *consequent* of R, respectively.

The rule states that *for all nodes* v_x and v_y in a (social) graph G, if *there exists* a match $h \in Q(G)$ such that $h(x) = v_x$ and $h(y) = v_y$, *i.e.,* v_x and v_y match the designated nodes x and y in Q, respectively, then the consequent $q(v_x, v_y)$ will likely hold. Intuitively, v_x is a potential customer of v_y.

We model $R(x,y)$ as *a graph pattern* P_R, by extending Q with a (dotted) edge $q(x,y)$. We refer to pattern P_R as R when it is clear from the context. We treat $q(x,y)$ as pattern P_q, and $q(x,G)$ as the set of matches of x in G by P_q.

We consider practical and nontrivial GPARs by requiring that (1) P_R is connected; (2) Q is *nonempty*, *i.e.,* it has at least one edge; and (3) $q(x,y)$ does not appear in Q.

Example 7: Recall the first association rule described in Example 4. It can be expressed as a GPAR $R_1(x,y)$: $Q_1(x,y) \Rightarrow \mathsf{visit}(x,y)$, where its antecedent is the pattern Q_1 given in Example 4, and its consequent is $\mathsf{visit}(x,y)$. The GPAR can be depicted as the graph pattern of Fig. 3.1(a), by extending $Q_1(x,y)$ with a dotted edge for $\mathsf{visit}(x,y)$.

The last rule of Example 4 is written as $R_4(x,y)$: $Q_4(x,y) \Rightarrow \mathsf{is_a}(x,y)$, where in Q_4, y = *fake* is a value binding. The GPAR is depicted as the pattern of Fig. 3.1(d). In $\mathsf{is_a}(x,y)$, the same search condition y = *fake* is imposed. □

Remark. (1) To simplify the discussion, we define the consequent of GPAR with a single predicate $q(x,y)$ following [AIS93]. However, a consequent can be readily extended to *multiple* predicates and even to a *graph pattern*. (2) Conventional associ-

ation rules [AIS93] and a range of predication and classification rules [RVDB04] are a special case of GPARs, since their antecedents can be modeled as a graph pattern in which nodes denote items. Conditional functional dependencies [FGJK08] can also be represented by GPARs (see Q_3 of Fig. 3.1(c)).

3.2 Support and Confidence

We next define support and confidence for GPARs.

Support. The support of a graph pattern Q in a graph G, denoted by $\mathsf{supp}(Q,G)$, indicates how often Q is applicable. As for association rules for itemsets, the support measure should be *anti-monotonic*, *i.e.,* for patterns Q and Q', if $Q' \sqsubseteq Q$, then in any graph G, $\mathsf{supp}(Q',G) \geq \mathsf{supp}(Q,G)$.

One may want to define $\mathsf{supp}(Q,G)$ as *the number* $\|Q(G)\|$ of matches of Q in $Q(G)$, following its counterpart for itemsets [ZZ02]. However, as observed in [BN08, EASK14, JCZ13], this conventional notion is *not* anti-monotonic. For example, consider pattern Q' with a single node labeled cust, and Q with a single edge like(cust, F*rench restaurant*). When posed on G_1, $\|Q(G)\| = 18 > \|Q'(G)\| = 6$ (since French restaurant^3 denotes 3 nodes labeled French restaurant), although $Q' \sqsubseteq Q$.

To cope with this, we revise the support measure proposed in [BN08]. We define the support of the designated node x of Q as $\|Q(x,G)\|$, *i.e.,* the number of distinct matches of x in $Q(G)$. We define *the support of Q in G* as

$$\mathsf{supp}(Q,G) = \|Q(x,G)\|.$$

One can verify that this support measure is *anti-monotonic*.

For a GPAR $R(x,y)$: $Q(x,y) \Rightarrow q(x,y)$, we define

$$\mathsf{supp}(R,G) = \|P_R(x,G)\|,$$

by treating R as pattern $P_R(x,y)$ with designated nodes x,y.

Example 8: For GPAR $R_1(x,y)$: $Q_1(x,y) \Rightarrow \mathsf{visit}(x,y)$ of Example 7 and graph G_1 of Fig 3.2, (1) $\|Q_1(x,G_1)\| = 4$ (see Example 6); hence $\mathsf{supp}(Q_1,G_1)$ is 4; and (2) $\mathsf{supp}(R_1,G_1) = \|P_{R_1}(x,G_1)\| = 3$, where x has 3 matches cust_1–cust_3.

Similarly, consider $R_4(x,y)$: $Q_4(x,y) \Rightarrow \mathsf{is_a}(x,y)$ of Example 7 and graph G_2 in Fig 3.2, where $y = \mathit{fake}$. When $k = 2$, $\mathsf{supp}(R_4,G_2) = \mathsf{supp}(Q_4,G_2) = \|Q_4(x,G_2)\| = 3$, with matches acct_1–acct_3 for the designated node x in Q_4. □

Confidence. To find how likely $q(x,y)$ holds when x and y satisfy the constraints of $Q(x,y)$, we study the *confidence* of $R(x,y)$ in G, denoted as $\mathsf{conf}(R,G)$. One may want to adopt the conventional confidence for association rules, and define $\mathsf{conf}(R,G)$ as $\frac{\mathsf{supp}(R,G)}{\mathsf{supp}(Q,G)}$. That is, every match x in Q but not in R is considered as negative example for R. However, as observed in [GTHS13, Don14], the standard confidence is blind

to the distinction between "negative" and "unknown". This is particularly an overkill when G is incomplete [Don14, MZL12].

Example 9: Consider pattern Q_2 in Fig. 3.1(b). Let $Q_2(x,G)$ contain three matches v_1, v_2, v_3 of x_1, x_2, x_3 in a social graph G, all living in Ecuador, where (1) v_1 has an edge like to Shakira album, (2) v_2 has only a single edge like to MJ's album, and (3) v_3 has no edge of type like. Conventional confidence treats v_2 and v_3 both as negative examples, with $\mathsf{conf}(R_2,G) = \frac{1}{3}$. However, G may be incomplete: v_3 has not entered any albums she likes. Thus we should treat v_3 as "unknown", not as a counterexample to R_2. □

Indeed, closed world assumption may not hold for social network [MZL12]. To distinguish "unknown" cases from true negative for GPAR mining in incomplete social networks, we adopt the *local closed world assumption* [GTHS13, Don14], as commonly used in mining incomplete knowledge bases.

Local closed world assumption (LCWA). Given a predicate $q(x,y)$, we introduce the following notations.

(1) $\mathsf{supp}(q,G) = \|P_q(x,G)\|$, the number of matches of x;

(2) $\mathsf{supp}(\bar{q},G)$, the number of nodes u in G that (a) have the same label as x, (b) have at least one edge of type q, but (c) $u \notin P_q(x,G)$; and

(3) $\mathsf{supp}(Q\bar{q},G)$, the number of nodes that satisfy conditions (a) to (c) of (2), and are also in $Q(x,G)$.

Given an (incomplete) social network G and a predicate $q(x,y)$, the local closed world assumption (LCWA) distinguishes the following three cases for a node u.

(1) "positive" case, if $u \in P_q(x,G)$;

(2) "negative" case, for every u counted in $\mathsf{supp}(\bar{q},G)$; and

(3) "unknown" case, for every u that satisfies the search condition of x but has *no* edge labeled as q.

That is, G is assumed "locally complete": it either gives all correct local information of u in connection with predicate q, or knows nothing about q at node u (hence unknown cases).

Based on LCWA, we define conf (R, G) by revising Bayes Factor (BF) of association rules [LTP07] as follows:

$$\mathsf{conf}(R,G) = \frac{\mathsf{supp}(R,G) \, * \, \mathsf{supp}(\bar{q},G)}{\mathsf{supp}(Q\bar{q},G) \, * \, \mathsf{supp}(q,G)}.$$

Intuitively, $\mathsf{conf}(R,G)$ measures the product of *completeness* and *discriminant*. A GPAR $R(x,y)$ has a better completeness if it holds on more matches x of $Q(x,y)$, and is more discriminant if it is less likely to hold on more nodes from $Q\bar{q}$. In addition, BF-based $\mathsf{conf}(R,G)$ is better justified than conventional confidence. As verified in [KS96, LTP07], BF satisfies a set of principles for reasonable interestingness measures, including fixed under independence ($\mathsf{conf}(R,G)$ = 1 if Q and q are statistically independent), fixed under incompatibility ($\mathsf{conf}(R,G)$=0 if $\mathsf{supp}(R,G)$=0), and monotonicity (increases monotonically with $\mathsf{supp}(R,G)$ when$\mathsf{supp}(\bar{q},G)$, $\mathsf{supp}(Q,G)$ and $\mathsf{supp}(q,G)$ are fixed). Hence we adapt BF by incorporating LCWA and topological support.

Example 10: Consider GPAR R_2 and $Q_2(x,G)$ described in Example 9. Under the LCWA, match v_1 accounts for "positive" for R_2, while v_2 and v_3 are "negative" and "unknown", respectively. Indeed, assuming that G provides complete local information for v_2, then v_2 is a counter-example to people who live in Ecuador but do not like Shakira album; in contrast, G knows nothing about what albums v_3 likes.

One can see that $\mathsf{supp}(R_2,G) = 1$ (match v_1), $\mathsf{supp}(\bar{q},G) = 1$ (match v_2), $\mathsf{supp}(Q\bar{q},G)$ = 1 (match v_2), and $\mathsf{supp}(q,G)$ = 1 (match v_1). The BF-based confidence $\mathsf{conf}(R_2,G)$ is 1, larger than its conventional counterpart ($\frac{1}{3}$) as the LCWA removes the impact of the unknown case v_3. □

There are other alternatives to define support and confidence for GPARs. (1) Following minimum image-based support [BN08], $\mathsf{supp}(R,G)$ can be defined as the the maximum number of matches for x in non-overlap matches (*i.e.,* no shared nodes and edges) of R. However, this excludes potential customers from matches that share even a single node (*e.g.,* only one of the three matches cust_1-cust_3 of Fig. 3.2 is counted), and thus underestimates the significance. (2) Similar to PCA confidence [GTHS13], $\mathsf{conf}(R,G)$ can be computed as $\frac{\mathsf{supp}(R,G)}{\mathsf{supp}(Q\bar{q},G)}$ under LCWA. However, this only considers the "coverage" of R instead of its interestingness in terms of completeness and discriminant [KS96, LTP07] (see Section 3.5).

Remark. We identify the following two "trivial" cases when $\mathsf{conf}(R,G) = \infty$: (1) $\mathsf{supp}(Q\bar{q},G)$ is 0, which interprets R as a logic rule that holds on the entire G, *i.e.,* "if v is in $Q(x,G)$ then v is a match in $P_q(x,G)$ (hence $P_R(x,G)$)"; and (2) $\mathsf{supp}(q,G)$ = 0, which means that $q(x,y)$ in R specifies no user in G; hence R should be discarded as uninteresting case. These two cases can be easily detected and distinguished in the GPAR discovery process (see Section 3.3).

symbols	notations
$Q(x,G)$	the set of distinct nodes that match x in $Q(G)$
$R(x,y)$	GPAR $Q(x,y) \Rightarrow q(x,y)$, represented as pattern P_R
$r(Q,x)$	the radius of Q at node x
$N_r(v_x)$	the set of nodes within radius r of v_x
$\mathsf{supp}(Q,G)$	the number $\|Q(x,G)\|$ of distinct matches of x in $Q(G)$
$\mathsf{conf}(Q,G)$	$(\mathsf{supp}(R,G) * \mathsf{supp}(\bar{q},G))/(\mathsf{supp}(Q\bar{q},G) * \mathsf{supp}(q,G))$
$\Sigma(x,G,\eta)$	$\{v_x \mid v_x \in Q(x,G), Q \Rightarrow q \in \Sigma, \mathsf{conf}(R,G) \geq \eta\}$

Table 3.1: Notations in Chapter 3

The notations of this chapter are summarized in Table 3.1.

3.3 Diversified Rule Discovery

We now study how to discover useful GPARs.

3.3.1 The Diversified Mining Problem

We are interested in GPARs for a particular event $q(x,y)$. However, this often generates an excessive number of rules, which often pertain to the same or similar people [XCYH06, AYLVY09].

This motivates us to study a diversified mining problem, to discover GPARs that are *both* interesting and diverse.

Objective function. To formalize the problem, we first define a function $\mathsf{diff}(,)$ to measure the difference of GPARs. Given two GPARs R_1 and R_2, $\mathsf{diff}(R_1,R_2)$ is defined as

$$\mathsf{diff}(R_1,R_2) = 1 - \frac{|P_{R_1}(x,G) \cap P_{R_2}(x,G)|}{|P_{R_1}(x,G) \cup P_{R_2}(x,G)|}$$

in terms of the Jaccard distance of their match set (as social groups). Such diversification has been adopted to battle against over-concentration in social recommender systems when the items recommended are too "homogeneous" [AYLVY09].

Given a set L_k of k GPARs that pertain to the same predicate $q(x,y)$, we define the objective function $F(L_k)$ again by following the practice of social recommender systems [GS09]:

$$(1-\lambda) \sum_{R_i \in S} \frac{\mathsf{conf}(R_i)}{N} + \frac{2\lambda}{k-1} \sum_{R_i, R_i \in S, i<j} \mathsf{diff}(R_i, R_j).$$

This, known as *max-sum diversification*, aims to strike a balance between interestingness (measured by revised Bayes Factor) and diversity (by distance $\mathsf{diff}(,)$) with a parameter λ controlled by users. We consider nontrivial GPARs (Section 3.2) with $\mathsf{conf}(R,G) \in [0, \mathsf{supp}(R,G) * \mathsf{supp}(\bar{q},G)]$, and normalize (1) the confidence metric with $N = \mathsf{supp}(q,G) * \mathsf{supp}(\bar{q},G)$ (a constant for fixed $q(x,y)$), and (2) the diversity metric with $\frac{2\lambda}{k-1}$, since there are $\frac{k(k-1)}{2}$ numbers for the difference sum, while only k numbers for the confidence sum.

Example 11: Consider GPARs R_1 of Fig. 3.1, and R_7 and R_8 shown in Fig. 3.3, all pertaining to visits(x, French restaurant). Then in graph G_1 (Fig. 3.2), (1) $\mathsf{supp}(q,G_1) = 5$ (cust$_1$-cust$_4$, cust$_6$), $\mathsf{supp}(\bar{q},G_1) = 1$ (cust$_5$); (2) $R_1(x,G_1) = R_7(x,G_1) = \{\mathsf{cust}_1, \mathsf{cust}_2, \mathsf{cust}_3\}$,

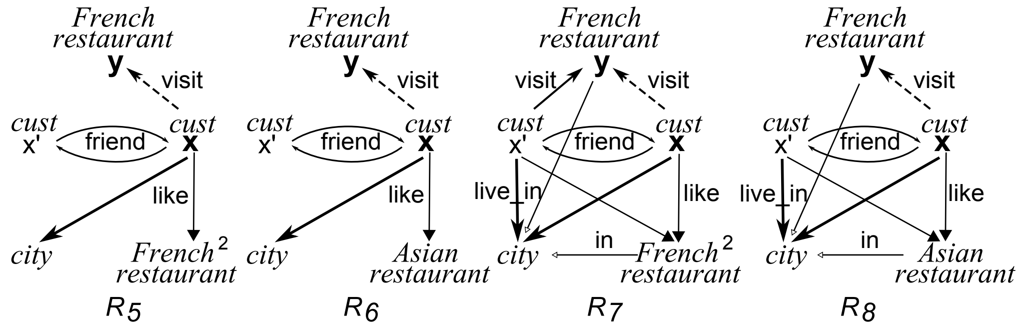


Figure 3.3: Diversified GPARs

$R_8(x,G_1)$ = {cust$_6$}; (3) conf(R_1,G_1) = conf(R_7,G_1) = 0.6, conf(R_8,G_1) = 0.2; and (4) diff(R_1,R_7) = 0, diff(R_1,R_8) = diff(R_7,R_8) = 1.

For $\lambda = 0.5$, a top-2 diversified set of these GPARs is $\{R_7,R_8\}$ with $F(R_7,R_8) = 0.5*\frac{0.8}{5}+1*1 = 1.08$ (similarly for $\{R_1,R_8\}$). Indeed, R_7 and R_8 find two disjoint customer groups sharing interests in French restaurant and Asian restaurant, respectively, with their friends.

□

Problem. Based on the objective function, the *diversified* GPAR *mining problem* (DMP) is stated as follows.

- *Input*: A graph G, a predicate $q(x,y)$, a support bound σ and positive integers k and d.
- *Output*: A set L_k of k nontrivial GPARs pertaining to $q(x,y)$ such that (a) $F(L_k)$ is maximized; and (b) for each GPAR $R \in L_k$, supp$(R,G) \geq \sigma$ and $r(P_R,x) \leq d$.

DMP is a bi-criteria optimization problem to discover GPARs for a particular event $q(x,y)$ with high support, bounded radius, and a balanced confidence and diversity. In practice, users can freely specify $q(x,y)$ of interests, while proper parameters (*e.g.,* support, confidence, diversity) can be estimated from query logs or recommended by domain experts.

The problem is nontrivial. Consider its decision problem to decide whether there exists a set L_k of k GPARs with $F(L_k) \geq B$ for a given bound B. One can show the following by reduction from the dispersion problem (cf. [GS09]).

Proposition 3: *The* DMP *decision problem is* NP-*hard.* □

3.3.2 Discovery Algorithm

As claimed in Chapter 2, to solve common and well studied graph computation problems, one can make use of existing sequential algorithms and their incremental version. However, the DMP (as well as the problems in the following chapters) is a new problem and no algorithms are in place for it. Hence we develope parallel algorithms and provide them in the Algorithm Library of GRAPE as out-of-the-box applications, leaving the PIE API to users for solving simple problems.

One might want to follow a "discover and diversify" approach that (1) first finds all GPARs pertaining to $q(x,y)$ by frequent graph pattern mining [PH02], and then (2) selects top-k GPARs via result diversification [GS09]. However, this is costly: (a) an excessive number of GPARs are generated; and (b) for all GPARs R generated, it has to compute $\mathsf{conf}(R,G)$ and their pairwise distances, and moreover, pick a top-k set based on $F()$; the latter is an intractable process itself.

One can do it more efficiently, with accuracy guarantees.

Theorem 4: *There exists a parallel algorithm for* DMP *that finds a set L_k of top-k diversified* GPARs *such that (a) L_k has approximation ratio* 2*, and (b) L_k is discovered in d rounds by using n processors, and each round takes at most $t(|G|/n,k,|\Sigma|)$ time, where Σ is the set of* GPARs *$R(x,y)$ such that* $\mathsf{supp}(R,G) \geq \sigma$ *and $r(P_R,x) \leq d$.* □

Here $t(|G|/n,k,|\Sigma|)$ is a function that takes $|G|/n$, k and $|\Sigma|$ as parameters, *rather than* the size $|G|$ of the entire G.

As a proof, we give such an algorithm, denoted as DMine and shown in Fig. 3.4. It designates one processor as *coordinator* S_c and the rest as *workers* S_i. It works as follows.

(1) It divides G into $n-1$ fragments $(F_1,\ldots,F_{n-1})$ such that (a) for each "candidate" v_x that satisfies the search condition on x in $q(x,y)$, its *d-neighbor* $G_d(v_x)$, *i.e.,* the subgraph of G induced by $N_d(v_x)$, is in some fragment; and (b) the fragments have roughly even size. These are possible since 98% of real-life patterns have radius 1, 1.8% have radius 2 [GFMPdlF11], and the average node degree is 14.3 in social graphs [BW13]; thus $G_d(v_x)$ is typically small compared with fragment size.

Fragment F_i is stored at worker S_i, for $i \in [1,n-1]$.

(2) DMine discovers GPARs *in parallel* by following bulk synchronous processing, in d rounds. The coordinator S_c maintains a list L_k of diversified top-k GPARs, initially

empty. In each round, (a) S_c posts a set M of GPARs to all workers, initially $q(x,y)$ only; (b) each worker S_i generates GPARs *locally* at F_i in parallel, by extending those in M with new edges if possible; (c) these GPARs are collected and assembled by S_c in the barrier synchronization phase; moreover, S_c *incrementally* updates L_k: it filters GPARs that have low support or cannot make top-k *as early as possible*, and prepares a set M of GPARs for expansion in the next round.

As opposed to the "discover and diversify" method, DMine (a) combines diversifying into discovering to *terminate* the expansion of non-promising rules *early*, rather than to conduct diversifying after discovering; and (b) it *incrementally* computes top-k diversified matches, rather than recomputing the diversification function $F()$ starting from scratch.

We next present the details of algorithm DMine.

Auxiliary structures. Algorithm DMine maintains the following: (a) at the coordinator S_c, a set L_k to store top k GPARs, and a set Σ to keep track of generated GPARs; and (b) at each worker S_i, a set C_i of candidates v_x for x at F_i.

Messages. In each round, coordinator S_c and workers S_i communicate via messages. (1) Each worker S_i generates a set M_i of messages. Each message is a triple $<R$, conf, flag$>$, where (a) R is a GPAR generated at S_i, (b) conf includes, *e.g.*, $\mathsf{supp}(R(x,y),F_i)$ and $\mathsf{supp}(Q\bar{q}(x,y),F_i)$, and (c) a Boolean flag to indicate whether R can be extended at S_i. (2) After receiving M_i, S_c generates a set M of messages, which are GPARs to be extended in the next round.

Algorithm. DMine initializes L_k and Σ as empty, and M as $\{q(x,y)\}$ (line 1). For r from 1 to d, it improves L_k by incorporating GPARs of radius r (lines 2-11), following a *levelwise* approach. In each round, it invokes localMine with M at all workers (line 4). Below we present the details.

Parallel GPARs generation (line 13). In the first round, procedure localMine receives $q(x,y)$ from S_c, and computes the following: (a) three sets: C_i, nodes v_x that satisfy the search condition of x in discovered GPARs, $P_q(x,F_i)$, matches of x in $q(x,y)$, and $\bar{q}(x,F_i)$, nodes v in F_i that account for $\mathsf{supp}(\bar{q},F_i)$ (Section 3.1.2); and (b) $\mathsf{supp}(q,F_i) = \|P_q(x,F_i)\|$, $\mathsf{supp}(\bar{q},F_i) = \|P_{\bar{q}}(x,F_i)\|$. Note that $\mathsf{supp}(q,F_i)$ and $\mathsf{supp}(\bar{q},F_i)$ never change and hence are derived *once for all*. Each match $v_x \in q(x,F_i)$ is referred to as a *center node*.

In round r, upon receiving M from S_c, localMine does the following. For each

---

Algorithm DMine

Input: A graph G, $q(x,y)$, bound σ, and positive integers k and d.
Output: A set L_k of top-k diversified GPARs.

/* executed at coordinator */
1. $L_k := \emptyset$; $\Sigma := \emptyset$; $r := 1$; $M := \{q(x,y)\}$;
2. **while** $r \leq d$ **do**
3. $\quad r := r + 1$;
4. $\quad$ post M to all workers and invoke localMine (M) in parallel;
5. $\quad$ collect in ΔE candidate GPARs in M_i from all workers;
6. $\quad$ check automorphism and assemble confidence for these GPARs;
7. $\quad \Delta E$ includes R with $\text{supp}(R,G) \geq \sigma$; $\Sigma := \Sigma \cup \Delta E$; $M := \emptyset$;
8. $\quad$ **for each** GPAR $R \in \Delta E$ **do**
9. $\quad\quad$ incDiv (L_k, R, Σ); /* incrementally update L_k, prune $\Sigma, \Delta E$ */
10. $\quad\quad$ **if** R is "extendable"
11. $\quad\quad$ **then** $M := M \cup \{R\}$; /* next round */
12. **return** L_k;

/* executed at each worker S_i in parallel, upon receiving M */
13. $\Sigma_i :=$ localMine (M);
14. construct message set M_i from Σ_i;
15. send M_i to the coordinator;

Figure 3.4: Algorithm DMine

GPAR $R(x,y) : Q(x,y) \Rightarrow q(x,y)$ in M, and each center node v_x, it expands Q by including *at least* one *new edge* that is at hop r from v_x, for all such edges.

<u>*Message construction*</u> (lines 14–15). For each GPAR $R(x,y)$: $Q(x,y) \Rightarrow q(x,y)$, its *local confidence* conf is computed: (1) $\text{supp}(R,F_i)$ and $\text{supp}(Q,F_i)$ count nodes in $P_q(x,F_i)$ and C_i that match x in $R(x,y)$ and $Q(x,y)$, respectively; and (2) $\text{supp}(Q\bar{q},F_i)$ = $\|Q(x,F_i) \cap P_{\bar{q}}(x,F_i)\|$. Then conf contains $\text{supp}(R,F_i)$, $\text{supp}(Q\bar{q},F_i)$, $\text{supp}(q,F_i)$ and $\text{supp}(\bar{q}(x,F_i))$; where $\text{supp}(q,F_i)$ and $\text{supp}(\bar{q},F_i)$ values are from the first round. A Boolean flag is also set to indicate whether R can be extended by checking whether

there exists a center node v_x that has edges at $r+1$ hops from v_x. Message M_i includes $<R, \mathsf{conf}, \mathsf{flag}>$for each R, and is sent to S_c.

<u>*Message assembling*</u> (lines 4-7). Upon receiving M_i from each S_i, coordinator S_c does the following. (1) It groups automorphic GPARs from all M_i. (2) For each group of m_i = $<R$, conf_i, $\mathsf{flag}_i>$ that refers to the same (automorphic) R, it assembles $\mathsf{conf}(R)$ into a single m = $<R$, $\mathsf{conf}(R,G)$, $\mathsf{flag}>$, where (a) $\mathsf{conf}(R,G)$= $\frac{\sum \mathsf{supp}(R,F_i) \sum \mathsf{supp}(\bar{q},F_i)}{\sum \mathsf{supp}(Q\bar{q},F_i) \sum \mathsf{supp}(q,F_i)}$; and (b) flag is the disjunction of all flag_i, for $i \in [1,n-1]$. This suffices since by the partitioning of graph G, nodes accounted for local support in F_i are disjoint from those in F_j if $i \neq j$; hence $\mathsf{conf}(R)$ can be directly assembled from local conf from F_i. Similarly, $\mathsf{supp}(R,G) = \sum_{i\in[1,n-1]} \mathsf{supp}(R,F_i)$. For each GPAR R, if $\mathsf{supp}(R,G) \geq \sigma$, it is added to ΔE and Σ.

<u>*Incremental diversification*</u> (lines 8-9). Next, DMine incrementally updates L_k by invoking procedure incDiv.

It uses a max priority Queue of size $\lceil \frac{k}{2} \rceil$, where (1) each element in Queue is a pair of GPARs, and (2) all GPAR pairs in Queue are pairwise disjoint. In round r, starting from Queue of top-k diversified GPARs with radius at most $r-1$, DMine improves Queue by incorporating pairs of GPARs from ΔE, with radius r. (1) If Queue contains less than $\lceil \frac{k}{2} \rceil$ GPARs pairs, incDiv iteratively selects two distinct GPARs R and R' from ΔE that maximize a revised diversification function:

$$F'(R,R') = \frac{1-\lambda}{N(k-1)}(\mathsf{conf}(R) + \mathsf{conf}(R')) + \frac{2\lambda}{k-1}\mathsf{diff}(R,R').$$

and insert (R,R') into Queue, until $|\text{Queue}|$ = $\lceil \frac{k}{2} \rceil$. It bookkeeps each pair (R,R') and $F'(R,R')$. (2) If $|\text{Queue}|$=$\lceil \frac{k}{2} \rceil$, for each new GPAR $R \in \Delta E$ (not in any pair of Queue) and $R' \in \Sigma$, it incrementally computes and adds a new pair $(R,R') \in \Delta E \times \Sigma$ that maximizes $F'(R,R')$ to Queue. This ensures that a pair (R_1,R_2) with minimum $F'(R_1,R_2)$ is replaced by (R,R'), if $F'(R_1,R_2) < F'(R,R')$.

After all GPAR pairs are processed, incDiv inserts R and R' into L_k, for each GPARs pairs $(R,R') \in$ Queue.

<u>*Message generation at S_c*</u> (lines 10-11). DMine next selects promising GPARs for further parallel extension at the workers. These include $R \in \Delta E$ that satisfy two conditions: (1) $\mathsf{supp}(R,G) \geq \sigma$, since by the anti-monotonic property of support, if $\mathsf{supp}(R,G)$ $< \sigma$, then any extension of R cannot have support no less than σ; and (2) R is "Extendable", *i.e.,* flag = true in $<R$, conf, flag$>$. It includes such R in M, and posts M to all workers in the next round.

Example 12: Suppose that graph G_1 in Fig. 3.2 is distributed to two workers S_1 and S_2, where S_1 (resp. S_2) contains subgraphs induced by cust_1-cust_3 (resp. cust_4-cust_6) and their 2-hop neighborhoods in G_1. Let predicate q be visits(x, French restaurant), λ=0.5, d=2 and k=2. We demonstrate algorithm DMine using example GPARs R_5-R_8 (Fig. 3.3).

(1) Coordinator S_c sends q to all workers, and computes $\mathsf{supp}(q, G_1) = 5$ (cust_1-cust_4, cust_6), $\mathsf{supp}(\bar{q}, G_1) = 1$ (cust_5).

(2) In round 1, R_5 (among others) is generated at S_1 from 1-hop neighbors of cust_1-cust_3, which are matches in $q(x, G_1)$ (Fig. 3.3). At S_2, R_5 and R_6 are generated by expanding cust_4 and cust_6. Local messages M_i from S_i include the following:

site	message	GPAR	$R(x,G_1)$	$Q\bar{q}(x,y)$	flag
S_1	M_1	R_5	cust_1-cust_3	$\emptyset$	T
S_2	M_2	R_5	cust_4	cust_5	T
		R_6	cust_4,cust_6	cust_5	T
S_c	M	R_5	cust_1-cust_4	cust_5	T
	M	R_6	cust_4,cust_6	cust_5	T

Table 3.2: Running example for DMine, round 1

(3) Coordinator S_c assembles M_1 and M_2, and builds ΔE including $\{R_5, R_6\}$. It computes $\mathsf{conf}(R_5) = 0.8$, $\mathsf{conf}(R_6) = 0.4$, $\mathsf{diff}(R_5, R_6) = 0.8$. It updates $L_k = \{R_5, R_6\}$, with $F'(R_5, R_6) = 0.5 * \frac{1.2}{5} + 1 * 0.8 = 0.92$. It includes R_5 and R_6 in message M (the table above), and posts it to S_1 and S_2.

(4) In round 2, R_5 is extended to R_7 and R_1 at S_1 and S_2, and R_6 to R_8 at S_2 (Fig. 3.3); the messages include:

site	message	GPAR	$R(x,G_1)$	$Q\bar{q}(x,y)$	flag
S_1	M_1	R_7, R_1	cust_1-cust_3	$\emptyset$	F
S_2	M_2	R_7	$\emptyset$	cust_5	F
		R_8	cust_6	cust_5	F

Table 3.3: Running example for DMine, round 2

(5) Given these, coordinator S_c assembles the messages and computes $\mathsf{conf}(R_7)$=0.6, $\mathsf{conf}(R_8)$=0.2 and $\mathsf{diff}(R_7,R_8)$=1. DMine computes $F'(R_7,R_8) = 0.5 * \frac{0.8}{5}$+$1 * 1$=1.08 $> F'(R_5,R_6)$=0.92. Hence, it replaces (R_5,R_6) with (R_7,R_8) and updates L_k to be $\{R_7,R_8\}$. As R_7 and R_8 are marked as "not extendable" at radius 2 (since d=2), DMine returns $\{R_7,R_8\}$ as top-2 diversified GPARs, in total 2 rounds. □

Message reduction. By maintaining additional information, DMine reduces the sizes of Σ, M and M_i. The idea is to test whether an upper bound of marginal benefit for any GPAR pairs can improve the minimum F'-value of L_k.

In each round r, incDiv filters non-promising GPARs from Σ and ΔE that cannot make top-k even after new GPARs are discovered. It keeps track of (1) a value F'_m=$\min F'(R_1,R_2)$ for all pairs (R_1,R_2) in L_k, (2) for each GPAR R_j in ΔE, an estimated maximum confidence $\mathsf{Uconf}^+(R_j,G)$ for all the possible GPARs extended from R_j, and (3) $\mathsf{conf}(R,G)$ for each GPAR R in Σ. Here $\mathsf{Uconf}^+(R_j,G)$ is estimated as follows. (a) Each S_i computes $\mathsf{Usupp}_i(R_j,F_i)$ as the number of matches of x in $R_j(x,F_i)$ that connect to a center node in F_i at hop $r+1$ $(r \leq d-1)$. (b) Then $\mathsf{Uconf}^+(R_j)$ is assembled at S_c as $\frac{\sum \mathsf{Usupp}_i(R_j,F_i)\mathsf{supp}(\bar{q},G)}{1 * \mathsf{supp}(q,G)}$. Denote the maximum $\mathsf{Uconf}^+(R_j,G)$ for $R_j \in \Delta E$ as $\max \mathsf{Uconf}^+(\Delta E)$, and the maximum $\mathsf{conf}(R,G)$ for $R \in \Sigma$ as $\max \mathsf{conf}(\Sigma)$. Then incDiv reduces Σ and M based on the reduction rules below.

Lemma 5: ***[Reduction rules]***: *(1) A* GPAR $R \in \Sigma$ *cannot contribute to* L_k *if* $\frac{1-\lambda}{N(k-1)}(\mathsf{conf}(R,G) + \max \mathsf{Uconf}^+(\Delta E)) + \frac{2\lambda}{k-1} \leq F'_m$. *(2) Extending a* GPAR $R_j \in \Delta E$ *does not contribute to* L_k *if either (a)* R_j *is not extendable, or (b)* $\frac{1-\lambda}{N(k-1)}(\mathsf{Uconf}^+(R_j,G) + \max \mathsf{conf}(\Sigma)) + \frac{2\lambda}{k-1} \leq F'_m$.

□

For the correctness of the rules, observe the following. (1) For each $R \in \Sigma$, $\mathsf{conf}(R) + \max \mathsf{Uconf}^+(\Delta E) + 1$ is an upper bound for its maximum possible increment to the F'-value of L_k; similarly for any R_j from ΔE. (2) If GPAR R does not contribute to L_k, then any GPARs extended from R do not contribute to L_k. Indeed, (a) upper bounds $\mathsf{Uconf}(R)$, $\mathsf{Usupp}_i(R)$, and $\mathsf{Uconf}^+(R)$ are *anti-monotonic* with any R' expanded of R, and (b) $\max \mathsf{Uconf}^+(\Delta E)$ and $\max \mathsf{conf}(\Sigma)$ are *monotonically decreasing*, while F'_m is *monotonically increasing* with the increase of rounds. Hence R can be safely removed from Σ, ΔE or M_i. Note that the removal of GPARs from Σ benefit the reduction of ΔE with smaller $\max \mathsf{conf}(\Sigma)$), and vice versa. DMine repeatedly applies the rules until no GPARs can be reduced from Σ and ΔE.

Automorphism checking. To reduce redundant GPARs, DMine checks whether GPARs in ΔE are automorphic at coordinator S_c (line 6) and locally at each S_i (localMine). It is costly to conduct pairwise automorphism tests on all GPARs in ΔE, since it is equivalent to graph isomorphism.

To reduce the cost, we use *bisimulation* [DPP01]. A graph pattern P_{R_1} is *bisimilar* to P_{R_2} if there exists a binary relation O_b on nodes of P_{R_1} and P_{R_2} such that (a) for all nodes u_1 in P_{R_1}, there exists a node u_2 in P_{R_2} with the same label such that $(u_1, u_2) \in O_b$, and vice versa for all nodes in P_{R_2}; and (b) for all edges (u_1, u_1') in P_{R_1}, there exists an edge (u_2, u_2') in P_{R_2} with the same label such that $(u_1', u_2') \in O_b$; and vice versa for all edges in P_{R_2}. The connection between bisimulation and automorphism is stated as follows.

Lemma 6: *If graph pattern P_{R_1} is not bisimilar to P_{R_2}, then R_1 is not an automorphism of R_2,* □

Hence, for a pair R_1 and R_2 of GPARs, DMine first checks whether P_{R_1} is bisimilar to P_{R_2}. It checks automorphism between R_1 and R_2 *only if* so. It takes $O(|\Delta E|^2)$ time to check pairwise bisimilarity O_b for all GPARs in ΔE [DPP01]. Moreover, O_b can be incrementally maintained when new GPARs are added [Sah07]. These allow us to use efficient (incremental) bisimulation tests instead of automorphism tests.

Trivial GPARs. DMine detects trivial GPARs $R(x, y)$: $Q(x, y) \Rightarrow q(x, y)$ at S_c as follows: (1) if $\mathsf{supp}(q, G)$ is 0, it returns $\emptyset$ to indicate that no interesting GPARs exist; and (2) if an extension leads to $\mathsf{supp}(Q\bar{q}) = 0$, *i.e.,* no match in $Q(x, G)$ violates $q(x, y)$, S_c removes R from ΔE and Σ.

Analyses. DMine returns a set L_k of k diversified GPARs with approximation ratio 2 (line 12), for the following reasons. (1) Parallel generation of GPARs finds all candidate GPARs within radius d. This is due to the *data locality* of subgraph isomorphism: for any node v_x in G, $v_x \in P_R(x, G)$ iff $v_x \in P_R(x, G_d(v_x))$ for any GPAR R of radius at most d at x. That is, we can decide whether v_x matches x via R by checking the d-neighbor of v_x locally at a fragment F_i. (2) Procedure incDiv updates L_k following the greedy strategy of [GS09], with approximation ratio 2. This is verified by approximation-preserving reduction to the max-sum dispersion problem, which maximizes the sum of pairwise distance for a set of data points and has approximation ratio 2 [GS09]. The reduction maps each GPAR to a data point, and sets the distance between two GPARs R and R' as $F'(R, R')$.

For time complexity, observe that in each round, the cost consists of (a) local parallel generation time T_1 of candidate GPARs, determined by $|F_i|$, M and M_i; and (b) total assembling and incremental maintenance cost T_2 of L_k at S_c, dominated by $|\Sigma|$, k and $|M_i|$. The cost of message reduction (by applying Lemma 5) takes in total $O(d|\Sigma|)$ time, where in each round, it takes a linear scan of ΔE and Σ to identify redundant GPARs. Note that $\sum_{i \in [1,n-1]} |M_i| \leq |\Delta E| \leq |\Sigma|$, $|M| \leq |\Sigma|$, and $|F_i|$ is roughly $|G|/n$ by our partitioning strategy. Hence T_1 and T_2 are functions of $|G|/n$, k and $|\Sigma|$.

This completes the proof of Theorem 4.

Remarks. Algorithm DMine can be easily adapted to the following two cases. (1) When a set of predicates instead of a single $q(x,y)$ is given, it groups the predicates and iteratively mines GPARs for each distinct $q(x,y)$. (2) When no specific $q(x,y)$ is given, it first collects a set of predicates of interests (*e.g.,* most frequent edges, or with user specified label q), and then mines GPARs for the predicate set as in (1).

3.4 Identifying Customers

We study how to identify potential customers with GPARs.

3.4.1 The Entity Identification Problem

Consider a set Σ of GPARs *pertaining to the same* $q(x,y)$, *i.e.,* their consequents are the same event $q(x,y)$. We define *the set of entities* identified by Σ in a (social) graph G with confidence η, denoted by $\Sigma(x,G,\eta)$, as follows:

$$\{v_x \mid v_x \in Q(x,G), Q(x,y) \Rightarrow q(x,y) \in \Sigma, \mathsf{conf}(R,G) \geq \eta\}$$

Problem. We study the *entity identification problem* (EIP):

- *Input*: A set Σ of GPARs pertaining to the same $q(x,y)$, a confidence bound $\eta > 0$, and a graph G.
- *Output*: $\Sigma(x,G,\eta)$.

It is to find potential customers x of y in G identified by *at least one* GPAR in Σ, with confidence of at least η.

Intractability. The decision problem of EIP is to determine, given Σ, G and η, whether $\Sigma(x,G,\eta) \neq \emptyset$. It is equivalent to decide whether there exists a GPAR $R \in \Sigma$ such that $\mathsf{conf}(R,G) \geq \eta$. The problem is nontrivial, as it embeds the subgraph isomorphism problem, which is NP-hard.

Proposition 7: *The decision problem for* EIP *is* NP*-hard, even when* Σ *consists of a single* GPAR. □

A naive way to compute $\Sigma(x,G,\eta)$ is as follows. For each $R(x,y): Q(x,y) \Rightarrow q(x,y)$ in Σ, (a) enumerate all matches of $Q\bar{q}$ and P_R in G by using an algorithm for subgraph isomorphism, *e.g.,* VF2 [CFSV04]; (b) compute $\mathsf{supp}(q,G)$ and $\mathsf{supp}(\bar{q},G)$ once in G; then based on the findings, (c) identify those R with $\mathsf{conf}(R,G) \geq \eta$, and return matches of x by these GPARs. This is cost-prohibitive (*e.g.,* takes $O(|G|!|G||\Sigma|)$ time using VF2 [CFSV04]) in real-life social graphs G, which often have billions of nodes and edges [UKBM11]. It is thus not practical to simply apply graph pattern matching algorithms to EIP over large G.

One might think that parallelization would solve the problem. However, parallelization is *not always effective*.

Parallel scalability. To characterize the effectiveness of parallelization, we formalize parallel scalability following [KRS88]. Consider a problem A posed on a graph G. We denote by $t(|A|,|G|)$ the worst-case running time of a *sequential algorithm* for solving A on G. For a parallel algorithm, we denote by $T(|A|,|G|,n)$ the time taken by the algorithm for solving A on G by using n processors. Here we assume $n \ll |G|$, *i.e.,* the number of processors does not exceed the size of the graph; this typically holds in practice since G has billions of nodes and edges, much larger than n.

We say that the algorithm is *parallel scalable* if

$$T(|A|,|G|,n) = O(t(|A|,|G|)/n) + (n|A|)^{O(1)}.$$

That is, the parallel algorithm achieves a polynomial reduction in sequential running time, plus a "bookkeeping" cost $O((n|A|)^l)$ for a constant l that is *independent of* $|G|$.

Obviously, if the algorithm is parallel scalable, then for a given G, it *guarantees* that the more processors are used, the less time it takes to solve A on G. It allows us to process big graphs by adding processors when needed. If an algorithm is not parallel scalable, we may not get reasonable response time *no matter how many* processors are used.

We say that problem A is *parallel scalable* if there exists a parallel scalable algorithm for it. Unfortunately, parallel scalability is *not* warranted for all problems, *e.g.,* it is beyond reach for graph simulation [FWWD14]. The good news is as follows.

Theorem 8: EIP *is parallel scalable.* □

As a proof, we outline a parallel algorithm for EIP, denoted by Match_c. Given Σ, $G = (V,E,L)$, η and a positive integer n, it computes $\Sigma(x,G,\eta)$ by using n processors. Note that Match_c is *exact*: it computes precisely $\Sigma(x,G,\eta)$.

To present Match_c, we use the following notations. (a) We use d to denote *the maximum radius* of $R(x,y)$ at node x, for all GPARs R in Σ. (b) For a node $v_x \in V$, $G_d(v_x)$ is *the d-neighbor* of v_x in G (see Section 3.3.2). (c) We denote by L the set of all *candidates* v_x of x, *i.e.,* nodes in G that satisfy the search condition of x in $q(x,y)$.

Algorithm. Match_c capitalizes on the data locality of subgraph isomorphism (see Section 3.3.2). It works as follows.

(1) *Partitioning*. It divides G into n fragments $\mathcal{F} = (F_1,\ldots,F_n)$ in the same way as algorithm DMine (Section 3.3.2), such that F_i's have roughly even size, and $G_d(v_x)$ is contained in one F_i for each $v_x \in L$. This is done in parallel. In particular, $G_d(v_x)$ can

be constructed in parallel by revising BFS (breadth-first search), within d hopes from v_x. Each fragment F_i is assigned to a processor S_i for $i \in [1,n]$.

(2) *Matching*. All processors S_i compute local matches in F_i *in parallel*. For each candidate $v_x \in L$ that resides in F_i, and for each GPAR $R(x,y):\ Q(x,y) \Rightarrow q(x,y)$ in Σ, S_i checks whether v_x is in $P_R(x, G_d(v_x))$, $P_Q(x, G_d(v_x))$ and $P_q(x, G_d(v_x))$, and whether v_x has an outlink labeled q.

(3) *Assembling*. Compute $\mathsf{conf}(R,G)$ for each R in Σ by assembling the partial results of (2) above. This is also done *in parallel*: first partition L into n fragments; then each processor operates on a fragment and computes partial support. These partial results are then collected to compute $\mathsf{conf}(R,G)$. Finally, output those v_x when there exists a GPAR R such that $v_x \in P_R(x,G)$ and $\mathsf{conf}(R,G) \geq \eta$.

Analysis. To show that Match_c is parallel scalable, observe the following. (1) Step 1 is in $O(|L||G_d^m|/n)$ time, since BFS is in $O(|G_d^m|)$ time, where G_d^m is the largest d-neighbor for all $v_x \in L$. (2) Step 2 takes $O(t(|G_d^m|, |\Sigma|)|L|/n)$ time, where $t(|G_d^m|, |\Sigma|)$ is the worst-case sequential time for processing a candidate v_x. (3) Step 3 takes $O(|L||\Sigma|/n)$ time. (4) By $|L| \leq |V|$, steps 1 and 2 take much less time than $t(|G|, |\Sigma|)$, since $t(,)$ is an exponential function by Proposition 7, unless P = NP. (5) In practice, $t(|G_d^m|, |\Sigma|)|L| \ll t(|G|, |\Sigma|)$ since $t(,)$ is exponential and G_d^m is much smaller than G. Indeed, (a) in the real world, graph patterns in GPARs are typically small, and hence so is the radius d; as argued in Section 3.3.2, $G_d(v_x)$ is thus often small.

Putting these together, we have that the parallel cost $T(|G|, |\Sigma|, n) < O(t(|G|, |\Sigma|)/n)$, and better still, the larger n is, the smaller $T(|G|, |\Sigma|, n)$ is.

Remark. Algorithm DMine (Section 3.3.2) takes $t(|A|/n, k)$ time and is parallel scalable if the problem size $|A|$ is measured as $|G| + |Q| + |\Sigma|$ [KS11]. Indeed, if one wants all candidate GPARs R with $\mathsf{supp}(R,G) \geq \sigma$, then $|\Sigma|$ is the size of the output, and $|\Sigma|$ is not large (due to small d and large σ).

3.4.2 Optimization Strategies

Algorithm Match_c just aims to show the parallel scalability of EIP. Its cost is dominated by step 2 for matching via subgraph isomorphism. To reduce the cost, we develop algorithm Match that improves Match_c by incorporating the following optimization techniques. To simplify the discussion, we start with a single GPAR $R(x,y):\ Q(x,y) \Rightarrow q(x,y)$.

Early termination. For each candidate $v_x \in L$ that resides in fragment F_i, we check whether *there exists* a match G_x of P_R in which v_x matches x. As soon as one G_x is verified a match of P_R, we include v_x in $P_R(x, F_i)$, *without* enumerating *all* matches of P_R at v_x. This is done locally at F_i: by our partitioning strategy, $G_d(v_x)$ is contained in F_i.

Guided search. To identify G_x at v_x, Match starts with pair (x, v_x) as a partial match m, and iteratively grows m with new pairs (u, v) for $u \in P_R$ and $v \in G_d(v_x)$ until a complete match is identified, *i.e., m* covers all the nodes in P_R. A complete m induces a subgraph G_x. It is in PTIME to verify whether m is an isomorphism from P_R to G_x.

To grow m, Match performs *guided search* based on *k-hop neighborhood sketch*. For each node v in G, a k-hop sketch $K(v)$ is a list $\{(1, D_1), \ldots, (k, D_k)\}$, where D_i denotes the distribution of the node labels and their frequency at i hop of v. Given a pair (u, v) newly added to m and a pattern edge (u, u') in Q, Match picks "the best neighbor" v' of v such that the pair (u', v') has a high possibility to make a match. This is decided by assigning a score $f(u', v')$ as $\sum_{i \in [1,k]}(D_i - D'_i)$, where $D'_i \in K(u')$, $D_i \in K(v')$, and $D_i - D'_i$ is the total frequency difference for each label in D_i. Indeed, (1) v' does not match u' if for some i, $D_i - D'_i < 0$; and (2) the larger the difference is, the more likely v' matches u'. If (u', v') does not lead to a complete m, Match backtracks and picks v'' with the next best score $r(u', v'')$.

Example 13: Consider GPAR R_1 of Fig. 3.1. For its designated node x, the 2-hop neighborhood sketch $L_2(x)$ in P_{R1} contains pair (1, D_1={(city,1), (cust,1), (French Restaurant, 4)}) and (2, D_2={(city,1),(cust,1),(French Restaurant, 4)}).

Given R_1 and G_1 of Fig. 3.2, Match identifies $P_{R_1}(x, G_1)$ as follows. (1) It finds $P_{q_1}(x, G)$={cust$_1$-cust$_4$, cust$_6$}, while cust$_5$ accounts for supp$(\bar{q}_1, G_1)$. (2) It computes $P_{R_1}(x, G_1)$ by verifying candidates v_x from $P_q(x, G_1)$, and calculates $f(x, v_x)$ in G_1, *e.g.,* L_2(cust$_2$) = {(1, D_1 = {(city, 1), (cust, 2), (French Restaurant, 8)}), (2, D_2={(city, 1), (cust, 2), (French Restaurant, 8)})}. Hence f(x, cust$_2$) = 5 + 5 = 10.

Match then ranks candidates cust$_2$, cust$_1$, cust$_3$, cust$_4$, where cust$_6$ is filtered due to mismatched sketches. (2) At cust$_2$, Match starts from $(x$, cust$_2)$, and extends to $(x'$, cust$_3)$ since $f(x'$, cust$_3)$ is the highest.

It continues to add pairs (city, NewYork), (French Restaurant, LeBernardin) and three pairs for French Restaurant[3]. This completes the match, and cust$_2$ is verified a match.

(3) Similarly, Match verifies cust$_1$ and cust$_3$, and finds $P_{R_1}(x, G_1)$ = {cust$_1$, cust$_2$, cust$_3$}.

Given $P_{R_1}(x, G_1)$, Match only needs to verify cust_5 for Q_1 in R_1; it finds $Q_1(x, G_1)$ = $P_{R_1}(x, G_1) \cup \{\text{cust}_5\}$.

It also finds $\text{supp}(q, G_1) = 5$ (cust_1–cust_4, cust_6), $\text{supp}(\bar{q}, G_1) = 1$ (cust_5), and computes $\text{conf}(R_1) = \frac{3*1}{1*5} = 0.6$. □

Algorithm Match. Given a set Σ of GPARs, Match revises step (2) of Match_c by checking whether v_x matches x via guided search and early termination; it reduces redundant computation for multiple GPARs by extracting common sub-patterns of GPARs in Σ [LKDL12]. It remains parallel scalable following the same complexity analysis for Match_c.

3.5 Experimental Study

Using real-life and synthetic graphs, we conducted three sets of experiments to evaluate (1) the scalability of algorithm DMine, (2) the effectiveness of DMine for discovering interesting GPARs, and (3) the scalability of algorithm Match for identifying potential customers in large graphs.

Experimental setting. We used two real-life graphs: (a) *Pokec* [Pok], a social network with 1.63 million nodes of 269 different types, and 30.6 million edges of 11 types, such as *follow*, *like*; and (b) *Google+* [Gon12], a social graph with 4 million entities of 5 types and 53.5 million links of 5 types.

We also designed a generator for synthetic graphs $G = (V, E, L)$, controlled by the numbers of nodes $|V|$ (up to 50 million) and edges $|E|$ (up to 100 million), with L drawn from an alphabet $\mathcal{L}$ of 100 labels.

Pattern generator. To evaluate Match, we generated GPARs R controlled by the numbers $|V_p|$ and $|E_p|$ of nodes and edges in P_R, respectively. (1) We found 48 meaningful GPARs on each of *Pokec* and *Google+*, with labels drawn from their data (domain, social groups).

(2) For synthetic graphs, we also generated 24 GPARs with labels drawn from $\mathcal{L}$. We denote the size of a GPAR R as $|R|= (|V_p|, |E_p|)$.

Algorithms. We implemented the following, all on GRAPE. (1) Algorithm DMine, compared with (a) DMine_{no}, its counterpart without optimization (incremental, reductions and bisimilarity checking), and (b) GRAMI [EASK14], an open source frequent subgraph mining tool. Since GRAMI uses *a single machine* [EASK14], we only compared the interestingness of patterns found by GRAMI with GPARs discovered by DMine. (2) Algorithm Match, compared with (a) Match_c (Section 3.4.1), (b) disVF2, a parallel implementation of VF2 for EIP, and (c) Match_s, Match by using the method of [RW15] instead of VF2.

Fragmentation and distribution. We revised the algorithm of [RPG+13] to evenly partition graph G into n fragments (see Section 3.3.2). We find that the gap between maximum and minimum time spent on different fragments by DMine is at most 14.4% (resp. 8.8%) of the time for processing fragments of *Pokec* (resp. *Google+*), and at most 6.0% (resp. 5.2%) of the time for identifying matches by Match. These indicate that the impact of skew from partitioning is fairly small.

We deployed the algorithms and n fragments on $n \in [4, 20]$. Each experiment was

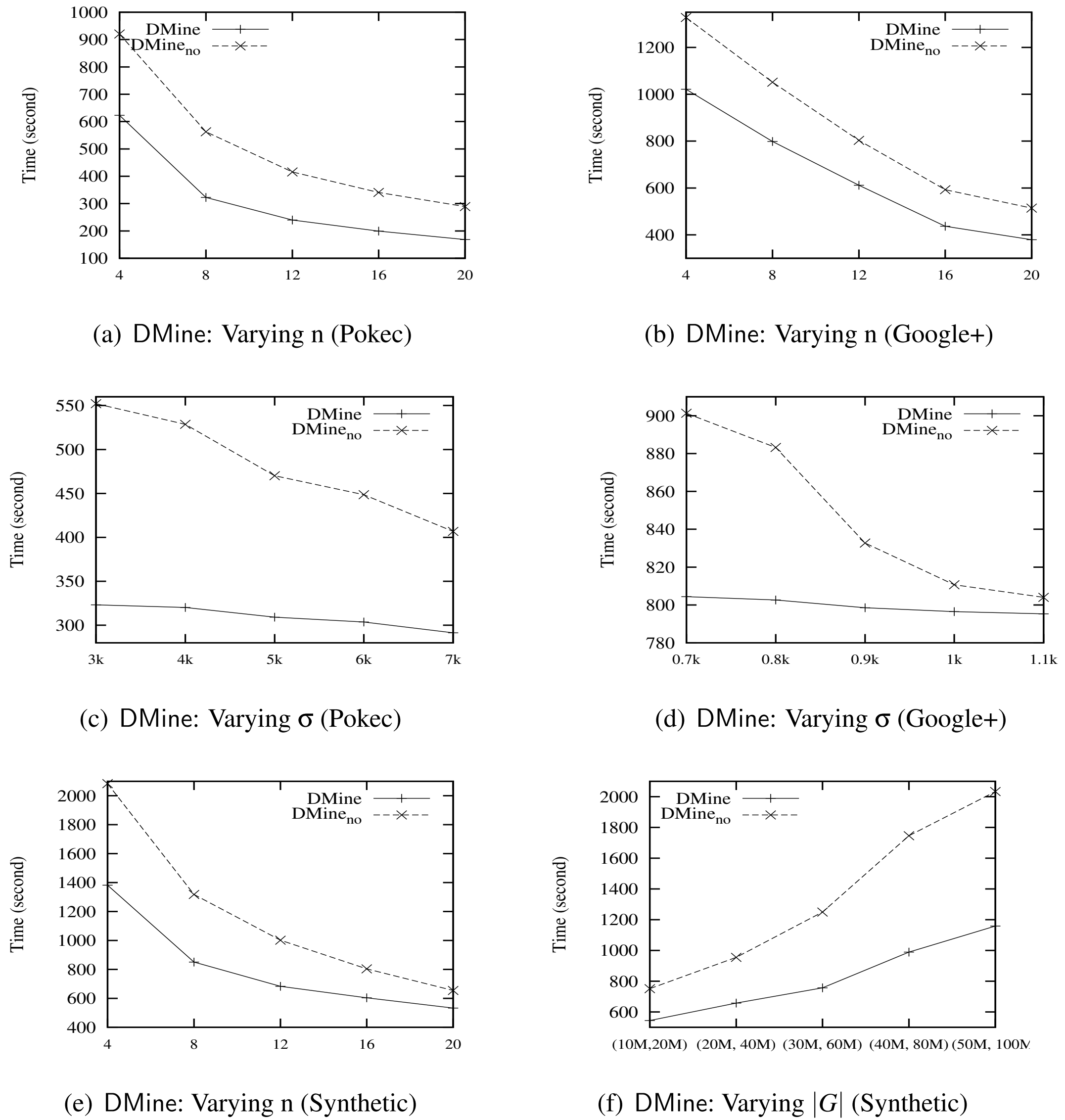


Figure 3.5: Parallel scalability of DMine

run 5 times and the average is reported here.

Experimental results. We next report our findings. We fixed parameter $\lambda = 0.5$ for diversification in Exp-1.

Exp-1: Scalability of DMine. We first evaluated the scalability of DMine vs. DMine$_{no}$. We used $k = 10$, and found that different k had little impact.

We found that GPARs mined in real-life graphs with infrequent edge labels usually denote unrelated facts. Hence we used 20 most frequent edge patterns, *i.e.,* graph patterns consisting of a single edge (with both node and edge labels), to grow GPARs in *Pokec*. We used all 5 types of edges in *Google+*.

Varying n. Fixing radius d = 2 and support σ = 5000 (500 for *Google+*), we varied the number n of processors from 4 to 20. The algorithms generated up to 300 patterns to be verified. As shown in Fig. 3.5(a) (resp. Fig. 3.5(b)), (1) DMine scales well with the increase of processors: the improvement is 3.7 (resp. 2.69) times when n increases from 4 to 20; and (2) it is on average 1.67 (1.37) times faster than $\mathsf{DMine}_{\mathsf{no}}$; this verifies that our optimization strategies effectively reduce confidence checking time, which is a major bottleneck in $\mathsf{DMine}_{\mathsf{no}}$. With 20 processors, DMine takes 168.3 (resp. 379) seconds on *Pokec* (resp. *Google+*).

Varying σ. Fixing $d = 2$ and $n = 4$, we varied σ from $3K$ to $7K$ (resp. 700 to 1100) on *Pokec* (resp. *Google+*). Figures 3.5(c) and 3.5(d) tell us the following. (1) All algorithms takes longer with smaller σ, because more patterns satisfy the support constraint and are checked. (2) DMine outperforms $\mathsf{DMine}_{\mathsf{no}}$ in all cases. Moreover, it is less sensitive to the increment of σ. This is because DMine checks much less patterns than $\mathsf{DMine}_{\mathsf{no}}$ due to its filtering strategy.

Using large synthetic graphs of size up to $(50M, 100M)$, we evaluated the impact of n, the size of G and radius d.

Varying n (Synthetic). Fixing $|G|$ = (10M, 20M), d = 2 and σ = 100, we varied n from 4 to 20. The results (Fig. 3.5(e)) are consistent with Figures 3.5(a) and 3.5(b). DMine takes 533.2 seconds over synthetic G with 20 processors.

Varying $|G|$ (Synthetic). Fixing $n = 16$, $d = 2$ and $\sigma = 100$, we varied $|G|$ from $(10M, 20M)$ to $(50M, 100M)$. As shown in Fig. 3.5(f), (1) both algorithms take longer on larger graphs; and (2) DMine outperforms $\mathsf{DMine}_{\mathsf{no}}$ by 1.76 times, verifying the effectiveness of our optimization methods.

Varying d. Fixing n = 16, $|G|$ = $(50M, 100M)$ and σ = 100, we varied d from 1 to 3. We find that DMine and $\mathsf{DMine}_{\mathsf{no}}$ take longer over larger d (not shown), as expected. However, DMine is less sensitive to d, since its optimization strategies reduces GPAR candidates and checking time.

Exp-2: Effectiveness of DMine. We manually examined GPARs discovered by DMine from *Pokec* and *Google+*. Three GPARs are shown in Fig. 4.11, with support above 100:

(1) R_9 (*Pokec*): *if x follows* user_1*,* user_1 *follows* user_2*,* user_2 *follows x,* user_1 *and* user_2 *share the hobby to* listen to music*, x and* user_1 *share the hobby of* party*, and if* user_2 *likes* Disco *music, then x likes* Disco. This suggests regularity between types of music

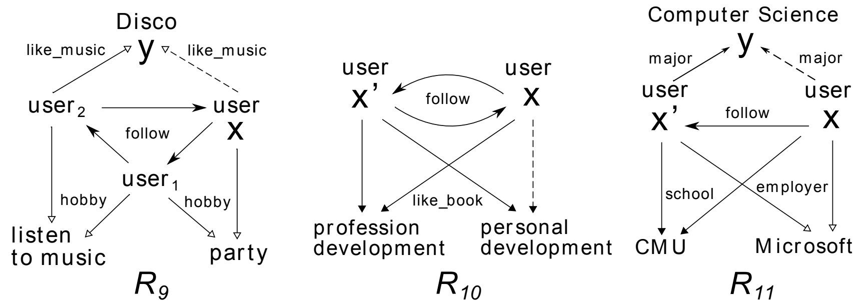


Figure 3.6: Effectiveness of DMine

people like and their friends' hobbies.

(2) R_{10} (*Pokec*): *if x and x′ follow each other and both like books of* profession development*, and if x′ likes books about* personal development*, then so does x.* This suggests that potential customers x favor books liked by their friends.

(3) R_{11} (*Google+*): *if x follows x′, both x and x′ went to* CMU*, both x and x′ are employees of* Microsoft*, and if x′ was majored in* CS*, then x was also likely majored in* CS. This indicates a social pattern between Microsoft employees and CMU computer science students.

We also found that most patterns mined by GRAMI are cycles of users. These patterns, although quite frequent, reveals little insight about entity associations.

GPARs *with different metrics*. We also evaluated different confidence metrics for GPARs (Section 3.2). Given a GPAR R, we define its (1) PCA confidence [GTHS13] $\mathsf{PCAconf}(R,G)$ as $\frac{\mathsf{supp}(R,G)}{\mathsf{supp}(Q\bar{q},G)}$, and (2) image-based $\mathsf{Iconf}(R,G)$ by replacing $\mathsf{supp}(\cdot,G)$ in $\mathsf{conf}(R,G)$ with the image-based support [BN08].

We evaluated prediction precision of these metrics for social networks following [GTHS13]. We partitioned *Pokec* into two fragments F_1 (as training data) and F_2 (for cross validation), and selected 5 predicates as in Exp-1 from F_1.

We set λ = 0 to focus on the relevance of GPARs, and mined top 10, 30 and 60 GPARs from F_1 with highest conf, PCAconf and Iconf, respectively. We evaluate the precision for each GPAR R as $\mathsf{prec}(R)=\frac{\mathsf{supp}(R,F_2)}{\mathsf{supp}(Q,F_2)}$, indicating correctly predicted customers in F_2, constrained by GPARs mined from F_1.

As shown in the table 3.4, (1) DMine is able to identify GPARs that "predict" predicates with average precision up to 42.3%, and (2) GPARs ranked by our conf metric provides better prediction precision than PCAconf and Iconf.

Exp-3: Scalability of Match. Finally, we evaluated (1) the scalability of Match with

	Top 10	Top 30	Top 60
PCAconf	0.276	0.280	0.277
Iconf	0.267	0.273	0.265
conf	0.423	0.388	0.381

Table 3.4: Prediction precision

the number n of processors, and the impacts of (2) the number $\|\Sigma\|$ of GPARs in Σ, (3) the maximum radius d of GPARs in Σ, and (4) the size $|G|$ of graphs. We started with real-life graphs and fixed $\eta = 1.5$.

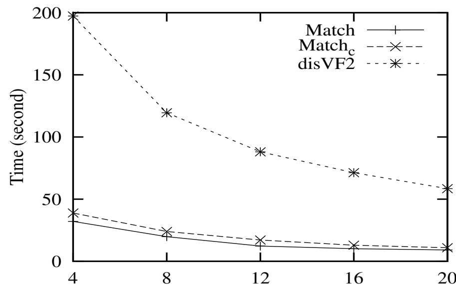
Varying n. Fixing $\|\Sigma\| = 24$, $|R| = (5,8)$ and $d = 2$, we varied n from 4 to 20. Figures 3.7(a) and 3.7(b) report the results on *Pokec* and *Google+*, respectively, which tell us the following.

(1) Match, Match_c and Match_s allow a high degree of parallelism. For instance, Match is 3.52 (resp. 3.54) times faster when n increases from 4 to 20 on *Pokec* (resp. *Google+*). This is consistent with Theorem 8. are efficient. In particular, Match takes 9.1 seconds on social graph *Pokec* with 20 processors, and it scales better than Match_c and disVF2. We find that Match_s and Match have very similar performance, and thus we report Match only.

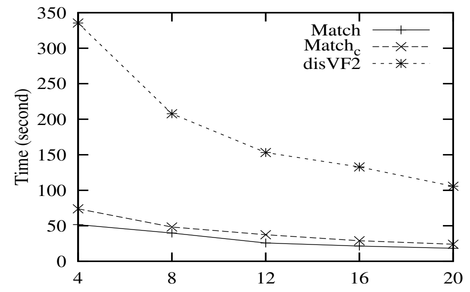
(2) Our optimization strategies are effective. (a) Compared to disVF2, Match_c and Match are 4.79 and 6.24 times faster on average, since for each GPAR $R : Q \Rightarrow q$, disVF2 invokes two isomorphic checks at each candidate v_x (one for P_R and one for $Q_{\bar{q}}$) vs. one by Match_c and Match; this justifies *the need for new algorithms for* EIP instead of applying conventional pattern matching algorithms. (b) Match is 1.2 and 1.35 times times faster than Match_c on *Pokec* and *Google+*, respectively, demonstrating the effectiveness of early termination and guided search, without enumerating all matches.

Varying $\|\Sigma\|$. Fixing $n = 8$ and $d = 2$, we varied $\|\Sigma\|$ from 8 to 48. As shown in Figures 3.7(c) and 3.7(d), (1) all algorithms take longer time with larger $\|\Sigma\|$, as expected; (2) Match is less sensitive to $\|\Sigma\|$ than Match_c and disVF2; (3) the improvement of Match over the others is greater on larger Σ. These are because optimization by early termination and guided search works better for more GPARs in Σ.

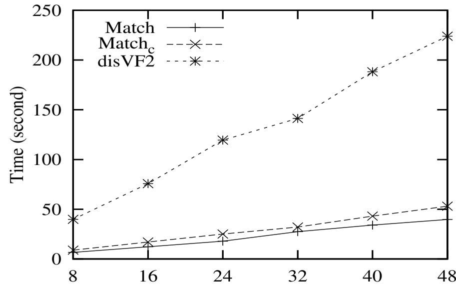
Varying d. Fixing $n = 8$ and $\|\Sigma\| = 20$, we varied d from 1 to 5. As shown in Figures 3.7(e) and 3.7(f) (in *logarithmic scale*), all algorithms take longer time with larger



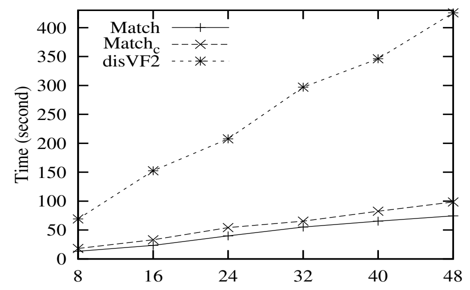
(a) Match: Varying n (Pokec)



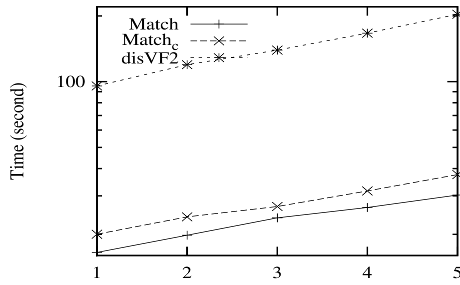
(b) Match: Varying n (Google+)



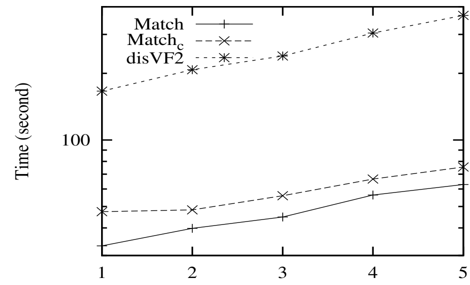
(c) Match: Varying $\|\Sigma\|$ (Pokec)



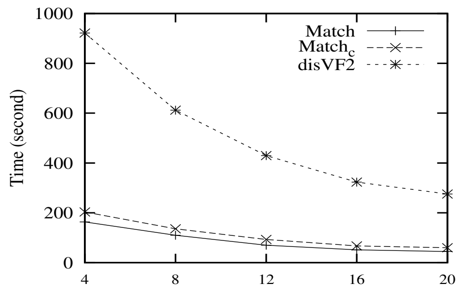
(d) Match: Varying $\|\Sigma\|$ (Google+)



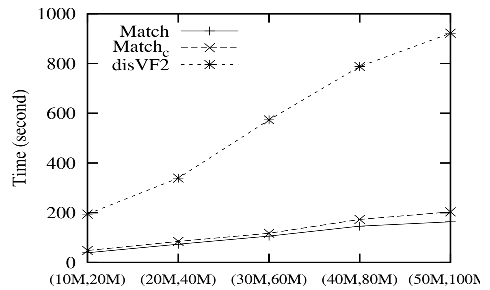
(e) Match: Varying d (Pokec)



(f) Match: Varying d (Google+)



(g) Match: Varying n (Synthetic)



(h) Match: Varying $|G|$ (Synthetic)

Figure 3.7: Performance evaluation of Match

d, since more nodes in the d-neighbors of candidates need to be visited. Nonetheless, Match and Match$_c$ are less sensitive to d than disVF2 due to their optimization

techniques (data locality leveraged by Match_c, and early termination by Match).

Synthetic graphs. Using larger synthetic graphs, we evaluated the impact of n. Fixing $|G| = (50M, 100M)$, $d = 2$, $\eta = 1.5$ and $\|\Sigma\| = 24$, we varied n from 4 to 20. As shown in Fig. 3.7(g), the result is consistent with its counterparts on real-life graphs (Figures. 3.7(a) and 3.7(b)). The improvement for Match is 3.65 times when n increases from 4 to 20.

Fixing $n = 4$, $\|\Sigma\| = 24$, $\eta = 1.5$ and $d = 2$, we varied $|G|$ from $(10M, 20M)$ to $(50M, 100M)$. As shown in Fig. 3.7(h), (1) all the algorithms take longer on larger $|G|$, as expected; (2) Match performs the best, and is less sensitive to $|G|$ than the others; and (3) despite Proposition 7, Match is reasonably efficient: when $|G| = (50M, 100M)$, Match takes 163 seconds with 4 processors, while disVF2 takes 922 seconds.

Summary. We find the following. (1) It is not very expensive to mine diversified top-k GPARs in large social networks. For instance, DMine takes 533.2 seconds on graphs with $|G| = $ (10M,20M) by using 20 processors, when $k = 10$, $\sigma = 100$ and $d = 2$. (2) The number of candidate GPARs is not very large (up to 300), and hence DMine is "parallel scalable" (Section 3.4.1): it is 3.2 times faster on average when n increases from 4 to 20, on real-world social networks. (3) Moreover, discovered GPARs based on our conf metric predict more precise potential customers in social networks than its PCA and image-based counterparts. (4) Match is parallel scalable: it is 3.53 times faster on average when n increases from 4 to 20 over real-life social networks. (5) It is practical to apply GPARs to large graphs: on graphs with $|G| = (50M, 100M)$ and a set Σ of 24 GPARs, Match takes less than 45 seconds with 20 processors. (6) Our optimization strategies are effective: DMine outperforms $\mathsf{DMine}_{\mathsf{no}}$ by 1.52 times, and Match is 1.27 and 6.24 times faster than Match_c and disVF2, respectively, on real-life graphs, on average.

3.6 Related Work

We categorize related work in this chapter as follows.

Association rules. Introduced in [AIS93], association rules are defined on relations of transaction data. Prior work on association rules for social networks [SHJS06] and RDF knowledge bases resorts to mining conventional rules and Horn rules (as conjunctive binary predicates) [GTHS13] over tuples with extracted attributes from social graphs, instead of exploiting graph patterns. While [BBBG09] studies time-dependent rules via graph patterns, it focuses on evolving graphs and hence adopts different semantics for support and confidence.

GPARs extend association rules from relations to graphs. (a) It demands topological support and confidence metrics. Moreover, incomplete information is common in social graphs [Don14, GTHS13] and has to be incorporated into the metrics. (b) GPARs are interpreted with isomorphic functions and hence, cannot be expressed as conjunctive queries, which do not support negation or inequality needed for functions. (c) Applying GPARs becomes an intractable problem of multi-pattern-query processing in big graphs. (d) Mining (diversified) GPARs is beyond rule mining from itemsets [ZZ02].

Graph pattern mining. There have been algorithms for pattern mining in graph databases [IWM00, HCD$^+$94] (see [JCZ13] for a survey). Large-scale mining techniques are also studied in a single graph [EASK14], notably top-k algorithms [KCY09, FVT12, XCYH06, SQC14]. To reduce the cost, scalable subgraph isomorphism algorithms, *e.g.,* [RW15], can be adopted to generate pattern candidates. Diversity of graph patterns is not studied there.

However, (a) pattern mining over graph databases [IWM00, KCY09] cannot be used to mine GPARs, as their anti-monotonic property does not hold in a single graph [JCZ13]. (b) While mining single graphs is based only on isomorphic counting [EASK14], DMP is bi-criteria optimization problem for confidence and diversity of GPARs, apart from [FVT12, XCYH06]. We are not aware of prior work on discovering diversified graph patterns.

Graph pattern matching. Several parallel algorithms have been developed for subgraph isomorphism, *e.g.,* [KLCL13, RW15, RvRH$^+$14], and for multi-pattern optimization, *e.g.,* [LKDL12, HVA14].

Our algorithms for EIP differ from the prior work in the following. (a) Instead of enumerating isomorphic matches, EIP identifies a potential customer *once* one match

is found, and moreover, computes its associated confidence. That is, EIP is beyond conventional subgraph isomorphism. (b) We provide parallel scalable algorithms for multi-pattern matching. To the best of our knowledge, these are among the first algorithms on big graphs that *guarantee a polynomial speedup over sequential algorithms* with the increase of processors [KRS88]. (c) We propose optimization strategies that are not studied by previous work. This said, prior optimization techniques can be incorporated into GPAR-based entity identification; *e.g.,* the methods of [LKDL12] to extract common sub-patterns.

3.7 Summary

In this chapter, we have proposed association rules with graph patterns (GPARs), from syntax, semantics to support and confidence metrics. We have studied DMP and EIP, for mining GPARs and for identifying potential customers with GPARs, respectively, from complexity to parallel (scalable) algorithms. Our experimental study has verified that while DMP and EIP are hard, it is feasible to discover and make practical use of GPARs. We contend that GPARs provide a promising tool for social media marketing, among other applications.

Chapter 4

Extending Pattern Matching on GRAPE with Quantifiers

We have shown that the GRAPE is capable of handling complicated applications such as association rules in Chapter 3. However, till now the graph pattern is typically defined in the same format of the graph, which lacks a rich expressiveness and limits the effectiveness of the applications such as GPARs.

In this chapter, we consider revision of graph pattern. By adding counting quantifiers, the graph patterns get more expressive power. Better still, this does not introduce extra complexity for its matching problem. We then extend GPARs with counting quantifiers to meet the need in the social marketing.

Given a graph pattern $Q(x_o)$ and a graph G, *graph pattern matching* is to find $Q(x_o, G)$, the set of matches of x_o in subgraphs of G that are isomorphic to Q. Here "query focus" x_o is a designated node of Q denoting search intent [BMC10]. Traditionally, pattern Q is modelled as a (small) graph in the same form as G. This notion of patterns is used in social group detection and transportation network analysis.

However, in applications such as social media marketing, knowledge discovery and cyber security, more expressive patterns are needed, notably ones with counting quantifiers.

Example 14: (1) Consider an association rule for specifying regularities between entities in social graphs:

- *If (a) person x_o is in a music club, and (b) among the people whom x_o follows, at least* 80% *of them like an album y, then the chances are that x_o will buy y.*

Its antecedent specifies conditions (a) and (b). If these two conditions hold, then we can recommend album y to x_o. This is an example of social media marketing, which is predicted to trump traditional marketing. Indeed, empirical studies suggest that "90% of customers trust peer recommendations versus 14% who trust advertising" [tru], and "the peer influence from one's friends causes more than 50% increases in odds of buying products" [BU12].

The antecedent is specified as a *quantified graph pattern* (QGP) $Q_1(x_o)$ shown in Fig. 4.1, where x_o is its query focus, indicating potential customers. Here edge follow(x_o, z) carries a *counting quantifier* "$\geq$ 80%", for condition (b) above. In a social graph G, a node v_x matches x_o in Q_1, *i.e.,* $v_x \in Q_1(x_o, G)$, if (a) there exists an isomorphism h from Q_1 to a subgraph G' of G such that $h(x_o) = v_x$, *i.e.,* G' satisfies the topological constraints of Q_1, and (b) among all the people whom v_x follows, 80% of them account for matches of z in $Q_1(G)$, satisfying the counting quantifier.

The following association rules are also found useful in social media marketing, with various counting quantifiers:

- *If for all the people z whom x_o follows, z recommends Redmi 2A (cell phone), then x_o may buy a Redmi 2A.*
- *If among the people followed by x_o, (a) at least p of them recommend Redmi 2A, and (b) no one gives Redmi 2A a bad rating, then x_o may buy Redmi 2A.*

The antecedents of these rules are depicted in Fig. 4.1 as QGPs $Q_2(x_o)$ and $Q_3(x_o)$, respectively. Here Q_2 uses a *universal* quantification (= 100%), while Q_3 carries numeric

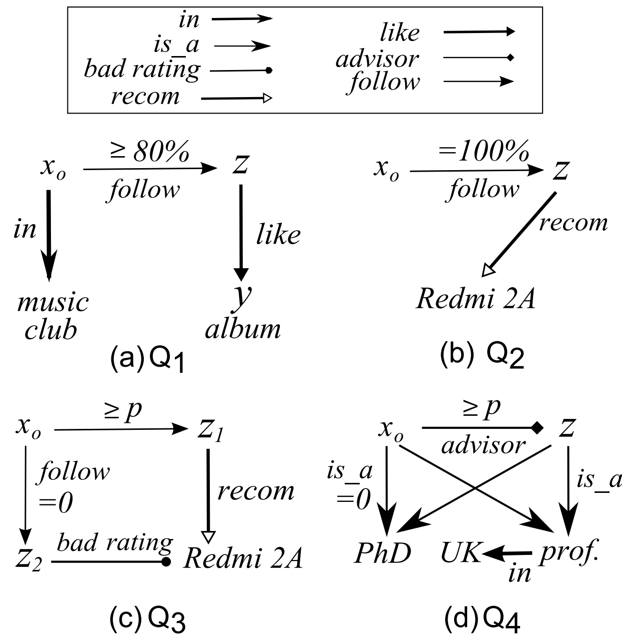


Figure 4.1: Quantified graph patterns

aggregate ($\geq p$) and *negation* (= 0). In particular, a node v_x in G matches x_o in Q_3 only if there exists *no* node v_w in G such that $\mathsf{follow}(v_x, v_w)$ is an edge in G and there exists an edge from v_w to Redmi 2A labeled "bad rating". That is, counting quantifier "$= 0$" on edge $\mathsf{follow}(x_o, z_2)$ enforces negation.

(2) Quantified graph patterns are also useful in knowledge discovery. For example, QGP $Q_4(x_o)$ of Fig. 4.1 is to find

- *all people who (a) are professors in the UK, (b) do not have a PhD degree, and (c) have at least p former PhD students who are professors in the UK.*

It carries negation ($= 0$) and numeric aggregate ($\geq p$). □

These counting quantifiers are not expressible in traditional graph patterns. Several questions about QGPs are open. (1) How should QGPs be defined, to balance their expressive power and complexity? (2) Can we efficiently conduct graph pattern matching with QGPs in real-life graphs, which may have trillions of nodes and edges [GBDS14]? (3) How can we make use of QGPs in emerging applications? The need for studying these is highlighted in, *e.g.,* social marketing, knowledge discovery and cyber security.

4.1 Quantified Graph Patterns

We next introduce quantified graph patterns QGPs. To define QGPs, we first review conventional graph patterns.

4.1.1 Conventional Graph Pattern Matching

We consider labeled, directed *graphs*, defined as $G = (V,E,L)$, 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 node v to v'; and (3) each node v in V (resp. edge e in E) carries $L(v)$ (resp. $L(e)$), indicating its label or content as commonly found in social networks and property graphs.

Two example graphs are depicted in Fig. 4.2.

We review two notions of subgraphs. (1) A graph $G' = (V',E',L')$ is a *subgraph of* $G = (V,E,L)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and for each edge $e \in E'$ (resp. node $v \in V'$), $L'(e) = L(e)$ (resp. $L'(v) = L(v)$). (2) We say that G' is *a subgraph induced by* a set V' of nodes if $G' \subseteq G$ and E' consists of all the edges in G whose endpoints are in V'.

Patterns. A *graph pattern* is traditionally defined as a graph $Q(x_o)$ = (V_Q, E_Q, L_Q), where (1) V_Q (resp. E_Q) is a set of pattern nodes (resp. edges), (2) L_Q is a function that assigns a node label $L_Q(u)$ (resp. edge label $L_Q(e)$) to each pattern node $u \in V_Q$ (resp. edge $e \in E_Q$), and (3) x_o is a node in V_Q, referred to as the *query focus* of Q, for search intent.

Pattern matching. A *match* of pattern Q in graph G is a *bijective function* h from nodes of Q to nodes of a subgraph G' of G, 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 if and only if $(h(u),h(u'))$ is an edge in G', and $L_Q(u,u') = L(h(u),h(u'))$. From h, subgraph G' can be readily deduced.

We denote by $Q(G)$ *the set of matches* of Q in G, *i.e.,* the set of bijective functions h that induce a match of Q in G. *Query answer* is the set of all matches of x_o in $Q(G)$.

Given $Q(x_o)$ and G, *graph pattern matching* is to compute $Q(x_o,G)$, *i.e.,* all matches of query focus x_o in G via Q.

4.1.2 Quantified Graph Patterns

We next define QGPs, by extending conventional graph patterns to express quantified search conditions.

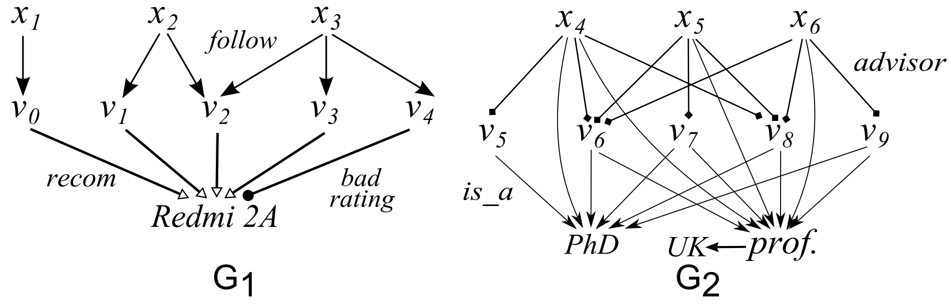


Figure 4.2: Examples of social graph

Syntax. A *quantified graph pattern* (QGP) $Q(x_o)$ is defined as (V_Q, E_Q, L_Q, f), where V_Q, E_Q, L_Q and x_o are the same as their traditional counterparts, and f is a function such that for each edge $e \in E_Q$, $f(e)$ is a predicate of

- a *positive form* $\sigma(e) \odot p\%$ for a real number $p \in (0, 100]$, or $\sigma(e) \odot p$ for a positive integer p, or
- $\sigma(e) = 0$, where e is referred to as a *negated edge*.

Here $\odot$ is either $=$ or $\geq$, and $\sigma(e)$ will be elaborated shortly. We refer to $f(e)$ as the *counting quantifier* of e, and $p\%$ and p as *ratio* and *numeric aggregate*, respectively.

Counting quantifiers express logic quantifiers as follows:

- *negation* when $f(e)$ is $\sigma(e) = 0$ (*e.g.,* Q_3 in Example 14);
- *existential quantification* if $f(e)$ is $\sigma(e) \geq 1$; and
- *universal quantifier* if $f(e)$ is $\sigma(e) = 100\%$ (*e.g.,* Q_2).

A conventional pattern Q is a special case of QGP when $f(e)$ is $\sigma(e) \geq 1$ for all edges e in Q, *i.e.,* if Q has existential quantification only. We leave out $f(e)$ if it is $\sigma(e) \geq 1$.

We call a QGP Q *positive* if it contains no negated edges (*i.e.,* edges e with $\sigma(e) = 0$), and *negative* otherwise.

Example 15: Graph patterns Q_1–Q_4 given in Example 14 are QGPs with various counting quantifiers, *e.g.,* (1) edge (x_o, z) in Q_1 has a quantifier $\sigma(x_o, z) \geq 80\%$; (2) Q_2 has a universal quantifier $\sigma(x_o, z)$=100% on edge (x_o, z), and an existential quantifier for edge $(z, \texttt{Redmi 2A})$; and (3) Q_3 has a negated edge (x_o, z_2) with $\sigma(x_o, z_2) = 0$. Among the QGPs, Q_1 and Q_2 are positive, while Q_3 and Q_4 are negative. □

Remark. To strike a balance between the expressive power and the complexity of pattern matching with QGPs in large-scale graphs, we assume a predefined *constant*

l such that on any simple path (*i.e.,* a path that contains no cycle) in $Q(x_o)$, (a) there exist at most l quantifiers that are not existential, and (b) there exist no more than one negated edge, *i.e.,* we exclude "double negation" from QGPs.

The reason for imposing the restriction is twofold. (1) Without the restriction, quantified patterns would be able to express first-order logic (FO) on graphs. Indeed, FO sentences $P_1X_1 \ldots P_lX_l\ \varphi$ can be encoded in such a pattern, where P_i is either $\forall$ or $\exists$, φ is a logic formula, and l is unbounded. Such patterns inherit the complexity of FO [Lib13], in addition to #P complication. Then even the problem for deciding whether there exists a graph that matches such a pattern is beyond reach in practice. As will be seen shortly, the restriction makes QGPs discovery and evaluation feasible in large-scale graphs. (2) Moreover, we find that QGPs with the restriction suffice to express quantified patterns commonly needed in real-life applications, for *small* l. Indeed, empirical study suggests that l is at most 2, and "double negation" is rare, as "99% of real-world queries are star-like" [GFMPdlF11]. One can extend $f(e)$ to support $>$, $\neq$ and $\leq$ as $\odot$, and conjunctions of predicates. To simplify the discussion, we focus on QGPs $Q(x_o)$ in the simple form given above.

Semantics. We next give the semantics of QGPs. We consider positive QGPs first, and then QGPs with negation.

Positive QGPs. We use the following notations. Striping all quantifiers $f(e)$ off from a QGP $Q(x_o)$, we obtain a conventional pattern, referred to as the *stratified pattern* of $Q(x_o)$ and denoted by $Q_\pi(x_o)$. Consider an edge $e = (u,u')$ in $Q(x_o)$, a graph G and nodes v_x and v in G. When x_o is mapped to v_x, we define *the set of children of v via e and Q*, denoted by $M_e(v_x,v,Q)$ when G is clear from the context:

$$\{v' \mid h \in Q_\pi(G), h(x_o) = v_x, h(e) = (v,v')\},$$

i.e., the set of children of v that match u' when u is mapped to v, subject to the constraints of Q_π. Abusing the notion of isomorphic mapping, $h(e) = (v,v')$ denotes $h(u) = h(v)$, $h(u') = h(v')$, $(v,v') \in G$ and $L_Q(u,u') = L(v,v')$.

We define $M_e(v) = \{v' \mid (v,v') \in G, L(v,v') = L_Q(e)\}$, the set of the children of v connected by an e edge.

For a positive QGP $Q(x_o)$, *a match* $h_0 \in Q(G)$ satisfies the following conditions: for *each* node u in Q and *each* edge $e = (u,u')$ in Q,

- if $f(e)$ is $\sigma(e) \odot p\%$, then $\frac{|M_e(h_0(x_o),h_0(u),Q)|}{|M_e(h_0(u))|} \odot p\%$, in terms of the ratio of the number of children of v via e and Q to the total number of children of v via e;

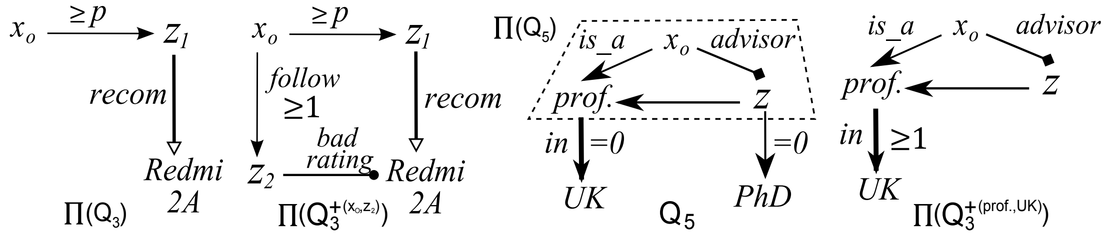


Figure 4.3: Negative QGPs

and

- if $f(e)$ is $\sigma(e) \odot p$, then $|M_e(h_0(x_o), h_0(u), Q)| \odot p$, in terms of the number of children of v via e and Q.

That is, $\sigma(e)$ is defined as ratio $\frac{|M_e(h_0(x_o),h_0(u),Q)|}{|M_e(h_0(u))|}$ or cardinality $|M_e(h_0(x_o), h_0(u), Q)|$, for $p\%$ or p, respectively. Intuitively, $\sigma(e)$ requires that at least $p\%$ of nodes or p nodes in $M_e(v)$ are matches for u' when v is mapped to u. A match in $Q(G)$ must satisfy the topological constraints of Q_π and moreover, the counting quantifiers of Q. Note that the counting quantifier on edge $e = (u, u')$ is applied at *each match* $h_0(u)$ of u, to enforce the semantics of counting.

We denote by $Q(u, G)$ the set of matches of a pattern node u, *i.e.,* nodes $v = h(u)$ induced by all matches h of Q in G. The *query answer* of $Q(x_o)$ in G is defined as $Q(x_o, G)$.

Example 16: For graph G_1 in Fig. 4.2 and QGP Q_2 of Example 14, $Q_2(x_o, G_1)$ = $\{x_1, x_2\}$. Indeed, 100% of the friends of x_1 and x_2 recommend Redmi 2A. More specifically, for pattern edge $e = \text{follow}(x_o, z)$, when x_o is mapped to x_1 via h_0, $M_e(h_0(x_o), x_1, Q)$ = $\{v_0\}$, which is the set $M_e(x_1)$ of all people whom x_1 follows; similarly when x_o is mapped to x_2. In contrast, while x_3 matches x_o via the stratified pattern of Q_2, $x_3 \notin Q_2(x_o, G_1)$ since at least one user whom x_3 follows (*i.e.,* v_4) has no recom edge to Redmi 2A. □

<u>*Negative* QGPs.</u> To cope with QGP $Q(x_o)$ with negated edges, we define the following: (1) $\Pi(Q)$: the QGP induced by those nodes in $Q(x_o)$ that are connected to x_o (via a path from or to x_o) with non-negated edges in $Q(x_o)$, *i.e.,* $\Pi(Q)$ excludes all those nodes connected via at least one negated edge; (2) Q^{+e}, obtained by "positifying" a negated edge e in Q, *i.e.,* by changing $f(e)$ from $\sigma(e) = 0$ to $\sigma(e) \geq 1$; and (3) E_Q^-, the set of all negated edges in Q.

Then in a graph G, *query answer* to $Q(x_o)$ is defined as

$$Q(x_o, G) = \Pi(Q)(x_o, G) \setminus \left(\bigcup_{e \in E_Q^-} \Pi(Q^{+e})(x_o, G)\right).$$

That is, we enforce negation via set difference. One can verify that for *each* node u in Q and *each* negated edge $e = (u, u')$ in Q, $|M_e(h_0(x_o), h_0(u), Q)| = 0$.

Example 17: Consider G_1 and Q_3 of Example 14 with p=2. Pattern $\Pi(Q_3)$, which excludes the negated edge e = (x_o, z_2) in Q_3, and $\Pi(Q_3^{+e})$, which"positifies" e in Q_3, are shown in Fig. 4.3. One can verify the following: (1) $\Pi(Q_3)(x_o, G_1)$ is $\{x_2, x_3\}$; note that x_1 is not a match since only 1 user whom x_1 follows recommends Redmi 2A, and hence violates the counting quantifier $\geq p$; and (2) $\Pi(Q_2^{+e})$ is $\{x_3\}$, which is a "negative" instance for Q_3. Hence, $Q_3(x_o, G_1)$ is $\{x_2\}$, where x_3 is excluded since he follows v_4 who gave a bad rating on Redmi 2A, *i.e.,* violating the negation $\sigma(e) = 0$.

Similarly, for QGP Q_4 and graph G_2 of Fig. 4.2, when p=2, $Q_4(x_o, G_2)$ is $\{x_5, x_6\}$. Note that node x_4 matches the stratified pattern of Q_4, but it violates the negation on (x_o, PhD), which requires that matches of x_o must not be a PhD.

As another example, consider $Q_5(x_o)$ with two negated edges e_1 = (Prof, UK) and e_2 = (z, PhD). It is to find non-UK professors who supervised students who are professors but have no PhD degree. As shown in Fig. 4.3, $\Pi(Q_5)$ finds professors who supervised students who are professors. In contrast, $\Pi(Q_5^{+e_1})$ finds such professors in the UK, and $\Pi(Q_5^{+e_2})$ (not shown) retrieves professors with students who are professors and have a PhD. In a graph G, $Q(x_o, G)$ = $\Pi(Q_5)(x_o, G) \setminus (\Pi(Q_5^{+e_1})(x_o, G) \cup \Pi(Q_5^{+e_2})(x_o, G))$.

□

The notations in this chapter are summarized in Table 4.1.

symbols	notations
$Q(x_o)$	QGP, defined as (V_Q, E_Q, L_Q, f)
$Q(x_o, G)$	query answer, the set of matches of x_o
$Q_\pi(x_o)$	stratified pattern of Q by removing quantifiers
$G' \subseteq G$	G' is a subgraph of G
$M_e(v_x, v, Q)$	$\{v' \mid h \in Q_\pi(G), h(x_o) = v_x, h(e) = (v, v')\}$, $e = (u, u')$
$M_e(v)$	$\{v' \mid (v, v') \in G, L(v, v') = L_Q(e)\}$
$\sigma(e) \odot p\%$	$\frac{\lvert M_e(h_0(x_o), h_0(u), Q)\rvert}{\lvert M_e(h_0(u))\rvert} \odot p\%$, $e = (u, u')$, $h_0 \in Q(G)$
$\sigma(e) \odot p$	$\lvert M_e(h_0(x_o), h_0(u), Q)\rvert \odot p$, $e = (u, u')$, $h_0 \in Q(G)$
$\Pi(Q)$	$Q(x_o)$ excluding nodes with negated edges
Q^{+e}	by positifying a negated edge e in Q
$R(x_o)$	QGAR $Q_1(x_o) \Rightarrow Q_2(x_o)$

Table 4.1: Notations in Chapter 4

4.2 The Complexity of Quantified Matching

In the next three sections, we study *quantified matching*:

- *Input*: A QGP $Q(x_o)$ and a graph G.
- *Output*: $Q(x_o, G)$,

to compute the set of all matches of query focus x_o of Q in G. The need for studying this is highlighted in, *e.g.,* social marketing, knowledge discovery and cyber security.

We establish the complexity of the problem in this section.

Decision problem. Its decision problem, referred to as *the quantified matching problem*, is stated as follows.

- *Input*: A QGP $Q(x_o)$, a graph G and a node v in G.
- *Question*: Is $v \in Q(x_o, G)$?

When $Q(x_o)$ is a conventional pattern, the problem is NP-complete. When it comes to QGPs, however, ratio aggregates $\sigma \odot p\%$ and negation $\sigma = 0$ increase the expressive power, and make the analysis more intriguing. To handle $\sigma \odot p\%$, for instance, a brute-force approach invokes an NP algorithm that calls a #P oracle to check the ratio aggregate.

We show that while the increased expressive power of QGPs comes with a price, their complexity bound does not get much higher. In particular, #P is not necessary.

Theorem 9: *The quantified matching problem remains* NP*-complete for positive* QGPs*, and it becomes* DP*-complete for (possibly negative)* QGPs. □

Here DP is the class of languages recognized by oracle machines that make a call to an NP oracle and a call to a coNP oracle. That is, L is in DP if there exist languages $L_1 \in$ NP and $L_2 \in$ coNP such that $L = L_1 \cap L_2$ [Pap03]

That is, adding positive quantifiers to conventional graph patterns does not increase the complexity, although ratio aggregates add extra expressive power. Note that such positive patterns alone are already useful in practice. In contrast, the presence of negation makes quantified matching harder, but it remains low in the polynomial hierarchy [Pap03].

The proof is nontrivial. Below we present lemmas needed, and provide detailed proofs.

The lower bounds follow from the stronger results below, which are in turn verified by reductions from SUBGRAPH ISOMORPHISM and EXACT-CLIQUE, which are NP-complete and DP-complete, respectively (cf. [Pap03]).

Lemma 10: *For* QGPs *with numeric aggregates only, the quantified matching problem is* NP*-hard for positive* QGPs*, and* DP*-hard for (possibly negative)* QGPs. □

The upper bounds are verified by the next two lemmas. In particular, Lemma 12 shows that ratio aggregates can be encoded as numeric aggregates by transforming *both* query Q and graph G, in PTIME. This explains why positive QGPs with ratio aggregates retain the same complexity as conventional patterns, despite their increased expressivity.

Lemma 11: *For* QGPs *with numeric aggregates only, the quantified matching problem is in* NP *for positive* QGPs*, and is in* DP *for (possibly negative)* QGPs. □

Lemma 12: *Any* QGP $Q(x_o)$ *and graph* G *can be transformed in* PTIME *to* QGP $Q_d(x_o)$ *with numeric aggregates only and graph* G_d*, respectively, such that* $Q(x_o, G) = Q_d(x_o, G_d)$. □

Proof:

Lemma 10*: Lower bounds*. We first prove the lower bounds of quantified matching, for positive and negative QGPs.

(1) NP-hardness. We start by showing that quantified matching for positive QGPs with numeric aggregates is already NP-hard. To do this, we construct a polynomial time reduction from subgraph isomorphism (**subISO**). An instance of **subISO** consists of a graph pattern Q' and a graph G. It is to decide whether there exists a subgraph G' of G such that Q' is isomorphic to G'. Given Q' and G, we construct a corresponding QGP Q by (a) adding quantifier $\delta(e) \geq 1$ for all the edges e in Q', and (b) selecting an arbitrary pattern node in Q' and marking it as x_0 in Q. We keep G as is. One may verify that a node v_x is in $Q(x_0, G)$ if and only if there exists a subgraph isomorphism h' from Q' to G, such that $h'(x_0) = v_x$ for x_0 in Q'. As **subISO** is NP-hard (cf. [Pap03]), so is quantified matching.

(2) DP-hardness. We next prove that quantified matching for (possibly) negative QGPs is DP-hard by reduction from EXACT-CLIQUE, which is DP-complete (cf. [Pap03]). Given a graph G and a natural number k, EXACT-CLIQUE is to determine whether the

largest clique of G has size exactly k.

Given an instance of EXACT-CLIQUE, *i.e.,* a graph $G = (V,E)$ and a number k, we construct a new graph G' that contains G as its subgraph, and contains an additional node v_o, as well as a new edge from v_o to each of the nodes in G. We construct a QGP $Q(x_o)$ that consists of a k-clique Q_1, a $k+1$-clique Q_2 such that x_o has a unique match v_o in G', there is an edge e from x_o to each node in q_1 with $\sigma(e) \geq 1$, and to a single node v' in Q_2 via a negated edge.

Obviously the transformation is in PTIME. Moreover, the largest clique of G has size k if and only if $v_o \in Q(x_o, G')$. (1) If G has a largest clique G_k of size k, then we map Q_1 to the clique G_k and v_o to x_o. One may verify that v_o is a match of x_o. (2) If v_o is a match of x_o, then $v_o \in \Pi(Q)(x_o, G) \setminus \Pi(Q^{+(v_o,v')})$ by definition. (a) Since $v_o \in \Pi(Q)(x_o, G)$, G has a clique of size k, matching Q_1 in $\Pi(Q)$. (b) Since v_o is the only candidate for x_o and is in both $\Pi(Q)(x_o, G)$ and $Q(x_o, G)$, we must have that $\Pi(Q^{+(v_o,v')})$=$\emptyset$. This shows that there exists no subgraph in G that matches Q_2 as a $k+1$ clique. G has a largest clique of size k.

Therefore, the transformation above is a reduction. As EXACT-CLIQUE is DP-complete, quantified matching with (possibly) negative QGPs is DP-hard. Note that in the reduction above, only one non-existential quantifier is used.

Lemma 11*: Upper bounds*. We next prove the upper bounds for quantified matching. We first consider positive QGPs Q with numeric aggregates $\sigma(e) \odot p$ only (Lemma 11). We then extend the result to ratio aggregates $p\%$ (Lemma 12).

(1) Given a QGP $Q(x_o)$, we construct a traditional graph pattern $Q_e(x_o)$ without quantifiers, by (a) stripping off all quantifiers from Q, and (b) for each edge $e(u,u')$ associated with $\sigma(e) \odot p$, if $p > 1$, we make p copies of u' in Q_e as children of u, along with copies of edges from u' and so on. Then one can easily verify the following: (a) for any graph G, $Q(x_o, G) = Q_e(x_o, G)$, and (b) the time for constructing Q_e and hence $|Q_e|$ are both a polynomial in $|Q|$; this is because on each simple path in Q, there are at most k non-existential quantifiers, for a predefined constant k. These make a PTIME reduction from quantified matching with positive numeric aggregates to conventional subgraph isomorphism. Since the latter is in NP, so is the former.

(2) We next prove that the quantified matching for (possibly) negative QGPs is in DP. Following [Pap03], it suffices to construct two languages L_1 and L_2, such that a node v_x is in $Q(x_o, G)$ if and only if v_x is in $L_1 \cap L_2$.

We consider two languages below:

- L_1, the set $\Pi(Q)(x_o, G)$, and
- L_2, the set of "yes" instances for a node v_x that is not a match of x_o for $\bigcup_{e \in E_Q^-}(\Pi(Q^+)(x_o, G)$.

One can verify that (1) $L_1 \in \mathsf{NP}$, (2) $L_2 \in \mathsf{coNP}$, and (3) a node v_x is in $Q(x, G)$ if and only if v_x is in $L_1 \cap L_2$, by the definition of QGPs. Thus quantified matching is in DP.

Lemma 12*: Ratio aggregates*. Given a QGP Q that contains ratio aggregates $\sigma(e) \odot p\%$, we construct a QGP Q_d and graph G_d in PTIME such that Q_d consists of numeric aggregates only, and $Q_d(x_o, G_d) = Q(x_o, G)$. To simplify the discussion, we consider *w.l.o.g.* positive Q. For negated edges e, by the definition of $Q(x_o, G)$, e is positified in Q^{+e}. Hence, it suffices to consider positive edges.

(a) We transform G to a graph G_d as follows. For each node v with g child in G, we add $(1 - p\%)(d - g)$ dummy children with a label that does not match any pattern node in Q, and $p\%(d - g)$ dummy children that complete a dummy subgraph G_Q at v that is isomorphic to Q_π of Q.

(b) We transform Q to Q_d such that for each edge e with quantifier $\sigma(e) \odot p\%$, we replace $p\%$ with a constant $p\% * d$.

One may verify that a node $v_x \in Q(x_o, G)$ if and only if its G_d-counterpart $v_d \in Q_d(x_d, G_d)$. Moreover, the transformation is obviously in PTIME. Since quantified matching for all numeric quantified Q_d is in NP by Lemma 11, so is its counterpart for QGPs Q with ratio quantifiers. □

This completes the proof of Theorem 9.

Algorithm Match

Input: pattern $Q(x_o)$, graph G
Output: the answer set $Q(x_o,G)$

1. $Q(x_o,G)$:= $\emptyset$; $Q(G)$:=$\emptyset$; M := $\emptyset$;
2. **for each** u of Q **do**
3. $C(u)$:=Filtercandidate(Q,G,u);
4.   **if** $C(u)$=$\emptyset$ **then return** $\emptyset$;
5. SubMatch$(Q,G,M,Q(G))$;
6. **for each** isomorphic mapping $h \in Q(G)$ **do**
7. $Q(x_o,G)$:=$Q(x_o,G) \cup \{h(x_o)\}$;
8. **return** $Q(x_o,G)$;

Procedure SubMatch$(Q,G,M,Q(G))$

1. **if** Verify(M) **then**
2. $Q(G)$:=$Q(G) \cup \{h\}$; /*h: the isomorphism defined by M*/
3. **else** u:=SelectNext(Q);
4.   **for each** $v \in C(u)$ not matched in M **do**
5.   **if** IsExtend(Q,G,M,u,v) **then**
6. $M := M \cup \{(u,v)\}$;
7.   SubMatch$(Q,G,M,Q(G))$;
8.   Restore(M,u,v);
9. **return**;

Figure 4.4: Generic search procedure Match

4.3 Algorithms for Quantified Matching

We next provide an algorithm, denoted by QMatch, for quantified matching. It takes a QGP $Q(x_o)$ and a graph G as input, and computes $Q(x_o,G)$ as output. It extends existing algorithms $\mathcal{T}$ for conventional subgraph isomorphism, to incorporate quantifier checking and process negated edges.

Generic graph pattern matching. We start by reviewing a generic procedure for subgraph isomorphism, denoted by Match and shown in Fig. 4.4, slightly adapted from [LHKL12] to output $Q(x_o, G)$ for query focus x_o. As observed in [LHKL12], state-of-the-art graph pattern matching algorithms $\mathcal{T}$ typically adopt Match, and differ only in how to optimize key functions (*e.g.,* SelectNext, IsExtend; see below). Given a traditional pattern $Q(x_o)$ and a graph G, Match initializes $Q(x_o, G)$, as well as a partial match M as a set of node pairs (line 1). Each pair (u, v) in M denotes that a node from G matches a pattern node u in Q. It identifies a candidate match set $C(u)$ for each pattern node u in Q (lines 3-4) (FilterCandidate). If there exists a pattern node u with no candidate, it returns $\emptyset$ (line 4). Otherwise, it invokes SubMatch to compute *all matches* (isomorphic mappings) $Q(G)$ (lines 5). It then computes and returns query answer $Q(x_o, G)$ from mappings $h \in Q(G)$ (lines 6-8).

Procedure SubMatch recursively extends partial match M by using three key functions. (1) It picks a pattern node u from Q that has no match yet (SelectNext, line 3). (2) It then checks whether a candidate v of u not yet in M matches u (IsExtend), and if so, it adds (u, v) to M (lines 4-6). (3) It recursively calls SubMatch to extend M with steps (1) and (2) (line 7), and restores it when SubMatch backtracks (line 8). If M is a valid isomorphism (Verify, line 1), it adds M to $Q(G)$ (line 2). This continues until $Q(G)$ is completed.

4.3.1 Quantified Graph Pattern Matching

Algorithm QMatch revises the generic Match to process quantifiers. (1) It first adopts a *dynamic selection and pruning strategy* to compute $\Pi(Q)(x_o, G)$. The dynamic search picks top p promising neighbors based on a potential score, with p adapted to the corresponding quantifiers. (2) It then employs *optimal incremental evaluation* to process negated edges, which maximally reuses cached matches for $\Pi(Q)$ when processing Q^{+e} for positified e, instead of recomputing $Q^{+e}(G)$ starting from scratch. The strategies are supported by optimized data structures and key functions from Match.

Auxiliary structures. QMatch maintains auxiliary structures for each node v in $C(u)$ as follows: (1) a Boolean variable $X(u, v)$ indicating whether v is a match of u via isomorphism from $\Pi(Q)$ to G, and (2) a vector T, where entry $T(v, e)$ for an edge $e{=}(u, u')$ in Q is a pair $\langle c(v, e), U(v, e)\rangle$, in which c (resp. U, initialized as $M_e(v)$) records the current size (resp. an estimate upper bound for) $|M_e(v_x, v, Q)|$.

Algorithm. Algorithm QMatch (outlined in Fig. 4.5) revises Match to process QGP

Algorithm QMatch

Input: a QGP $Q(x_o)$, graph G
Output: the answer set $Q(x_o,G)$

1. $Q(x_o,G)$:= $\emptyset$; M:=$\emptyset$; $\Pi(Q)(x_o,G)$:=$\emptyset$; $\Pi(Q)(G)$:=$\emptyset$;
2. **for each** u of Q **do**
3. initializes $C(u)$ and auxiliary structures;
4. $\Pi(Q)(x_o,G)$:=DMatch$(\Pi(Q),G,M,\Pi(Q)(G))$;
5. **for each** negative edge e in E_Q^- **do**
6. $Q^{+e}(x_o,G)$:=IncQMatch$(\Pi(Q)(x_o,G),Q^{+e})$;
7. $Q(x_o,G)$:=$\Pi(Q)(x_o,G) \setminus \bigcup_{e\in E_Q^-} Q^{+e}(x_o,G)$;
8. **return** $Q(x_o,G)$;

Figure 4.5: Algorithm QMatch

$Q(x_o)$ in three steps. (1) It first initializes the candidate set and auxiliary structures with a revised Filtercandidate (lines 1-3). For each pattern node u in $Q(x_o)$, it initializes (a) $C(u)$ with nodes v of the same label, and (b) $X(u,v) = \perp$, $c(v,e)$=0 and $U(v,e)$ = $|M_e(v)|$ for each e=(u,u') in Q. It removes v from $C(u)$ if $U(v,e)$ does not satisfy the quantifier of e. (2) It next invokes a procedure DMatch revised from SubMatch in Fig. 4.4 to compute $\Pi(Q)(x_o,G)$ (line 4). (3) It then processes each negated edge e by constructing its positified pattern Q^{+e}, and computes $Q^{+e}(x_o,G)$ with an incremental procedure IncQMatch (lines 5-6). (4) It computes $Q(x_o,G)$ by definition (line 7). We next present DMatch, and defer IncQMatch to Section 4.3.2.

Example 18: Given Q_3 with p=2 (Fig. 4.1) and G_1 (Fig. 4.2), QMatch first computes $\Pi(Q_3)(x_o,G_1)$ (Fig. 4.3). It initializes variables for nodes in G_1, partially shown in Table 4.2 ($i \in [0,4]$).

At this stage, since $U(x_1,(x_o,z_1))$=1 $\leq$ 2, x_1 fails the quantifier of (x_o,z_1), and is removed from $C(x_o)$. □

Procedure DMatch. Given positive QGP $\Pi(Q)$, DMatch revises SubMatch (Fig 4.4) by adopting dynamic search. To simplify the discussion, we consider numeric $\sigma(e) \odot p$ first.

	X	c	U
x_1	$X(x_o,x_1)=\perp$	$c(x_1,(x_o,z_1))=0$	$U(x_1,(x_o,z_1))=1$
x_2	$X(x_o,x_2)=\perp$	$c(x_2,(x_o,z_1))=0$	$U(x_2,(x_o,z_1))=2$
x_3	$X(x_o,x_3)=\perp$	$c(x_3,(x_o,z_1))=0$	$U(x_3,(x_o,z_1))=3$
v_i	$X(x_o,v_i)=\perp$	$c(v_i,(z_1,\texttt{Redmi}))=0$	$U(v_i,(z_1,\texttt{Redmi}))=1$

Table 4.2: Running example for QMatch, quantifier check

(1) Given a selected pattern node u' (line 3 of SubMatch), a candidate $v \in C(u)$, and an edge $e=(u,u')$ with quantifier $\sigma(e) \odot p$, DMatch *dynamically* finds p best nodes (recorded in a heap $S_P(u')$) from $C(u')$ that are children of v (lines 4-5 of SubMatch, IsExtend), using selection and pruning rules. Denote as $P(v')$ the parent set of v' in G, the *potential* of a match $v' \in C(u')$ is defined as:

$$(1+\frac{|P(v')\cap C(u)|}{|C(u)|})*\Sigma_{\forall e=(u',u'')}\frac{U(v',e)}{p_e},$$

where p_e is the number in $\sigma(e) \odot p_e$ for edge $e=(u',u'')$. It favors those candidates that (a) benefit the verification of more candidates during future backtracking, and (b) have high upper bounds *w.r.t.* p (hence more likely to be a match itself). We select candidates with the highest scores.

DMatch then updates M by including (u,v), and recursively conducts the next level of search by forking p verifications in the order of the selected p candidates (line 7, SubMatch). It keeps a record of M and a cursor to memorize the candidates in S_P for backtracking, using a stack.

(2) When backtracking to a candidate $v \in S_P(u)$ from a child v' of v, DMatch restores M and the cursor (Restore, line 8 of SubMatch). It next dynamically updates $S_P(u)$. (a) If $X(u',v')$=false, it reduces $U(v,e)$ by 1. (b) It applies the selection and pruning rules to $C(u)$ using the *updated* potentials *w.r.t.* the changes in (a). If the upper bound $U(v,e)$ fails the quantifier of e, v is removed from $C(u)$ and $S_P(u)$ *without* further verifying its other children. Otherwise, it picks a new set $S_P(u)$ of candidates with top potentials.

(3) When M is complete, *i.e.,* each node u in $\Pi(Q)$ has a match in M, DMatch checks whether M is an isomorphic mapping. If so, it updates $X(u,v)$=true for each pair $(u,v) \in M$, and increases the counter $c(v,e)$. It then checks whether the counters *satisfy the quantifiers* of $\Pi(Q)$. If so, it adds v_x to $Q(x_o,G)$. Otherwise, it proceeds. DMatch terminates when all the candidates of x_o are checked.

Example 19: Continuing with Example 18, given $C(x_o) = \{x_2, x_3\}$, DMatch selects x_2, and extends M with (x_o, x_2). In contrast to Fig 4.4, DMatch picks top 2 best candidates $S_P(z_1) = \{v_2, v_1\}$ from $C(z_1)$ following edge $e{=}(x_o, z_1)$. This adds (z_1, v_2) to M, and (Redmi 2A, Redmi 2A) for the next round. At verification, it finds M a complete isomorphism, and updates $X(v_o, x_2)$=true and $c(x_2, e)$=1. As x_2 cannot be verified as a match via $\Pi(Q_3)$ yet, DMatch next verifies v_1, and sets $c(x_2, e)$=2. As x_2 is a match and has a counter satisfying the quantifier, it is added to $\Pi(Q_3)(x_o, G_1)$. The updated variables for candidates of $C(x_o)$ are as follows.

	X	c	U
x_2	$X(x_o, x_2)$=True	$c(x_2, (x_o, z_1))$=2	$U(x_2, (x_o, z_1))$=2
x_3	$X(x_o, x_3)=\perp$	$c(x_3, (x_o, z_1))$=2	$U(x_3, (x_o, z_1))$=3

Table 4.3: Running example for QMatch, next verificaiton

DMatch next verifies x_3. It starts by selecting top 2 candidates $S_P(x_3)=\{v_2, v_3\}$. Once v_3 is processed, it finds that x_3 is in an isomorphism with $c(x_3, e)$=2, and hence is a match. It returns $\{x_2, x_3\}$ as $\Pi(Q_3)(x_o, G_1)$. □

One can readily verify the following.

Lemma 13: DMatch *computes $\Pi(Q)(x_o, G)$ by (a) verifying no more candidates than any* Match*-based subgraph isomorphism algorithm $\mathcal{T}$, and (b) with space cost $O(p_m|Q| + |V|)$, where p_m is the largest constant in all quantifiers of Q.* □

Proof: To show the correctness of DMatch, first observe that DMatch always terminates. Indeed, DMatch follows the verification process of conventional subgraph isomorphism algorithm. The process, in the worst case, enumerates all possible isomorphism mappings from the stratified pattern Q_π to G, which are finitely many. Hence DMatch terminates.

We next show that DMatch correctly verifies whether a candidate v_x is a match of x_o in $\Pi(Q)$ via an isomorphism $h_0 \in \Pi(Q)(G)$. It suffices to show that (1) h_0 is a match in $Q_\pi(G)$, and (2) for each u in $\Pi(Q)$ and each edge $e{=}(u, u')$, $|M_e(h_0(x_o), h_0(u), Q)| \odot p$ for $f(e) = \sigma(e) \odot p$.

(1) When DMatch terminates, for each $u \in \Pi(Q)$ and every candidate v in $C(u)$ with $X(u, v)$=true, $v = h(u)$ for some $h \in Q_\pi(G)$, guaranteed by the correctness of Match.

(2) For each edge (u,u') in $\Pi(Q)$ and a node v with $X(u,v)$=true, DMatch correctly verifies the quantifiers by checking the updated local counter of v that keeps track of the current $|M_e(h_0(x_o),h_0(u),Q)|$. In addition, DMatch *waits* until either v is determined not a valid match due to that the upper bound fails the quantifier (by the local pruning rule), or the lower bound satisfies the quantifier (in the verification). Hence, v_x is a match if and only if $v_x \in \Pi(Q)(x_o,G)$ when DMatch terminates.

For the space complexity, it takes $O(|V|)$ space to store the auxiliary structures for the nodes in G. During the search, DMatch keeps, at each level of the search, at most p_m best matches to be verified, where p_m is the largest constant in quantifiers. Since there are in total $|\Pi(Q)| \leq |Q|$ levels of search, it takes in total $O(p_m|Q|+|V|)$ space. □

That is, quantified matching can be evaluated following conventional $\mathcal{T}$ without incurring significant extra time and space cost. The performance of DMatch is further improved by selection and pruning rules.

Ratio aggregates. DMatch can be readily extended to process ratio aggregates. Indeed, for each pattern e=(u,u') with $\sigma(e) \odot p\%$ and at a candidate v of u, DMatch "transforms" the quantifier to its equivalent numeric counterpart $\sigma(e) \odot p'$ as follows. (a) DMatch computes $|M_e(v)|$ by definition. (2) It sets $p' = \lfloor |M_e(v)| * p\% \rfloor$. The transformation for e preserves all the exact matches for ratio quantifiers by definition, and takes a linear scan of G (in $O(|G|)$ time). In addition, QMatch easily extends to QGPs with quantifiers $\sigma(e) > p$, by replacing it with $\sigma(e) \geq p+1$.

4.3.2 Incremental Quantified Matching

If $\Pi(Q)(x_o,G)$ is nonempty, QMatch proceeds to compute $\Pi(Q^{+e})(x_o,G)$ for each negated edge $e \in E_Q^-$ (lines 5-6, Fig. 4.5). Observe the following: (1) $\Pi(Q^{+e})$=$\Pi(Q) \oplus \Delta E$, *i.e.,* $\Pi(Q^{+e})$ "expands" $\Pi(Q)$ with a set ΔE of positive edges; and (2) for any node u in $\Pi(Q)$, $\Pi(Q^{+e})(u,G) \subseteq \Pi(Q)(u,G)$, since $\Pi(Q^{+e})$ adds more constraints to $\Pi(Q)$.

This observation motivates us to study a novel *incremental quantified matching* problem. Given a graph G, a QGP Q, computed matches $Q(u,G)$ for each u in Q, and a new QGP Q'=$Q \oplus \Delta E$, it is to compute $Q'(x_o,G)$ = $Q(x_o,G) \oplus \Delta O$, *i.e.,* to find changes ΔO in the output. It aims to make maximum use of cached results $Q(u,G)$, instead of computing $Q'(x_o,G)$ from scratch. As opposed to conventional incremental problems [RR96a, FWW13], we compute ΔO in response to changes in *query* Q, rather

than to changes in graph G.

As observed in [RR96a], the complexity of incremental graph problems should be measured in the size of *affected area*, which indicates the amount of work that is necessarily performed by any algorithm for the incremental problem. For pattern matching via subgraph isomorphism, the number of verifications is typically the major bottleneck. Below we identify affected area for quantified matching, to characterize the optimality of incremental quantified matching.

Optimal incremental quantified matching. Given Q and $\Pi(Q^{+e})$, the *affected area* is defined as

$$\mathsf{AFF} = \bigcup C(u_i) \cup \{N(v) \mid v \in C(u_i)\},$$

where (1) u_i is in edge e_i=(u_i, u_i') or (u_i', u_i) for each $e_i \in \Delta E$; (2) $C(u_i)$ includes (a) the match sets cached after DMatch processed $\Pi(Q)$; and (b) the candidate sets initialized by QMatch (line 3 of Fig 4.5) for new nodes u_i introduced by $\Pi(Q^{+e})$, which have to be checked; and (c) $N(v)$ is the set of nodes in cached $C(\cdot)$ that are reachable from (or reached by) v, via paths that contains only the nodes in $C(\cdot)$.

An incremental quantified matching algorithm is *optimal* if it incurs $O(|\mathsf{AFF}|)$ number of verifications. Intuitively, AFF is the set of nodes that are necessarily verified in response to ΔE, for any such algorithms to find exact matches.

Proposition 14: *There exists an incremental algorithm that computes each* $\Pi(Q^{+e})(x_o, G)$ *by conducting at most* $|\mathsf{AFF}|$ *rounds of verification.* □

As a proof, we present an optimal algorithm IncQMatch.

Procedure IncQMatch. Algorithm IncQMatch (used in line 6, Fig. 4.5) incrementally computes $\Pi(Q^{+e})(x_o, G)$ by reusing the cached match sets and the counters computed in the process of DMatch for $\Pi(Q)$. It works as follows.

(1) IncQMatch initializes $\Pi(Q^{+e})(u, G)$ for each u with the cached matches $\Pi(Q)(u, G)$. It then computes the edge set ΔE in $\Pi(Q^{+e})(x_o, G)$ to be "inserted" into $\Pi(Q)$.

(2) IncQMatch then iteratively processes the edges e=(u, u') in ΔE. It first identifies those cached matches that are *affected* by the insertion. It considers two possible cases below.

- Both u and u' are in $\Pi(Q)$. For each match $v \in \Pi(Q)(u, G)$, IncQMatch adds v to AFF and verifies whether v matches u via isomorphism following DMatch. If

pattern node	$C(\cdot)$
x_o	$C(x_o)$=$\{x_2, x_3\}$
z_1	$C(z_1)$=$\{v_1, v_2, v_3\}$
z_2	$C(z_2)$=$\{v_4\}$
Redmi	C(`Redmi 2A`)=$\{$Redmi 2A$\}$

Table 4.4: Computation of IncQMatch

$X(u,v)$=true, it counts $c(v,e)$ as the number of v's children v' that are matches of u' and checks the quantifier of e. If $c(v,e)$ satisfies the quantifier, no change happens. Otherwise ($c(v,e)$ fails the quantifier) IncQMatch removes v from $\Pi(Q^{+e})(u,G)$.

- One or both of nodes u and u' are not in $\Pi(Q)$. For the new node u (or u'), IncQMatch treats e as a single edge pattern and verifies each candidate v_1 in $C(u)$. Since $\Pi(Q)$ is a "sub-pattern" of $\Pi(Q^{+e})$ and $\Pi(Q^{+e})$ is connected, we need only to inspect those matches v_1 reachable from some nodes in cached $C(.)$, *i.e.,* $v_1 \in N(v_2)$ for some cached v_2; hence $v_1 \in$ AFF.

For each v removed in the steps above, QMatch then propagates the impact recursively by (a) reducing all the counters of v's parents by 1, and (b) removing invalid matches due to the updated counters and adds them to AFF, following the same backtracking verification as in DMatch, until a fixpoint is reached, *i.e.,* no more matches can be removed.

Example 20: Continuing with Example 19, QMatch invokes IncQMatch to process $\Pi(Q_3^{+(x_o,z_2)})$, with ΔE = $\{(x_o,z_2)$, (z_2, `Redmi 2A`)$\}$ (see Fig. 4.3) as follows. (see Table 4.4)

(1) IncQMatch first initializes the candidate sets as the cached matches in DMatch (shown below). For node z_2 not in $\Pi(Q)$, IncQMatch finds $C(z_2)$ as initialized in QMatch.

(2) It starts with edge (z_2, `Redmi 2A`), and initializes AFF as $C(z_2) \cup C$(`Redmi 2A`)=$\{v_4$, Redmi 2A$\}$. It next checks whether v_4 and Redmi 2A remain matches with counter satisfying the quantifiers. In this process, it only visits the two cached matches x_3 and v_3 following the pattern edges. As both nodes are matches, no change needs to be made.

(3) IncQMatch next processes edge (x_o, z_2). It adds the set $C(x_o)$=$\{x_2, x_3\}$ to AFF, and checks whether x_2 and x_3 remain matches. As x_2 has no edge to v_4, $X(x_o, x_2)$ is updated to false, and x_2 is removed from $C(x_o)$. It next finds that x_3 is a valid match, by visiting v_2, v_3, Redmi 2A, and v_4.

As no more matches can be removed, IncQMatch stops the verification. It returns $\Pi(Q_3^{+(x_o,z_2)})(x_o, G_1)$ as $\{x_3\}$. After the process, AFF contains $\{v_4, x_2, x_3$, v_2, v_3, Redmi 2A$\}$. It incurs in total 3 rounds of verification for v_4, x_2 and x_3. □

Contrast IncQMatch with DMatch. (1) IncQMatch only visits the cached matches and their edges, rather than the entire G. (2) IncQMatch incurs at most |AFF| rounds of verifications; hence it is *optimal w.r.t.* incremental complexity.

Analysis of QMatch. Algorithm QMatch correctly computes $Q(x_o, G)$ following the definition of quantified matching (Section 4.1.2). For its complexity, observe the following.

(1) If Q is positive, *i.e.,* $E_Q^- = \emptyset$, IncQMatch is not needed. Then QMatch and a conventional Match-based algorithm $\mathcal{T}$ for subgraph isomorphism have the same complexity. Quantifier checking is incorporated into the search process.

(2) If E_Q^- is nonempty, IncQMatch invokes at most $|E_Q^-|$ rounds of incremental computation by *optimal* IncQMatch, while $|E_Q^-| \leq |Q|$ and Q is typically small in practice.

For each round, the overall time taken is bounded by $|\mathsf{AFF}| * K$, where K is the cost of a single verification.

Put together, QMatch takes $O(t(\mathcal{T}) + |E_Q^-||\mathsf{AFF}| * K)$ time in total, where $t(\mathcal{T})$ is the time complexity of a Match-based algorithm $\mathcal{T}$ for conventional subgraph isomorphism. We find in our experiments that QMatch and $\mathcal{T}$ have comparable performance, due to small $|E_Q^-|$ and |AFF|. Moreover, existing optimization for $\mathcal{T}$ can be readily applied to QMatch.

Algorithm QMatch also makes use of graph simulation [HHK95] to filter candidates and reduce verification cost.

4.4 Parallel Quantified Matching

Quantified matching – in fact even conventional subgraph isomorphism – may be cost-prohibitive over big graphs G. This suggests that we develop a parallel algorithm for quantified matching that guarantees to scale with big G. We develop such an algorithm, which makes quantified matching feasible in real-life graphs, despite its DP complexity.

4.4.1 Parallel Scalability

To characterize the effectiveness of parallelism, we recall the notion of *parallel scalability* introduced in Section 3.4.1. Consider a problem A posed on a graph G. We denote by $t(|A|,|G|)$ the running time of the best *sequential algorithm* for solving A on G, *i.e.,* one with the least worst-case complexity among all algorithms for A. For a parallel algorithm, we denote by $T(|A|,|G|,n)$ the time it takes to solve A on G by using n processors, taking n as a parameter. Here we assume $n \ll |G|$, *i.e.,* the number of processors does not exceed the size of G; this typically holds in practice as G often has trillions of nodes and edges, much larger than n [GBDS14].

Parallel scalability. An algorithm is *parallel scalable* if

$$T(|A|,|G|,n) = O(\frac{t(|A|,|G|)}{n}) + (n|A|)^{O(1)}.$$

That is, the parallel algorithm achieves a linear reduction in sequential running time, plus a "bookkeeping" cost $O((n|A|)^l)$ that is *independent of* $|G|$, for a constant l.

A parallel scalable algorithm *guarantees* that the more processors are used, the less time it takes to solve A on G. Hence given a big graph G, it is feasible to efficiently process A over G by adding processors when needed.

4.4.2 Parallel Scalable Algorithm

Parallel scalability is within reach for quantified matching under certain condition. We first present some notations. For a node v in graph G and an integer d, the *d-hop neighbor* $N_d(v)$ of v is defined as the subgraph of G induced by the nodes within d hops of v. The *radius* of a QGP $Q(x_o)$ is the longest shortest distance between x_o and any node in Q.

The main result of the section is as follows.

Theorem 15: *There exists an algorithm* PQMatch *that given* QGP $Q(x_o)$ *and graph* G*, computes* $Q(x_o, G)$*. It is parallel scalable for graphs* G *with* $\sum_{v\in G}|N_d(v)| \leq C_d * \frac{|G|}{n}$*, taking* $O(\frac{t(Q,G)}{n} + n)$ *time, where* d *is the radius of* $Q(x_o)$*,* C_d *is a predefined constant, and* $t(Q,G)$ *is the worst-case running time of sequential quantified matching algorithms.* □

The condition is practical: 99% of real-life patterns have radius at most 2 [GFMPdlF11], and the average node degree is 14.3 in social graphs [BW13]; thus $|N_d(v)|$ is often a small constant. In addition, we will show that PQMatch can be adapted to evaluate QGPs Q with radius larger than d.

As a proof, below we present PQMatch. The algorithm works with a coordinator S_c and n workers (processors) S_i. It utilizes two levels of parallelism. (a) At the *inter-fragment parallelism* level, it creates a partition scheme of G over multiple processors *once for all*, so that quantified matching is performed on all these fragments in parallel. The same partition is used for all QGPs $Q(x_0)$ within radius d. (b) At the *intra-fragment* level, local matching within each fragment is further conducted by multiple threads in parallel.

Hop preserving partition. We start with graph partition. To maximize parallelism, a partition scheme should guarantee that for any graph G, (1) each of n processors manages a small fragment of approximately equal size, and (2) a query can be evaluated locally at each fragment without incurring inter-fragment communication. We propose such a scheme.

Given a graph $G = (V, E, L)$, an integer d and a node set $V' \subseteq V$, a *d-hop preserving partition* $\mathcal{P}_d(V')$ of V' distributes G to a set of n processors such that it is

(1) *balanced*: each processor S_i manages a fragment F_i, which contains the subgraph G_i of G induced by a set V_i of nodes, such that $\bigcup V_i$=V' ($i \in [1,n]$) and the size of F_i is bounded by $c * \frac{|G|}{n}$, for a small constant $c < C_d$; and

(2) *covering*: each node $v \in V'$ is *covered* by $\mathcal{P}_d(V')$, *i.e.,* there exists a fragment F_i such that $N_d(v)$ is in F_i.

We say that $\mathcal{P}_d(V')$ is *complete* if $|V'| = |V|$.

One naturally wants to find an optimal partition such that the number $|V'|$ of covered nodes is maximized. Although desirable, creating a balanced d-hop preserving partition is NP-hard. Indeed, conventional balanced graph partition is a special case when d=1, which is already NP-hard [AR06].

Parallel d-hop preserving partition. We provide an approximation algorithm for d-hop preserving partition with an approximation ratio. Better still, it is parallel scalable.

Lemma 16: *If $\sum_{v \in G} |N_d(v)| \leq C_d * \frac{|G|}{n}$, for any constant $\varepsilon > 0$, there is a parallel scalable algorithm with approximation ratio $1+\varepsilon$ to compute a d-hop preserving partition.* □

Proof: The d-hop preserving partition problem at the coordinator S_c is to find an assignment for each $N_d(v)$ to a worker S_i, such that (a) $\sum |N_d(v_i)| \leq c * \frac{|G|}{n}$ for all $N_d(v_i)$ assigned to S_i, and (b) $|V_c|$ is maximized, where V_c refers the nodes covered in V'. We show that the problem is $1+\varepsilon$-approximable, by constructing an approximation preserving reduction (APR) to the multiple knapsack problem (MKP). An MKP instance consists of (1) an item set U, where each item u_i has a weight w_i and a value, and (2) a bin set B, where each bin B_i has a capacity b_i. It is to find a packing of items to the bins subject to their capacity, such that the total value is maximized. It is known that MKP is $1+\varepsilon$ approximable [CK00].

An APR from I_1 to I_2 consists of a pair of functions (f,g), where f transforms the instance I_1 to I_2, g transforms a feasible solution $s(I_2)$ for I_2 to a feasible solution $s(I_1)$, and if $s(I_2) \geq \frac{s^*(I_2)}{1+\alpha(\varepsilon)}$ given the optimal solution $s^*(I_2)$, then $g(s(I_2)) \geq \frac{s^*(I_1)}{1+\varepsilon}$. We construct an APR as follows.

- **function** f: (a) for each $v \in V'$, construct an item u_i with value 1 and weight $|N_d(v)|$; and (b) for each worker S_i, construct a bin B_i with capacity $c * \frac{|G|}{n} - |F_i|$.
- **function** g: for each item u_i packed to a bin B_i in $s(I_2)$, g maps u_i to v_i, and B_i to S_i.

We next show that the transformation above is an APR. Indeed, (1) f is in PTIME, and (2) for a feasible solution $s(I_2)$, $g(s(I_1))$ is also a feasible solution, since the packing does not exceed the capacity constraints in $s(I_2)$ if and only if the assignment $g(s(I_1))$ does not exceed the capacity of each fragment. (3) Assume $s(I_2) \geq \frac{s^*(I_2)}{1+\varepsilon}$ (α=1). One can verify that $|s^*(I_1)|= |V_c|$ (the size of covered nodes) = $|g(s^*(I_2)|$ (the size of packed items), $|s(I_1)| = |g(s(I_2)|$. Hence $g(s(I_2))$=$s(I_1) \geq \frac{s^*(I_2)}{1+\varepsilon} = \frac{g(s^*(I_2))}{1+\varepsilon}$. Thus, the transformation is an APR. As a result, from [CK00] it follows that d-hop preserving partition is $1+\varepsilon$ approximable.

Remarks. We use a balanced bound for all fragments. This guarantees the correctness of the reduction to MKP. The choice of MKP is to get a balanced fragment bound and at the same time, to minimize synchronization cost. Our experimental study shows that

this leads to quite balanced fragments (Exp-2), with minimum communication cost.

□

Below we present such an algorithm, denoted by DPar. Given a graph G stored at the coordinator S_c, it starts with a base partition of G, where each fragment F_i has a balanced size bounded by $c * \frac{|G|}{n}$. This can be done by using an existing balanced graph partition strategy (*e.g.,* [Kar11]). DPar then extends each fragment F_i to a d-hop preserving counterpart.

(1) It first finds the "border nodes" $F_i.O$ of F_i that have d-hop neighbors not residing in F_i, by traversing F_i in parallel.

(2) Each worker S_i then computes and loads $N_d(v)$ for each $v \in F_i.O$, by "traversing" G via disk-based parallel breadth-first search (BFS) search [Kor08].

Moreover, DPar uses a balanced loading strategy (see below) to load approximately equal amount of data to each worker in the search. The process repeats until no fragments can be expanded.

Balancing strategy. DPar enforces a balanced fragment size $c * \frac{|G|}{n}$. It conducts a d-hop preserving partition $\mathcal{P}_d(V')$ with approximation ratio $1-\varepsilon$ subject to the bound, for any given ε. That is, if the size of nodes covered by the optimal d-hop partition in G is $|V^*|$, then $\mathcal{P}_d(V')$ has $|V'| \geq (1-\varepsilon)|V^*|$.

More specifically, at the BFS phase, for each $v \in \bigcup F_i.O$, DPar assigns $N_d(v)$'s to workers by reduction to *Multiple Knapsack problem* (MKP) [CK00]. Given a set of weighted items (with a value) and a set of knapsack with capacities, MKP is to assign each item to a knapsack subject to its capacity, such that the total value is maximized. DPar treats each $N_d(v)$ as an item with value 1 and weight $|N_d(v)|$, and each fragment as a knapsack with capacity $c * \frac{|G|}{n} - |F_i|$, with the number of covered nodes as the total value. It solves the MKP instance by invoking the algorithm of [CK00], which computes an assignment with approximation ratio $1+\varepsilon$ for any given ε, in $O(|V'|^{\frac{1}{\varepsilon}})$ time. Each worker S_i then loads its assigned $N_d(v)$. This gives us a d-hop preserving partition $\mathcal{P}_d$ with ratio $1+\varepsilon$.

Partition $\mathcal{P}_d$ may not be complete, *i.e.,* not every node in V is covered. To maximize inter-fragment parallelism, DPar "completes" $\mathcal{P}_d$ while preserving the balanced partition size. For each uncovered node v, it assigns $N_d(v)$ to a worker S_i that minimizes estimated size difference $|F_{max}| - |F_{min}|$, where F_{max} (resp. F_{min}) is the largest (resp. smallest) fragment *if* $N_d(v)$ is merged to F_i. Since $\sum |N_d(v)| \leq C_d * \frac{|G|}{n}$, this

Algorithm PQMatch

Input: QGP $Q(x_o)$, graph G, coordinator S_c, n workers $S_1, \ldots, S_n$
Output: the answer set $Q(x_o, G)$.

1. DPar(G); /*Preprocessing*/
/*executed at coordinator S_c*/
2. $Q(x_o, G)$:= $\emptyset$; post Q to each worker;
3. **if** every worker S_i returns answer $Q(x_o, F_i)$ **then**
4. $Q(x_o, G)$:=$\bigcup Q(x_o, F_i)$;
5. **return** $Q(x_o, G)$;
/*executed at each worker in parallel*/
6. $Q(x_o, F_i)$:= mQMatch(b, Q, F_i); /* b: the # of threads*/
7. **return** $Q(x_o, F_i)$;

Figure 4.6: Algorithm PQMatch

suffices to make $\mathcal{P}_d$ both complete and d-preserving.

Example 21: Consider graph G_2 of Fig. 4.2 and a set $V' = \{v_5, \ldots, v_9\}$. Assume a base partition distributes $\{v_5\}$ to worker S_1, $\{v_7, v_9\}$ to S_2, $\{v_6, v_8\}$ to S_3, respectively. DPar creates a 1-hop preserving partition $\mathcal{P}_1$ for V' as follows. (1) Each S_i identifies its border nodes $F_i.O$ by a local traversal, *e.g.,* $v_5 \in F_1.O$. (2) Each S_i traverses G_2 at S_c and finds $N_1(v)$ for $v \in F_i.O$, in parallel. At the end, S_c keeps track of the nodes (edges) "requested" by workers as follows.

node	requested by
x_4	S_1, S_3
x_5, x_6, **prof.**	S_2, S_3
PhD.	S_1, S_2, S_3

Table 4.5: Vertex requests in DPar

DPar next determines which site to send the border nodes by solving an MKP instance, shown as follows.
Here $N_1(v_5)$ includes three nodes x_4, v_5, PhD, and three edges (x_4, v_5), $(v_5$, PhD) and

site	$N_1(\cdot)$	(estimated) $\|F_i\|$
S_1	$N_1(v_5), N_1(v_9)$	14 ($\|N_1(v_5)\|$ = 6, $\|N_1(v_9)\|$ = 8)
S_2	$N_1(v_7)$	8
S_3	$N_1(v_8)$	15

Table 4.6: Cost estimation in DPar

(x_4, PhD); similarly for the others. This induces a 1-hop preserving partition $\mathcal{P}_1$ that covers $\{v_5, v_7, v_8, v_9\}$. To complete $\mathcal{P}_1$, DPar selects S_2 to load $N_1(v_6)$, where $|N_1(v_6)| = 15$. This minimizes the estimated size $|F_{max}| - |F_{min}|$ = 19 − 14 = 5. Here $|F_{max}|$ is estimated as the sum of $|F_2| = 8$ and 11 additional nodes and edges in $N_1(v_6)$ that are not "requested" by S_2 (*e.g.,* (x_4, v_6), (v_6, PhD)). The completed $\mathcal{P}_1$ covers V' with fragment size 14, 19 and 15 for S_1, S_2 and S_3, respectively. □

Parallel algorithm. Using DPar, we next develop algorithm PQMatch. As shown in Fig. 4.6, PQMatch takes as input a QGP $Q(x_o)$ of radius at most d, and a graph G distributed across n workers by DPar, where fragment F_i of G resides at worker S_i. It works as follows. (1) The coordinator S_c posts $Q(x_o)$ to each worker S_i (line 2). (2) Each worker S_i then invokes a procedure mQMatch to compute local matches $Q(x_o, F_i)$ (line 7), where mQMatch implements QMatch using multi-threading (see below). Once verified, $Q(x_o, F_i)$ is sent to S_c (line 6). (3) Once all the workers have sent their partial matches to S_c, the coordinator computes $Q(x_o, G)$ as the union of all $Q(x_o, F_i)$ (lines 3-4).

Procedure mQMatch. Procedure mQMatch is a multi-threading implementation of PQMatch (Section 4.3), supporting inter-fragment level parallelism.

For pattern edge $e = (u, u')$ with quantifier $\sigma(e) \odot p$ and a candidate v in $C(u)$, it spawns p threads to simultaneously verify the top p selected candidates, one for each. Each thread i maintains local partial matches (in its local memory). When all the p threads backtrack to v, the local partial matches are merged, and the local counter of u is updated by aggregating the local storage of each thread i.

From Lemma 16 and Lemma 17 below, Theorem 15 follows. We remark that G is partitioned *once* by using a d-hop preserving partition process. Then for all QGPs with radius within d, *no re-partitioning* is needed. That is, condition $\sum |N_d(v)| \leq C_d * \frac{|G|}{n}$ is needed only for d-hop preserving partition to be parallel scalable.

Lemma 17: *Given G distributed over n processors by a d-hop preserving partition $\mathcal{P}_d$,*

(1) $Q(x_o, G) = \bigcup_{i\in[1,n]} Q(x_o, F_i)$*, and (2)* mQMatch *is parallel scalable for all* QGPs $Q(x_o)$ *with radius bounded by* d*.* □

Proof: We prove Theorem 15 by providing the correctness and complexity analysis below for algorithm PQMatch.

Correctness. Given graph G distributed over n processors by a d-hop preserving partition P_d, PQMatch computes $Q(x_o, G)$ as $\bigcup Q(x_o, F_i)$ ($i \in [1, n]$), for any QGP $Q(x_o)$ with radius bounded by d. It suffices to show Lemma 17(1).

Lemma 17(1). Observe the following. (1) For any match $v_x \in Q(x_o, F_i)$, QMatch only needs to visit $N_d(v_x)$ to verify whether v_x is a match. (2) For every candidate $v_x \in C(x_o)$, there exists a fragment F_i, such that $N_d(v_x) \subseteq F_i$ (including v_x) (by d-hop preservation). Hence, any match of x_o must be from at least one match set $Q(x_o, F_i)$ evaluated at fragment F_i, *i.e.,* $Q(x_o, G) \subseteq \bigcup Q(x_o, F_i)$ ($i \in [1, n]$). (3) For every match $v_x \in Q(x_o, F_i)$ locally computed at F_i, v_x is a match of x_o guaranteed by the correctness of QMatch. Hence PQMatch correctly computes $Q(x_o, G)$ as $\bigcup Q(x_o, F_i)$ ($i \in [1, n]$) over a d-hop preserving partition.

Complexity. Algorithm PQMatch consists of three steps: (1) the distribution of Q and construction of d-hop preserving partition, (2) the parallel evaluation, and (3) assembling of partial matches. The time for step (2) and (3) are in $O(\frac{t(Q,G)}{n})$ and $O(n)$, respectively, *i.e.,* are parallel scalable.

Hence it suffices to focus on the parallel scalability of step (1), by proving Lemma 16 and Lemma 17(2).

Lemma 16 (Parallel scalability of DPar*)*. We first show that procedure DPar is parallel scalable. We will show the approximation ratio of DPar separately in the next proof.

Observe the following. (1) For each worker S_i managing a fragment F_i in the base partition, the border nodes $F_i.O$ can be computed via a linear scan of F_i. Hence the overall time is in $O(|C_d * \frac{|G|}{n}|)$ by the condition of Lemma 16. (2) Given $V' = \bigcup F_i.O$, DPar applies the (1+ε) approximation algorithm of [CK00], which computes an assignment in time $O(|V'|^{\frac{1}{\varepsilon}})$. For ε small enough for a good approximation, *e.g.,* ε=1, the time cost is $O(|V'|)$. Since $|V'| = \sum F_i.O \leq \sum N_d(v)$ ($v \in V'$), and $\sum N_d(v)$ is bounded by $O(C_d * \frac{|G|}{n})$, the process takes time in $O(C_d * \frac{|G|}{n}) = O(\frac{|G|}{n})$. (3) For each border node $v \in F_i.O$, each S_i fetches $N_d(v)$ from G in parallel. In the worst case, each worker takes in total $O(\sum |N_d(v)|)$ time for all the border nodes $v \in F_i.O$. As the fetch process is bounded by $O(C_d * \frac{|G|}{n})$ at each fragment, the overall parallel partition time is bounded

by $O(C_d * \frac{|G|}{n})$=$O(\frac{|G|}{n})$. Hence, DPar is parallel scalable.

Lemma 17(2) (Parallel scalability of mQMatch). Procedure mQMatch is conducted locally in parallel at each worker. From the correctness of Lemma 17(1), each worker only performs local matching without the need to communicate with others once DPar terminates. Hence, the overall time complexity is $O(\frac{t(Q,G)}{n})$. The time cost for merging the answers is in $O(n)$ time. Putting these together, PQMatch is in $O(\frac{t(Q,G)}{n})$+$O(n)$ time. Lemma 17(2) thus follows, and so does the parallel scalability of PQMatch. □

Remark. Algorithm PQMatch can be easily adapted to dynamic query load and graphs. (1) For a query with radius $d' > d$, each worker S_i incrementally computes $N_{d'-d}(v)$ for each node $v \in F_i.O$, via the balanced parallel BFS traversal. (2) When G is updated, coordinator S_c assigns the changes (*e.g.,* node/edge insertions and deletions) to each fragment. Each worker then applies incremental distance querying [FWW13] to maintain $N_d(v)$ of all affected $v \in F_i.O$ for $i \in [1,n]$.

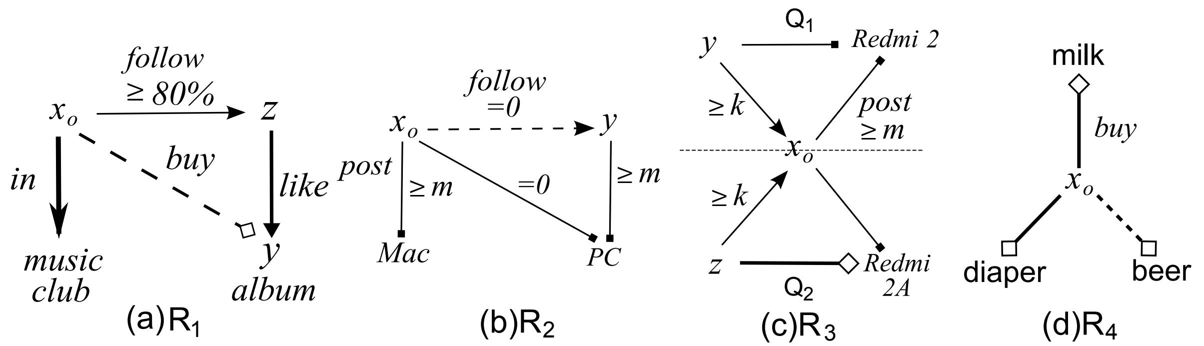


Figure 4.7: QGARs

4.5 Quantified Association Rules

As an application of QGPs, we introduce a set of graph association rules (QGARs) with counting quantifiers, to identify regularity between entities in graphs in general, and potential customers in social graphs in particular.

QGARs. A *quantified graph association rule* $R(x_o)$ is defined as $Q_1(x_o) \Rightarrow Q_2(x_o)$, where Q_1 and Q_2 are QGPs, referred to as the *antecedent* and *consequent* of R, respectively.

The rule states that for all nodes v_x in a graph G, if $v_x \in Q_1(x_o, G)$, then the chances are that $v_x \in Q_2(x_o, G)$.

Using QGPs, QGAR R can express *positive and negative* correlations [WZZ04] and social influence patterns with statistical significance [GBL08], which are useful in targeted advertising. (1) If Q_2 is a positive QGP, $R(x_o)$ states that if x_o satisfies the conditions in Q_1, then "event" Q_2 is likely to happen to x_o. For instance, $Q_2(x_o)$ may be a single edge buy(x_o, y) indicating that x_o may buy product y. In a social graph G, $R(x_o, G)$ identifies potential customers x_o of y. (2) When Q_2 is, *e.g.,* a single negated edge buy(x_o, y), $R(x_o)$ suggests that no v_x in $Q_1(x_o, G)$ will likely buy product y.

Example 22: A positive QGAR $R_1(x_o)$: $Q_1(x_o) \Rightarrow$ buy(x_o) is shown in Fig. 4.7, where Q_1 is the QGP given in Example 14, and Q_2 is a single edge buy(x_o) (depicted as a dashed edge). It states that if x_o is in a music club and if 80% of people whom x_o follows like an album y, then x_o will likely buy y.

A negative QGAR R_2 is also shown in Fig. 4.7, where Q_2 is a single negative edge follow(x_o, y). The QGAR states that if x_o and y actively ($\geq k$) tweet on competitive products (*e.g.,* "Mac" vs "PC"), then x_o is unlikely to follow y. Intuitively, R_2 demonstrates "negative" social influence [GBL08].

As another example, R_3 of Fig. 4.7 is a rule in which Q_2 consists of multiple nodes. Here Q_1 in R_3 specifies users x_o who actively promote mobile phone Redmi 2 and influence other users; and Q_2 predicts the impact of x_o on other users for a new release Redmi 2A. Putting these together, R_3 states that if x_o is influential over an earlier version, then x_o is likely to promote the selling of a new release [AJ08]. Intuitively, Q_1 identifies x_o as “leaders” [GBL08], who are often targeted by companies for promotion of a product series [AJ08].

To the best of our knowledge, these QGARs are not expressible as association rules studied so far (*e.g.,* [GTHS13, FWWX15]).

QGARs also naturally express conventional association rules defined on itemsets. For instance, $\mathsf{milk}, \mathsf{diaper} \Rightarrow \mathsf{beer}$ is depicted as QGAR $R_4(x_o)$ in Fig. 4.7. It finds customers x_o who, if buy milk and diaper, are likely to purchase beer. □

For real-world applications (*e.g.,* social recommendation), we consider practical and nontrivial QGARs by requiring: (a) Q_1 and Q_2 are connected and *nonempty* (*i.e.,* each of them has at least one edge); and (b) Q_1 and Q_2 do not overlap, *i.e.,* they do not share a common edge. We treat R as a QGP composed of both Q_1 and Q_2 such that in a graph G,

$$R(x_o, G) = Q_1(x_o, G) \cap Q_2(x_o, G).$$

Interestingness measure. To identify interesting QGARs, we define the support and confidence of QGARs.

Support. Given a QGAR $R(x_o)$ and a graph G, the *support* of R in G, denoted as $\mathsf{supp}(R, G)$, is the size $|R(x_o, G)|$, *i.e.,* the number of matches in $Q_1(x_o, G) \cap Q_2(x_o, G)$. We justify the support with the result below, which shows its *anti-monotonicity* for both pattern topology *and* quantifiers.

Lemma 18: *For any extension R' of R by (1) adding new edges (positive or negative) to Q_1 or Q_2, or (2) increasing p in positive quantifiers,* $|\mathsf{supp}(R', G)| \leq |\mathsf{supp}(R, G)|$. □

Confidence. We follow the local close world assumption (LCWA) [Don14], assuming that graph G is locally complete, *i.e.,* either G includes the complete neighbors of a node for any known edge type, or it has no information about these neighbors. We define the confidence of $R(x_o)$ in G as

$$\mathsf{conf}(R, G) = \frac{|R(x_o, G)|}{|Q_1(x_o, G) \cap X_o|},$$

where X_o is the set of candidates of x_o that are associated with an edge of the same type for every edge e=(x_o, u) in Q_2. Intuitively, X_o retains those "true" negative examples under LCWA, *i.e.,* those that have every required relationship of x_o in Q_2 but are not a match.

One might be tempted to define the confidence of $R(x_o)$ as $\frac{|R(x_o,G)|}{|Q_1(x_o,G)|}$, following traditional association rules [SA96]. However, this does not work well in incomplete graphs.

Example 23: For QGAR R_1, consider two matches v_1 and v_2 in $Q_1(x_o, G)$, where user v_1 has *no* edge labeled buy, and v_2 has a buy edge connected to a book. Since G is usually incomplete, it is an overkill to assume that v_1 is a negative example as a potential customer of books, since some of its buy edges may possibly be missing from G. □

To accommodate incomplete graphs, we follow the local close world assumption (LCWA) [Don14], which assumes that G is locally complete, *i.e.,* either G includes the complete neighbors of a node for any existing edge type, or it knows nothing about the neighbors. We define $\mathsf{conf}(R, G)$, the confidence of $R(x_o)$ in G under LCWA, as $\frac{|R(x_o,G)|}{|Q_1(x_o,G) \cap X_o|}$.

Continuing with Example 23, user v_2 is retained in X_o but v_1 is excluded due to missing buy edges. Hence, v_1 is no longer considered to be a negative match under LCWA.

Quantified entity identification. We want to use QGARs to identify entities of interests that match certain behavior patterns specified by QGPs. To this end, we define the set of entities identified by a QGAR $R(x_o)$ in a (social or knowledge) graph G with confidence η as follows:

$$R(x_o, \eta, G) = \{v_x \mid v_x \in R(x_o, G), \mathsf{conf}(R, G) \geq \eta\},$$

i.e., entities identified by R if its confidence is above η.

We study the *quantified entity identification* (QEI) problem: Given a QGAR $R(x_o)$, graph G, and a confidence threshold $\eta > 0$, it is to find all the entities in $R(x_o, \eta, G)$.

The QEI problem is DP-hard, as it embeds the quantified matching problem, which is DP-hard (Theorem 9). However, the (parallel) quantified matching algorithms for QGPs can be extended to QEI, *without* incurring substantial extra cost. Denote as $t(|Q|, |G|)$ the cost for quantified matching of QGP Q in G. Then we have the following.

Corollary 19: *There exist (1) an algorithm to compute $R(x_o, \eta, G)$ in $O(t(|R|, |G|))$ time; and (2) a parallel scalable algorithm to compute $R(x_o, \eta, G)$ in $O(\frac{t(|R|,|G|)}{n} + n)$ time with n processors, under the condition of Theorem 15.* □

Proof: As a constructive proof, we outline two algorithms for computing $R(x_o, \eta, G)$ with the desired complexity as follows.

<u>*Sequential quantified entity matching*</u>. Given a QGAR R, confidence threshold η and G, the first algorithm, denoted as garMatch, (1) invokes QMatch to compute $Q_1(x_o, G)$ and $Q_2(x_o, G)$, respectively; (2) computes $R(x_o, G) = Q_1(x_o, G) \cap Q_2(x_o, G)$; and (3) verifies whether $\text{conf}(R) = \frac{|R(x_o,G)|}{|Q_1(x_o,G) \cap X_o|} \geq \eta$. If so, it returns $R(x_o, G)$.

The correctness and complexity of garMatch follow from their QMatch counterparts (Lemmas 13 and 14). That is, garMatch is in $O(t(|R|, |G|))$ time, where $t(|R|, |G|)$ is the complexity of a quantified matching algorithm (QMatch).

<u>*Parallel quantified entity matching*</u>. We introduce an algorithm, denoted as dgarMatch, for parallel quantified entity matching. It follows the generic steps of PQMatch. The only difference is as follows: (a) each worker evaluates two patterns Q_1 and Q_2 in parallel, and (b) the coordinator S_c assembles the results to evaluate the confidence of R.

Algorithm dgarMatch starts with a set of base partitions. (1) It constructs a d-hop preserving partition, where d is a predefined upper bound of the largest radius Q_1 and Q_2 in R. (2) Each worker then computes local match $Q_1(x_o, F_i)$ and $Q_2(x_o, F_i)$ in parallel. It also computes the local set X_{oi}. (3) Each worker returns the local matches to the coordinator S_c. Then dgarMatch computes $R(x_o, G)$ as $(\bigcup Q_1(x_o, F_i)) \setminus (\bigcup Q_2(x_o, F_i))$, and computes the confidence $\text{conf}(R, G)$ as $\frac{|R(x_o,G)|}{|\bigcup Q_1(x_o,F_i) \cap \bigcup X_{oi}|}$. It next verifies whether $\text{conf}(R, G) \geq \eta$ and if so, returns $R(x_o, G)$. Otherwise, it returns $\emptyset$.

The correctness and complexity of dgarMatch follow from their PQMatch counterparts. More specifically, (1) dgarMatch takes $O(\frac{t(|R|,|G|)}{n} + n)$ time to compute the local matches for Q_1 and Q_2 in parallel, and (2) the verification of the confidence is in $O(n)$ time. Hence, dgarMatch computes $R(x_o, \eta, G)$ in $O(\frac{t(|R|,|G|)}{n} + n)$ time using n processors. It is thus parallel scalable by definition. □

4.6 Experimental Study

We conducted three sets of experiments to evaluate (1) the scalability and (2) parallel scalability of our quantified matching algorithms, and (3) the effectiveness of QGAR for identifying correlated entities in large real-world graphs.

Experimental setting. We used two real-life graphs: (a) Pokec [Pok], a social network with 1.63 million nodes of 269 different types, and 30.6 million edges of 11 types, such as *follow*, *like*; and (b) YAGO, an extended knowledge base of YAGO [SKW07] that consists of 1.99 million nodes of 13 different types, and 5.65 million links of 36 types.

We also developed a generator to produce synthetic social graphs $G = (V, E, L)$, controlled by the numbers of nodes $|V|$ (up to 50 million) and edges $|E|$ (up to 100 million), with L drawn from an alphabet $\mathcal{L}$ of 30 labels. The generator is based on GTgraph [BM] following the small-world model.

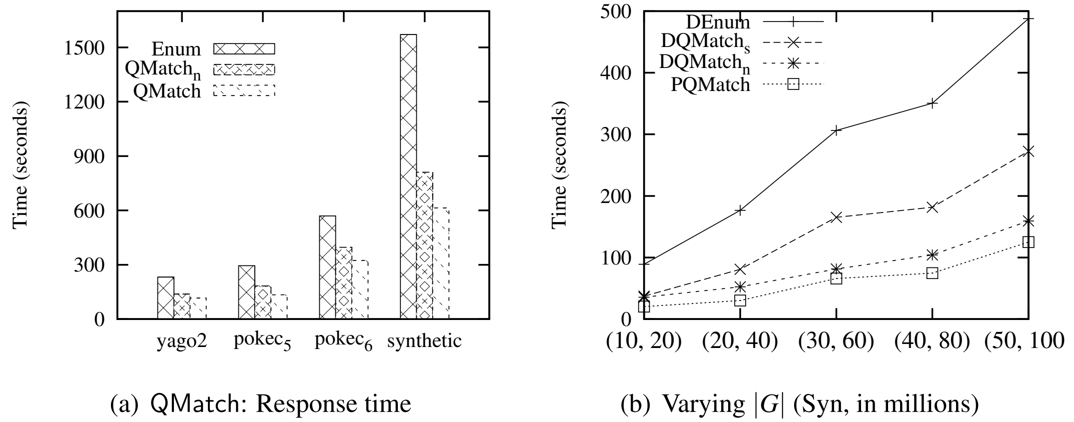
Pattern generator. For real-life graphs we generated QGPs Q controlled by $|V_Q|$ (size of pattern nodes), $|E_Q|$ (pattern edges), $p\%$ (in quantifiers) and $|E_Q^-|$ (size of negated edges). (1) We first mined frequent features, including edges and paths of length up to 3 on each of Pokec and YAGO. We selected top 5 most frequent features as "seeds", and combined them to form the stratified pattern Q_π of $|V_Q|$ nodes and $|E_Q|$ edges. (2) For frequent pattern edges e=(u, u'), we assigned a positive quantifier $\sigma(e) \geq p\%$, where $p\%$ is initialized as 30% unless otherwise specified. This completes the generation of $\Pi(Q)$. (3) We added $|E_Q^-|$ negated edges to $\Pi(Q)$ between randomly selected node pairs (u, u'), to complete the construction of Q. For synthetic graphs, we generated 50 QGPs with labels drawn from $\mathcal{L}$.

We denote by $|Q| = (|V_Q|, |E_Q|, p_a, |E_Q^-|)$ the size of QGP Q, where p_a is the average of p in all its quantifiers.

Algorithms. We implemented the following algorithms, all on GRAPE.

(1) Algorithm QMatch, versus (a) QMatch_n, a revision of QMatch that processes negated edges using DMatch, not the incremental IncQMatch, and (b) Enum, which adopts a state-of-the-art subgraph isomorphism algorithm [RW15] to enumerate all matches first, and then verify quantifiers.

(2) Algorithm PQMatch, versus (a) $\mathsf{PQMatch}_s$, its single-thread counterpart, (b) $\mathsf{PQMatch}_n$, the parallel version of QMatch_n, and (c) PEnum, a parallel version of Enum, which first invokes a parallel subgraph listing algorithm [SCC+14] to enumerate all matches, and



(a) QMatch: Response time

(b) Varying $|G|$ (Syn, in millions)

Figure 4.8: Response time and scalability for quentified match

then verifies quantifiers. We also implemented (d) DPar for d-hop preserving partition.

We deployed the parallel algorithms over n processors for $n \in [4,20]$. Each experiment was run 5 times and the average is reported here.

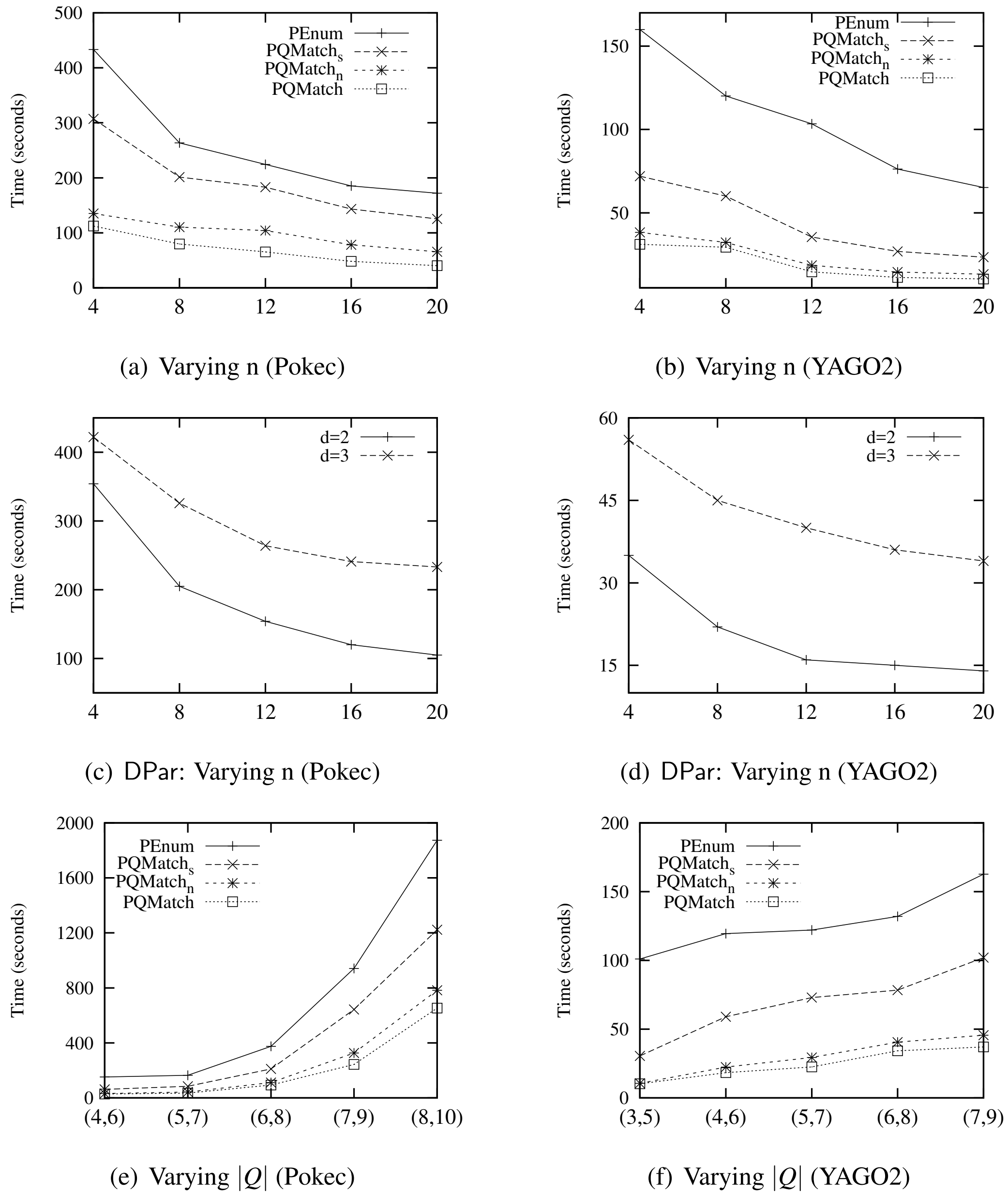
Experimental results. We next report our findings.

Exp-1: Performance of QMatch. We first evaluated the performance of QMatch compared with QMatch_n and Enum. Fixing $|Q|$=$(5,7,30\%,1)$, *i.e.,* patterns with 5 nodes and 7 edges, with p_a = 30% and one negative edge, Figure 4.8(a) reports the performance of QMatch over two real-world graphs Pokec and YAGO, and a larger synthetic graph G_s of 50 million nodes and 100 million edges. We find the following. (1) QMatch outperforms the other algorithms. It is on average 1.2 and 2.0 times faster than QMatch_n and Enum over YAGO, 1.3 and 2.0 times faster over Pokec, and 1.3 and 2.6 times faster over G_s, respectively. This verifies that our optimization strategies effectively reduce the verification cost.

(2) QMatch works reasonably well over real-world social and knowledge graphs. It takes up to 150 (resp. 116) seconds over Pokec (resp. YAGO), comparable to conventional subgraph isomorphism without quantifiers.

Exp-2: Scalability of PQMatch. This set of experiments evaluated the scalability of parallel algorithm PQMatch, compared to PQMatch_n, PQMatch_s, and PEnum. In these experiments, we fixed $|Q| = (6,8,30\%,1)$, $d = 2$ for d-hop preserving partition and $b = 4$ for the number of threads in intra-fragment parallelism, unless stated otherwise.

*Varying n (*PQMatch*).* We varied the number n of processors from 4 to 20. As shown



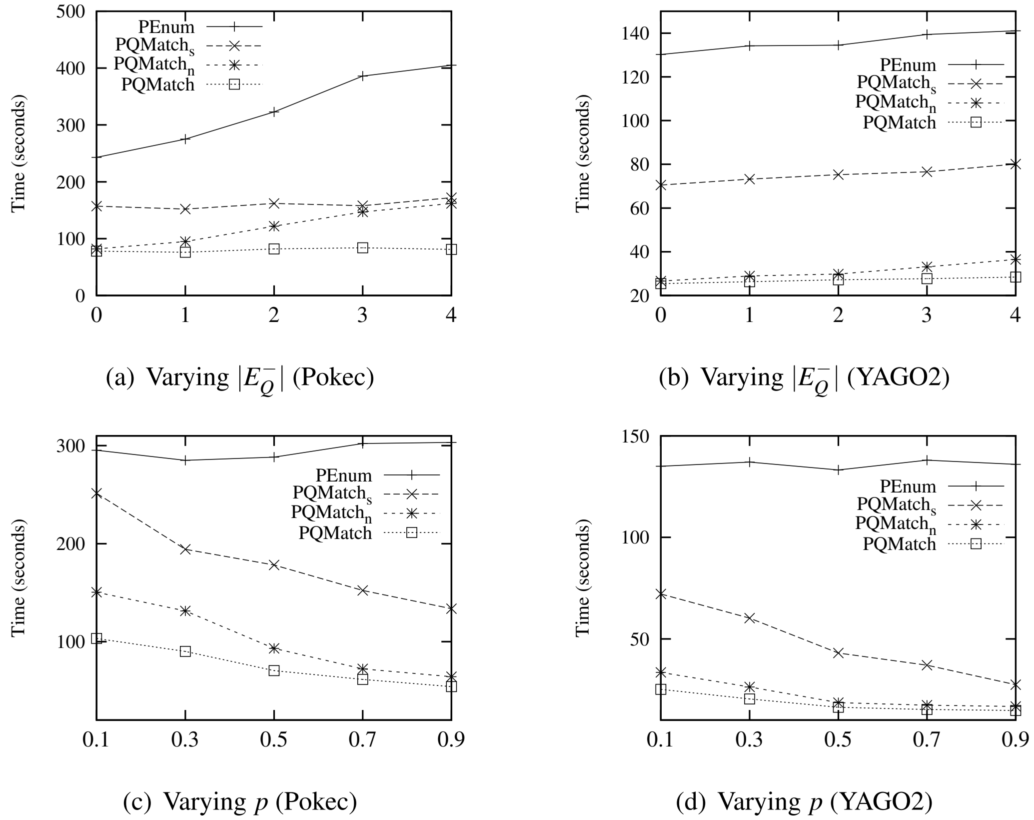
(a) Varying n (Pokec) (b) Varying n (YAGO2)

(c) DPar: Varying n (Pokec) (d) DPar: Varying n (YAGO2)

(e) Varying $|Q|$ (Pokec) (f) Varying $|Q|$ (YAGO2)

Figure 4.9: Parallel scalability of PQMatch

in Fig. 4.9(a) (resp. Fig. 4.9(b)) over Pokec (resp. YAGO), (1) PQMatch and PQMatch$_s$ scale well with the increase of processors: for PQMatch, the improvement is 2.8 (resp. 3.2) times when n increases from 4 to 20; this verifies Theorem 15; (2) PQMatch is 3.8 (resp. 5.8) times faster than PEnum; and (3) with optimization strategies (incremental evaluation and multi-threads), PQMatch outperforms PQMatch$_n$ and PQMatch$_s$ by 1.5 (resp. 1.1) times and 2.8 (resp. 2.3) times, respectively. (4) PQMatch works reasonably well on large graphs. With 20 processors, it takes 40.3 (resp. 10.2) seconds on Pokec (resp. YAGO).



(a) Varying $|E_Q^-|$ (Pokec) (b) Varying $|E_Q^-|$ (YAGO2)

(c) Varying p (Pokec) (d) Varying p (YAGO2)

Figure 4.10: Impact of negative edges and aggregate on PQMatch

Varying n (DPar). We also evaluated the scalability of DPar for d-hop preserving partition, with $d = 2$ and $d = 3$. As shown in Figures 4.9(c) and 4.9(d), (1) DPar scales well with n: when d=2, the improvement is 3.5 (resp. 2.5) times when n increases from 4 to 20 over Pokec (resp. YAGO).

(2) The fragments are well balanced: the "skew" (the ratio of the size of the smallest fragment to the largest one) is at least 80% when n=8, for both Pokec and YAGO.

These justify the parallel scalability of DPar andPQMatch.

Varying $|Q|$. Fixing p_a = 30%, $|E_Q^-|$ = 1 and n = 8, we varied $(|V_Q|, |E_Q|)$ from (4, 6) to (8, 10) (resp. (3, 5) to (7, 9)) on Pokec (resp. YAGO). As shown in Figures 4.9(e) and 4.9(f), (1) the larger $|Q|$ is, the longer time is taken by all the algorithms, as expected. (2) PQMatch works well on real-life queries. For queries with 5 nodes and 7 edges (close to real-world queries), it takes up to 35 (resp. 16.3) seconds over Pokec (resp. YAGO). It works better on sparse YAGO than on Pokec. (3) PQMatch outperforms the other algorithms, consistent Figures 4.9(a) and 4.9(b).

Varying $|E_Q^-|$. We also studied the impact of the number of negated edges. The purpose

of this test is to evaluate the effectiveness of incremental matching strategy IncQMatch.

Fixing $n = 8$, $(|V_Q|,|E_Q|) = (6,8)$ and $p_a = 30\%$, we varied $|E_Q^-|$ from 0 to 4 by selecting $|E_Q^-|$ edges e and "negating" them by setting $\sigma(e) = 0$. As shown in Figures 4.10(a) and 4.10(b), (1) PQMatch and $\mathsf{PQMatch}_s$ are rather indifferent to the change of $|E_Q^-|$, which incurs small extra cost due to their incremental strategy (IncQMatch).

(2) In contrast, $\mathsf{PQMatch}_n$ and PEnum are more sensitive to the increment of $|E_Q^-|$. Both algorithms, without IncQMatch, always recompute the matches of pattern Q^{+e} for each negated edge $e \in E_Q^-$, and hence take more time over larger $|E_Q^-|$. The improvement of PQMatch over $\mathsf{PQMatch}_n$ and PEnum becomes more significant (from 1.1 to 2 times, and 3.1 to 5 times) with larger $|E_Q^-|$ (from 1 to 4) over Pokec. These results verify the effectiveness of IncQMatch.

Varying p_a. Fixing n=8, $|E_Q^-| = 1$ and $(|V_Q|,|E_Q|) = (6,8)$ (resp. (5, 7)) for Pokec (resp. YAGO), we evaluated the impact of aggregates by varying p_a from 10% to 90%. As shown in Figures 4.10(c) and 4.10(d), (1) with larger p_a, PQMatch, $\mathsf{PQMatch}_s$ and $\mathsf{PQMatch}_n$ take less time, since more candidates are pruned in the verification process. (2) In contrast, PEnum is indifferent to the change of p_a, since it always enumerates all the matches regardless of p_a. This verifies the effectiveness of the pruning strategies of PQMatch.

Observe that PQMatch is less sensitive than $\mathsf{PQMatch}_n$ to p_a. When p_a is small, the overhead of $\mathsf{PQMatch}_n$ incurred by the recomputation of Q^{+e} for negated edges e is larger, since a large number of candidates need to be verified. With larger p_a (more strict quantifiers), the overhead reduces due to the effective pruning of candidates by $\mathsf{PQMatch}_n$. This explains the comparable performance of PQMatch and $\mathsf{PQMatch}_n$ when p_a is large (*e.g.*, p_a=0.9).

Varying $|G|$. Fixing $n = 4$, we varied $|G|$ from $(10M,20M)$ to $(50M,100M)$ using synthetic social graphs. As shown in Fig. 4.8(b), (1) PQMatch scales well with $|G|$ and is feasible on large graphs. It takes 125 seconds when $|G| = (50M,100M)$. (2) PQMatch is 1.5, 2.3 and 4.7 times faster than $\mathsf{PQMatch}_n$, $\mathsf{PQMatch}_s$ and PEnum on average, respectively.

Exp-3: Effectiveness of QGAR. We also evaluated the effectiveness of QGARs. We developed a simple QGAR mining algorithm by extending the algorithm of [FWWX15] for mining graph pattern association rule (GPARs). GPARs are a special case of QGARs $Q_1(x_o) \Rightarrow Q_2(x_o)$ that have no quantifiers and restrict Q_2 to a single edge. (1) We mined

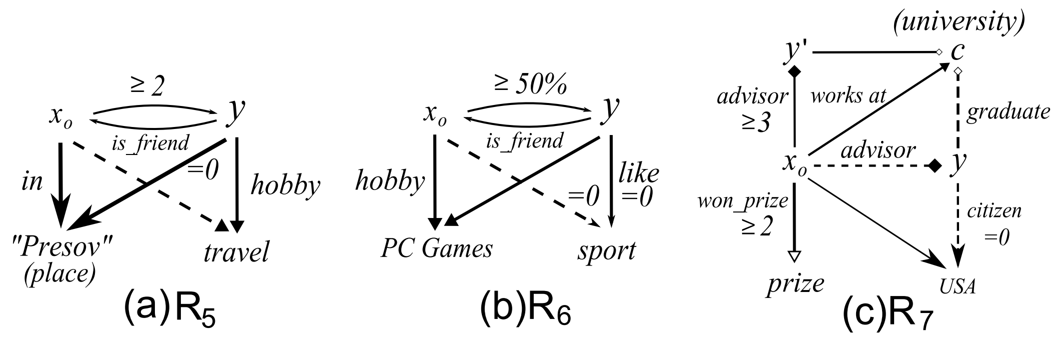


Figure 4.11: Real-world QGARs

a set of top GPARs using [FWWX15] over Pokec and YAGO, for confidence threshold $\eta = 0.5$. For each GPAR R, we initialized a QGAR R'. (2) We extended Q_2 in each R' by adding frequent edges whenever possible, and by gradually enlarging p_a for frequent edges by increment 10% (1 for numeric aggregates). We stopped when the confidence of R' got below η. We show three QGARs in Fig. 4.11, illustrated as follows.

(1) R_5 (Pokec) says that if a user has "long-distance" friends, *i.e.,* at least two of her friends do *not* live in the same city "`Presov`" where she lives, then the chances are that they share the hobby of traveling. We found 50 matches in Pokec.

(2) R_6 (Pokec; confidence 0.8) demonstrates a negative pattern: for a user x_o, if more than half of his friends share the same hobby "`PC Games`", and none of them like sports, then it is likely x_o does not like sports. R_6 has support 4000.

(3) R_7 (YAGO; confidence 0.75) states that if a US professor (a) won at least two academic prizes, and (b) graduated at least 4 students, then the chances are that at least one of her/his students is not a US citizen. It discovers scientists such as Marvin Minsky (Turing Award 1969) and Murray Gell-Mann (Nobel Prize Physics 1969) from YAGO. Here Q_2 in R_7 has three (dashed) edges, as opposed to GPARs [FWWX15].

These QGARs demonstrate quantified correlation between the entities in social and knowledge graphs, which cannot be captured by conventional association rules and GPARs [FWWX15].

Summary. We find the following. Over real-life graphs, (1) quantified matching is feasible: PQMatch (with 20 processors) and QMatch took 40.3s and 342s on Pokec, and 10.2s and 116s on YAGO, respectively. (2) Better still, PQMatch and DPar are parallel scalable: their performance is improved by 3 times on average with workers increased from 4 to 20. (3) Our optimization techniques improve the performance of

QMatch and PQMatch by 1.27 and 1.3 times on average, and 2.2 and 4.5 times over Enum and PEnum, respectively.

(4) QGARs capture behavior patterns that cannot be expressed with conventional graph patterns.

4.7 Related Work

We categorize the related work to this chapter as follows.

Quantified graph querying. The need for counting in graph queries has long been recognized. SPARQLog [LLP10] extends SPARQL with first-order logic (FO) rules, including existential and universal quantification over node variables. Rules for social recommendation are studied in [LAR00], using support count as constraints. QGRAPH [BIJ02] annotates nodes and edges with a counting range (count 0 as negated edge) to specify the number of matches that must exist in a database. Set regular path queries (SRPQ) [LSZD13] extends regular path queries with quantification for group selection, to restrict the nodes in one set connected to the nodes of another. For social networks, SocialScope [AYLY09] and SNQL [SMGW11] are algebraic languages with numeric aggregates on node and edge sets.

The study of QGPs is to strike a balance between the expressivity and the complexity. It differs from the prior work in the following. (1) Using a uniform form of counting quantifiers, QGPs support numeric and ratio aggregates (*e.g.,* at least p friends and 80% of friends), and universal (100%) and existential quantification (≥ 1). In contrast, previous proposals do not allow at least one of these. (2) We focus on graph pattern queries, which are widely used in social media marketing and knowledge discovery; they are beyond set regular expressions [LSZD13] and rules of [LAR00]. (3) Quantified matching with QGPs is DP-complete at worst, slightly higher than conventional matching (NP-complete) in the polynomial hierarchy [Pap03]. In contrast, SPARQL and SPARQLog are PSPACE-hard [LLP10], and SRPQ takes EXPTIME [LSZD13]; while the complexity bounds for QGRAPH [BIJ02], SocialScope [AYLY09] and SNQL [SMGW11] are unknown, they are either more expensive than QGPs, (*e.g.,* QGRAPH is a fragment of FO(count)), or cannot express numeric and ratio quantifiers [AYLY09, SMGW11]. (4) No prior work has studied parallel scalable algorithms for its queries.

Parallel pattern matching. A number of (parallel) matching algorithms have been developed for subgraph isomorphism [RW15, HAR11, LHKL12]. None of these addresses quantifiers. In contrast, (1) in the same general framework [LHKL12] used by these methods, our sequential quantified matching algorithms cope with quantifiers and negated edges without incurring considerable cost; and (2) our parallel scalable algorithms exploit both inter and intra-fragment parallelism for effective quantifier verification in QGP evaluation.

Various strategies have been studied for graph partition [Kar11, AR06, BLV14]. This work differs from the prior work in the following. (1) We propose a d-hop preserving partition scheme such that the d-hop neighbor of each node is contained in a fragment, and that all fragments have an even size, with an approximation bound. Closest to ours is the n hop-guarantee partition [HAR11]. However, [HAR11] provides no approximation bound to ensure both d-hop preserving and balanced fragment sizes, especially for nodes with a high degree.

(2) We propose a partition algorithm that is parallel scalable, a property that is not guaranteed by the prior strategies.

Quantified association rules. Association rules [AIS93] are traditionally defined on relations of transaction data. Over relations, quantified association rules [SA96] and ratio rules [KLKF98] impose value ranges or ratios (*e.g.,* the aggregated ratio of two attribute values) as constraints on *attribute values*. There has also been recent work on extending association rules to social networks [SHJS06, LAR00] and RDF knowledge bases, which resorts to mining conventional rules and Horn rules (as conjunctive binary predicates) [GTHS13] over tuples with extracted attributes from social graphs, instead of exploiting graph patterns. Closer to this work is [FWWX15], which defines association rules directly with patterns without quantifiers.

Our work on QGARs differs from the previous work in the following. (1) As opposed to [AIS93, SA96, KLKF98], QGARs extend association rules from relations to graphs. They call for topological support and confidence metrics, since the conventional support metric is not anti-monotonic in graphs. (2) QGARs allow simple yet powerful counting quantifiers to be imposed on matches of graph patterns, beyond attribute values. In particular, rules of [SA96, KLKF98] cannot express universal quantification and negation. When it comes to graphs, (3) the rules of [FWWX15] cannot express counting quantifiers, and limits their consequent to be a single edge, and (4) applying QGPs and QGARs becomes an intractable problem, as opposed to PTIME for conventional rules in relations.

4.8 Summary

In this chapter, we have proposed quantified matching, by extending traditional graph patterns with counting quantifiers. We have also studied important issues in connection with quantified matching, from complexity to algorithms to applications. The novelty of this chapter consists in quantified patterns (QGPs), quantified graph association rules (QGARs), and algorithms with provable guarantees (*e.g.,* optimal incremental matching and parallel scalable matching). Our experimental study has verified the effectiveness of QGPs and the feasibility of quantified matching in real-life graphs.

Quantified graph pattern matching opens new areas for the applications on GRAPE. With its expressive power, GRAPE is able to resolve more complex problems such as accurately entity identification, customer recommendations.

Chapter 5

Functional Dependencies on Graphs

To make practical use of big data on GRAPE, we have to cope with not only its quantity but also its quality. Query answers computed upon dirty data may not be correct and even do more harm than good.

To catch inconsistencies in graphs, we propose a class of functional dependencies for graphs in this chapter. We give the formulation and settle the classical problems for reasoning about GFDs. In addition, we make use of GFDs to catch errors in real-life graphs to verify its effectiveness.

Data dependencies have been well studied for relational data. In particular, our familiar functional dependencies (FDs) are found in every database textbook, and have been extended to XML [AL04]. Their revisions, such as conditional functional dependencies (CFDs) [FGJK08], have proven effective in capturing semantic inconsistencies in relations [FG12].

The need for FDs is also evident in graphs, a common source of data. Unlike relational databases, real-life graphs typically do not come with a schema. FDs specify a fundamental part of the semantics of the data, and are hence particularly important to graphs. Moreover, (1) FDs help us detect inconsistencies in knowledge bases [SSW09], which need to be identified as violations of dependencies [FG12]. (2) For social networks, FDs help us catch spams and manage blogs [CSYP12].

Example 24: Consider the following examples taken from real-life knowledge bases and social graphs.

(1) *Knowledge bases*, where inconsistencies are common [SSW09]:

- Flight A123 has two entries with the same departure time 14:50 and arrival time 22:35, but one is from Paris to NYC, while the other from Paris to Singapore [ZRM+13].
- Both Canberra and Melbourne are labeled as the capital of Australia [ECD+04].
- It is marked that all birds can fly, and that penguins are birds [HLH12], despite their evolved wing structures.

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

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

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

This is essentially an FD on graph-structured data.

Functional dependencies are also useful in catching spams.

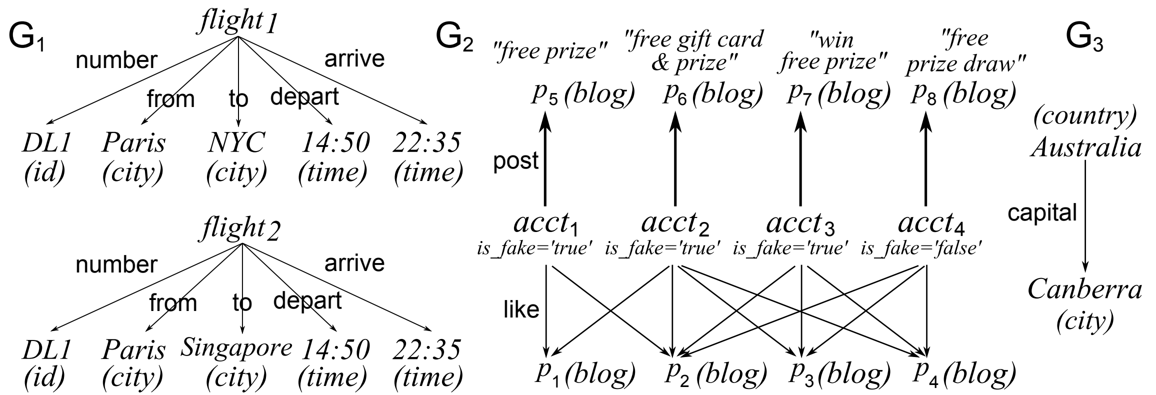


Figure 5.1: Graphs with dependencies

- Fake account [CSYP12]: If account x' is confirmed fake, both accounts x and x' like blogs $P_1, \ldots, P_k$, x posts blog y, x' posts y', and if y and y' have a particular keyword c, then x is also identified as a fake account.

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

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

5.1 Preliminaries

We start with a review of basic notations.

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

Example 25: Three graphs are depicted in Fig. 5.1: (a) G_1 is a fragment of a knowledge graph, where each flight entity (*e.g.,* flight_1) has id (with value val = DL1), departure city (Paris), destination (NYC), and departure and arrival time; each node has attribute val (not shown) for its value; (b) G_2 records fake accounts; each account has an attribute is_fake that is "true" if the account is fake, and "false" otherwise; an account may post blogs that contain keywords (*e.g.,* blog p_5 has attribute keyword = "free prize"), and may like other blogs; and (c) G_3 depicts a country entity and its capital, carrying attribute val (not shown) for their values. □

We review two notions of subgraphs.

- A graph $G' = (V', E', L', F'_A)$ is a *subgraph of* $G = (V, E, L, F_A)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$, $L'(v) = L(v)$ and $F'_A(v) = F_A(v)$; similarly for each edge $e \in E'$, $L'(e) = L(e)$.
- We say that G' is *a subgraph induced by* a set V' of nodes if $G' \subseteq G$ and E' consists of all the edges in G whose endpoints are both in V'.

Graph patterns. A *graph pattern* is defined as a directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (resp. E_Q) is a set of pattern nodes (resp. edges), (2) L_Q is a function that assigns a label $L_Q(u)$ (resp. $L_Q(e)$) to each pattern node $u \in V_Q$ (resp. edge $e \in E_Q$), (3) $\bar{x}$ is a list of variables such that its arity $\|\bar{x}\|$ is equal to the number $|V_Q|$ of nodes, and (4) μ is a bijective mapping from $\bar{x}$ to V_Q, *i.e.,* it assigns a distinct variable to each node v in V_Q. For $x \in \bar{x}$, we use $\mu(x)$ and x interchangeably when it is clear in the context.

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

Example 26: Figure 5.2 depicts six graph patterns Q_1–Q_6: (1) Q_1 specifies two flight

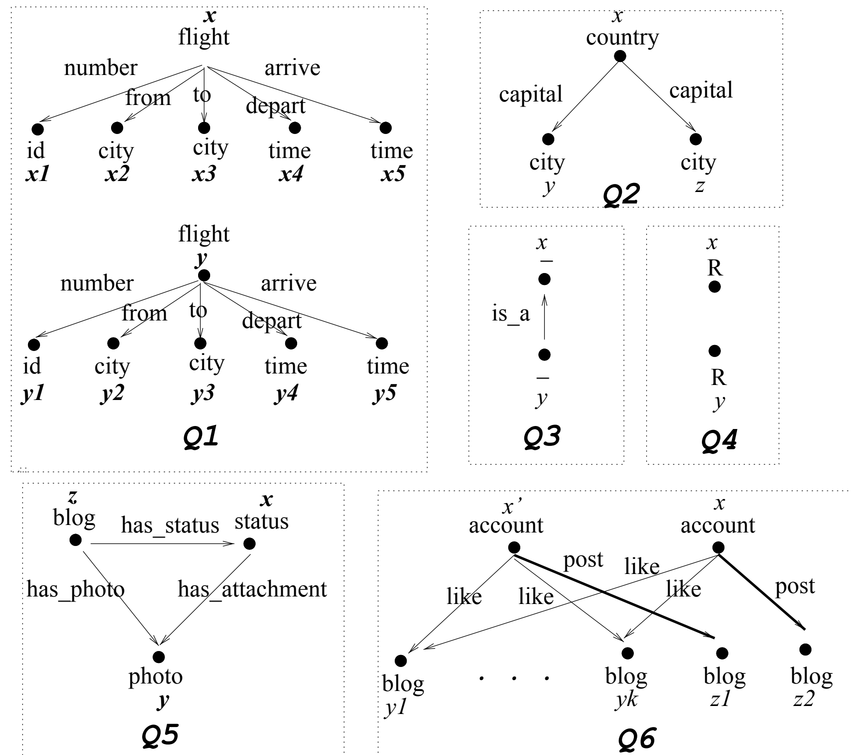


Figure 5.2: Graph patterns

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

Graph pattern matching. We adopt the conventional semantics of matching via subgraph isomorphism. A *match* of pattern Q in graph G is a subgraph $G' = (V', E', L', F'_A)$ of G that is isomorphic to Q, *i.e.,* there exists a *bijective function* h from V_Q to V' such that (1) for each node $u \in V_Q$, $L_Q(u) = L'(h(u))$; and (2) $e = (u, u')$ is an edge in Q if and only if $e' = (h(u), h(u'))$ is an edge in G' and $L_Q(e) = L'(e')$. In particular, $L_Q(u) = L'(h(u))$ always holds if $L_Q(u)$ is '_', *i.e.,* wildcard matches any label to indicate generic entities, *e.g.,* is_a in Q_3 of Example 26; similarly for edge labels.

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

$h(\bar{x})$ is such an instantiation in G, one node for each entity.

Example 27: A match of Q_1 of Example 26 in G_1 of Fig. 5.1 is h_1: $x \mapsto \mathsf{flight}_1$, $y \mapsto \mathsf{flight}_2$, $x_3 \mapsto$ NYC, $y_3 \mapsto$ Singapore, and similarly for the other variables in Q_1.

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

The notations of this chapter are summarized in Table 5.1.

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

Table 5.1: Notations in Chapter 5

5.2 GFDs: Syntax and Semantics

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

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

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

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

Intuitively, GFD φ specifies two constraints:

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

Recall that the "scope" of a relational FD $R(X \rightarrow Y)$ is specified by a relation schema R: the FD is applied only to instances of R. Unlike relational databases, graphs do not have a schema. Here Q specifies the scope of the GFD, such that the dependency $X \rightarrow Y$ is imposed only on the attributes of the vertices in each subgraph identified by Q. Constant literals $x.A = c$ enforce bindings of semantically related constants, along the same lines as CFDs [FGJK08].

Example 28: To catch the inconsistencies described in Example 24, we define GFDs with patterns Q_1–Q_6 of Fig. 5.2.

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

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

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

In particular, if x is labeled with bird, y with penguin, and A is can_fly, then φ_3 catches the inconsistency described in Example 24: penguins cannot fly but are classified as bird.

(4) FDs *and* CFDs. Consider an FD $R(X \rightarrow Y)$ over a relation schema R [AHV95]. When an instance of R is represented as a graph in which each tuple is denoted by a node labeled R, we write $\varphi_4 = (Q_4[x,y], X' \rightarrow Y')$. Here Q_4 consists of two vertices x and y denoting two tuples of R, X' consists of $x.A = y.A$ for all $A \in X$, and Y' includes $x.B = y.B$ for all $B \in Y$. Note that φ_4 is defined with variable literals only.

Using constant literals, GFDs can express CFDs [FGJK08]. For instance, $R(\text{country} = 44, \text{zip} \rightarrow \text{street})$ is a CFD defined on relation R, stating that in the UK, zip code uniquely determines street [FGJK08]. It can be written as GFD $\varphi_4' = (Q_4[x,y], X' \rightarrow Y')$, where X' consists of $x.\text{country} = 44$, $y.\text{country} = 44$, and $x.\text{zip} = y.\text{zip}$, and Y' is $x.\text{street} = y.\text{street}$.

As another example, CFD $R(\text{country} = 44, \text{area_code} = 131 \rightarrow \text{city} = \text{Edi})$ states that in the UK, if the area code of a city is 131, then the city is Edi [FGJK08]. It can be expressed as a GFD $\varphi_4'' = (Q_4''[x], X'' \rightarrow Y'')$, where Q_4'' consists of a single node x labeled R, and X'' includes $x.\text{country} = 44$ and $x.\text{area_code} = 131$, while Y'' is $x.\text{city} = \text{Edi}$.

(5) Blogs: $\varphi_5 = (Q_5[x,y,z], \emptyset \rightarrow x.\text{text} = y.\text{desc})$. It states that if entities x, y and z satisfy the topological constraint of Q_5 depicted in Fig. 26, then the annotation of status x of blog z must match the description of photo y included in z.

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

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

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

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

Example 29: Consider GFDs φ_1, φ_2 and φ_6 of Example 28 and G_1, G_2, G_3 of Fig. 5.1. One can verify the following.

(a) $G_1 \not\models \varphi_1$. Indeed, the match h_1 given in Example 6 satisfies X_1 since $h_1(x_1).\mathsf{val} = h_1(y_1).\mathsf{val}$, but it does not satisfy Y_1 since $h_1(x_3).\mathsf{val} \neq h_1(y_3).\mathsf{val}$. Similarly, $G_2 \not\models \varphi_6$, as witnessed by match h_2 of Example 6. Note that there are other matches of Q_6 in G_2 that satisfy $X_6 \to Y_6$, *e.g.,* when we map $x' \mapsto \mathsf{acct}_1$ and $x \mapsto \mathsf{acct}_2$, However, $G_2 \models \varphi_6$ only if *all* matches of Q_6 in G_2 satisfy $X_6 \to Y_6$.

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

Observe the following: (a) entities in the same match of Q may be far apart; *e.g.,* flight_1 and flight_2 are *disconnected* from each other; and (b) $X \to Y$ is imposed only on matches of Q (satisfying its topological constraint), *e.g.,* φ_2. □

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

Special cases. GFDs subsume the following special cases.

(1) As shown by φ_4, φ_4' and φ_4'' in Example 28, relational FDs and CFDs are special cases of GFDs, when tuples in a relation are represented as nodes in a graph. In fact, GFDs are able to express equality-generating dependencies (EGDs) [AHV95].

(2) A GFD $(Q[\bar{x}], X \to Y)$ is called a *constant* GFD if X and Y consist of constant literals of $\bar{x}$ only. It is called a *variable* GFD if X and Y consist of variable literals only. Intuitively, constant GFDs subsume constant CFDs [FGJK08], and variable GFDs are analogous to traditional FDs [AHV95].

In Example 28, φ_1-φ_5 are variable GFDs, φ_4'' and φ_6 are constant GFDs, while φ_4' is neither constant nor variable.

(3) GFDs can specify certain type information. For an entity x of type τ, GFD $(Q[x], \emptyset \to$

$x.A = x.A$) enforces that x must have an A attribute, where Q consists of a single vertex labeled τ and denoted by variable x. However, GFDs *cannot* enforce that x has a finite domain, *e.g.,* Boolean.

5.3 Reasoning about GFDs

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

5.3.1 The Satisfiability Problem for GFDs

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

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

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

Over relational data, a set Σ of CFDs may not be satisfiable [FGJK08]. The same happens to GFDs on graphs.

Example 30: Consider two GFDs defined with the same pattern Q_7 depicted in Fig. 5.3: $\varphi_7 = (Q_7[x], \emptyset \to x.A = c)$ and $\varphi_7' = (Q_7[x], \emptyset \to x.A = d)$, where c and d are distinct constants. Then there exists no graph G that includes a τ entity v and satisfies both φ_7 and φ_7'. For if such a node v exists, then by φ_7, v has an attribute A with value c, while by φ_7', $v.A$ must take a different value d, which is impossible.

As another example, consider GFDs $\varphi_8 = (Q_8[x,y,z], \emptyset \to x.A = c)$ and $\varphi_9 = (Q_9[x,y,z,w], \emptyset \to x.A = d)$ for distinct c and d, where Q_8 and Q_9 are shown in Fig. 5.3. One can verify that each of φ_8 and φ_9 has a model, when taken alone. However, they are not satisfiable when put together. Indeed, if they have a model G, then there must exist isomorphic mappings h and h' from Q_8 and Q_9 to G, respectively, such that $h(x) = h'(x) = v$ for some node v in G. Then again, v is required to have attribute A with distinct values. □

As shown in Example 30, GFDs defined with different graph patterns may interact with each other. Indeed, Q_8 and Q_9 are different, but φ_8 and φ_9 can be enforced on the same node, since Q_8 is isomorphic to a subgraph of Q_9. This tells us that the satisfiability analysis has to check subgraph isomorphism among the patterns of the GFDs, which is NP-complete (cf. [Pap03]). In light of this, we have the following.

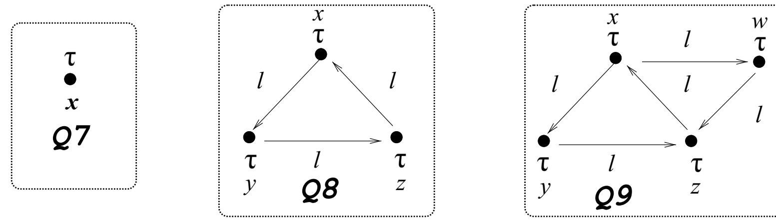


Figure 5.3: Graph patterns in GFDs

Theorem 20: *The satisfiability problem is* coNP*-complete for* GFDs. □

One might think that the problem would become simpler if Σ consists of constant GFDs only (see Section 5.2), or when all patterns in Σ are acyclic directed graphs (DAGs). However, the complexity bound is rather robust.

Corollary 21: *The satisfiability problem is* coNP*-complete for constant* GFDs *that are defined with DAG patterns.* □

The complexity of GFDs is not inherited from CFDs. Indeed, the satisfiability analysis of CFDs is NP-hard only under a schema that enforces attributes to have a *finite domain* [FGJK08], *e.g.,* Boolean, *i.e.,* when CFDs and finite domains are put together. In contrast, graphs do not come with a schema; while GFDs subsume CFDs, they cannot specify finite domains. That is, the satisfiability problem for GFDs is already coNP-hard in the absence of a schema.

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

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

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

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

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

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

One can verify that given Σ_Q, $\mathsf{enforced}(\Sigma_Q)$ can be computed in polynomial time (PTIME) along the same lines as how closures for traditional FDs are computed (see, *e.g.,* [AHV95]).

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

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

Conflicting GFDs characterizes the satisfiability of GFDs.

Lemma 22: *A set Σ of* GFDs *is satisfiable if and only if Σ is not conflicting.* □

Proof of Theorem 20. Based on the lemma, we develop an algorithm that, given a set Σ of GFDs, returns "yes" if Σ is *not* satisfiable, *i.e.,* the complement of GFD satisfiability. (a) Guess (i) a set $\Sigma' \subseteq \Sigma$, (ii) a pattern Q such that Q carries labels that appear in Σ and $|Q|$ is at most the size of the largest pattern in Σ, and (iii) a mapping from the pattern of each GFD in Σ' to Q. (b) Check whether the mappings are isomorphic to subgraphs of Q. (c) If so, derive the set Σ_Q of GFDs embedded in Q from Σ' and the guessed mappings. (d) Check whether Σ_Q is conflicting; if so, return "yes". The algorithm is correct by Lemma 22. It is in NP as steps (b), (c) and (d) are in PTIME. Thus GFD satisfiability is in coNP.

The lower bound is verified by reduction from subgraph isomorphism to the complement of the satisfiability problem. The reduction uses constant GFDs defined with DAG patterns only, and hence proves Corollary 21 as well. □

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

Corollary 23: *A set Σ of* GFDs *is always satisfiable if one of the following conditions is satisfied:*

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

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

5.3.2 The Implication Problem for GFDs

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

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

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

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

Example 31: Consider a set Σ of two GFDs $(Q_8[x,y,z], x.A = y.A \to x.B = y.B)$ and $(Q_9[x,y,z,w], x.B = y.B \to z.C = w.C)$. Consider GFD $\varphi_{11} = (Q_9[x,y,z,w], x.A = y.A \to z.C = w.C)$, where patterns Q_8 and Q_9 are given in Fig. 5.3. One can verify that $\Sigma \models \varphi_{11}$. □

The implication analysis of GFDs is NP-complete. In contrast, the problem is coNP-complete for CFDs [FGJK08].

Theorem 24: *The implication problem for* GFDs *is* NP-*complete.* □

As suggested by Example 31, to decide whether $\Sigma \models \varphi$, we have to consider the interaction between their graph patterns even when φ and all GFDs in Σ are variable GFDs, and when none of them has the form $(Q[\bar{x}], \emptyset \to Y)$. Thus the implication analysis of GFDs is more intriguing than their satisfiability analysis, in contrast to Corollary 21.

Corollary 25: *The implication problem is* NP-*complete for constant* GFDs *alone, and for variable* CFDs *alone, even when all the* GFDs *are defined with DAG patterns and when none of them has the form* $(Q[\bar{x}], \emptyset \to Y)$. □

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

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

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

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

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

Along the same lines as closures of relational FDs [AHV95], one can verify that $\mathsf{closure}(\Sigma_Q, X)$ can be computed in PTIME.

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

We characterize the implication analysis as follows.

Lemma 26: *For $\varphi = (Q[\bar{x}], X \to Y)$ and a set Σ of* GFDs*, $\Sigma \models \varphi$ if and only if Y is deducible from Σ and X.* □

The proof of the lemma is an extension of its relational FD counterpart (see [AHV95] for relational FDs).

Proof of Theorem 24. For the upper bound, we give an algorithm for deciding $\Sigma \models \varphi$ as follows. (a) Guess a set $\Sigma' \subseteq \Sigma$, and a mapping from the pattern of each GFD in Σ' to the pattern Q of φ. (b) Check whether the mappings are isomorphic to subgraphs of Q. (c) If so, derive the set Σ_Q of GFDs embedded in Q from Σ' and the guessed mappings. (d) Check whether $Y \in \mathsf{closure}(\Sigma_Q, X)$; if so, return "yes". The algorithm is in NP since steps (b), (c) and (d) are in PTIME. Its correctness follows from Lemma 26.

When the assumption about the satisfiability of Σ and X in φ is lifted, the algorithm can be extended with two initial steps: (i) check whether Σ is not satisfiable in NP;

if so, return "invalid", and otherwise continue; (ii) check whether X is satisfiable, in PTIME; if so, continue; otherwise return "yes". The extended algorithm is still in NP. That is, the assumption does not increase the complexity bound.

The lower bound is verified by reduction from a variant of subgraph isomorphism, which is shown NP-complete. The reduction uses constant GFDs only or variable CFDs only, all defined with DAGs. Thus it also proves Corollary 21. □

Tractable cases. An efficient special case is as follows.

Corollary 27: *The implication problem is in* PTIME *for* GFDs *defined with tree-structured patterns.* □

5.4 Inconsistency Detection

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

5.4.1 GFD Validation and Error Detection

Given a GFD $\varphi = (Q[\bar{x}], X \to Y)$ and a graph G, we say that a match $h(\bar{x})$ of Q in G is a *violation* of φ if $G_h \not\models \varphi$, where G_h is the subgraph induced by $h(\bar{x})$. For a set Σ of GFDs, we denote by $\mathsf{Vio}(\Sigma, G)$ the set of all violations of GFDs in G, *i.e.,* $h(\bar{x}) \in \mathsf{Vio}(\Sigma, G)$ if and only if there exists a GFD φ in Σ such that $h(\bar{x})$ is a violation of φ in G. That is, $\mathsf{Vio}(\Sigma, G)$ collects all entities of G that are inconsistent when the set Σ of GFDs is used as data quality rules.

The *error detection problem* is stated as follows:

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

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

Proposition 28: *Validation of* GFDs *is* coNP*-complete.* □

Proof: We show that it is NP-hard to check, given G and Σ, whether $G \not\models \Sigma$, by reduction from subgraph isomorphism.

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

In contrast, validation is in PTIME for FDs and CFDs, and errors can be detected in relations by two SQL queries that can be automatically generated from FDs and CFDs [FGJK08]. That is, error detection is more challenging in graphs.

A sequential algorithm. We give an algorithm that, given a set Σ of GFDs and a graph G, computes $\mathsf{Vio}(\Sigma, G)$ with a single processor. It is denoted as detVio and works as follows. (1) It starts with $\mathsf{Vio}(\Sigma, G) = \emptyset$. (2) For each $(Q[\bar{x}], X \to Y)$ in Σ, it enumerates

all matches $h(\bar{x})$ of Q in G, and checks whether $h(\bar{x}) \not\models X \to Y$; if so, it adds $h(\bar{x})$ to $\mathsf{Vio}(\Sigma, G)$.

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

5.4.2 Parallel Scalability

Is error detection feasible in large-scale graphs? Our answer is affirmative, by using parallel algorithms to compute $\mathsf{Vio}(\Sigma, G)$. To characterize the effectiveness of parallelization, we adopt the notion of *parallel scalability* given in Section 3.4.1. Denote by

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

An error detection algorithm is *parallel scalable* if

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

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

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

We start with notions. Consider a GFD $\varphi = (Q[\bar{x}], X \to Y)$, where $(Q_1, \ldots, Q_k)$ are (maximum) connected components of Q. Consider $\bar{z} = (z_1, \ldots, z_k)$, where for $i \in [1, k]$, z_i is a variable in $\bar{x}$ such that $\mu(z_i)$ is a node in Q_i, where μ is the mapping from variables to nodes in Q (see Section 5.1). We fix a $\bar{z}$, referred to as the *pivot* of φ, by picking z_i with the minimum radius in Q_i, where the *radius* is the longest shortest distance between $\mu(z_i)$ and any node in Q_i. We use $\mathsf{PV}(\varphi)$ to denote $((z_1, c_Q^1), \ldots, (z_k, c_Q^k))$, referred to as the *pivot vector* of φ, where c_Q^i is the radius of Q_i at $\mu(z_i)$.

Observe the following. (a) By the locality of subgraph isomorphism, for any graph G, match $h(\bar{x})$ of Q in G, and any node $v = h(x)$ for $x \in \bar{x}$, v is within c^i_Q hops of some $h(z_i)$. (b) Vector $\mathsf{PV}(\varphi)$ can be computed in $O(|Q|^2)$ time, where Q is much smaller than G in real life. (c) Pattern Q typically has 1 or 2 connected components, and 99% of the components have radius at most 2 [GFMPdlF11]. Hence in $\mathsf{PV}(\varphi)$, the arity $\|\bar{z}\|$ and each radius c^i_Q are typically 1 or 2.

Example 32: For GFDs of Example 28, $\mathsf{PV}(\varphi_1)$, $\mathsf{PV}(\varphi_2)$, $\mathsf{PV}(\varphi_4)$ and $\mathsf{PV}(\varphi_6)$ are $((x,1),(y,1))$, $((x,1))$, $((x,0),(y,0))$ and $((x,3))$, respectively (see Fig. 5.2); in particular, we take account x as a pivot of Q_6; similarly for φ_3 for φ_5. □

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

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

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

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

Challenges.

Computing $\mathsf{Vio}(\Sigma, G)$ is a bi-criteria optimization problem. (a) *Workload balancing*, to evenly partition $W(\Sigma, G)$ over n processors; it is to avoid "skewed" partitions, *i.e.,* when a processor gets far more work units than others, and hence, to maximize parallelism. (b) *Minimizing data shipment*, to reduce communication cost, which is

often a bottleneck [ABC+11]. When a graph G is fragmented and distributed across processors, to process a work unit $w = \langle \bar{v}_z, G_{\bar{z}} \rangle$, we need to ship data from one processor to another to assemble $G_{\bar{z}}$. The cost, denoted by $\mathsf{CC}(w)$, is measured by $c_s * |M|$, where c_s is a constant and M is the data shipped.

Parallel scalable error detection. We tackle these challenges in the following two settings, which are practical parallel paradigms as demonstrated by [HRN+15]. We show that parallel scalability is within reach in these settings.

Replicated G. Graph G is replicated at each processor [HRN+15]. We study *error detection with replicated G* (Section 5.5.1), to balance workload $W(\Sigma, G)$ over n processors such that the overall parallel time for computing $\mathsf{Vio}(\Sigma, G)$ is minimized.

Theorem 29: *There exists a parallel scalable algorithm that given a set Σ of* GFDs *and a graph G replicated at n processors, computes* $\mathsf{Vio}(\Sigma, G)$ *in* $O(\frac{t(|\Sigma|,|G|)}{n} + |W(\Sigma, G)|(n + \log|W(\Sigma, G)|))$ *parallel time.* □

Partitioned G. When G is partitioned across processors, data shipment in inevitable. We study *error detection with partitioned G* (Section 5.5.2), with *bi-criteria* objective to (a) minimize data shipment and (2) balance the workload.

Theorem 30: *There exists a parallel scalable algorithm that given a set Σ of* GFDs*, a partitioned graph G and n processors, computes* $\mathsf{Vio}(\Sigma, G)$ *in* $O(\frac{t(|\Sigma|,|G|)}{n} + n|W(\Sigma, G)|^2 \log|W(\Sigma, G)| + |\Sigma||W(\Sigma, G)|)$ *parallel time.* □

We defer the proofs to the next section.

5.5 Parallel Algorithms

We next develop parallel scalable algorithms for error detection in the settings given above, as proofs of Theorems 29 and 30 in Sections 5.5.1 and 5.5.2, respectively. Such algorithms make it feasible to detect errors in large-scale graphs. We should remark that there exist other criteria for measuring the effectiveness of parallel algorithms (see Section 5.7).

5.5.1 Parallel Algorithm for Replicated Graphs

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

Algorithm. The algorithm is denoted as repVal and shown in Fig. 5.4. Working with a coordinator S_c and n processors $S_1,\ldots,S_n$, it takes the following steps. (1) It first estimates workload $W(\Sigma,G)$, and creates a balanced partition $W_i(\Sigma,G)$ of $W(\Sigma,G)$ for $i \in [1,n]$, by invoking a parallel procedure bPar (line 1). It then sends $W_i(\Sigma,G)$ to processor S_i (line 2). (2) Each processor S_i detects its set of local violations, denoted by $\mathsf{Vio}_i(\Sigma,G)$, by a procedure localVio in parallel (line 3), which only visits the data blocks specified in $W_i(\Sigma,G)$. (3) When all processors S_i return $\mathsf{Vio}_i(\Sigma,G)$, S_c computes $\mathsf{Vio}(\Sigma,G)$ by taking a union of all $\mathsf{Vio}_i(\Sigma,G)$ (lines 4-5). It then returns $\mathsf{Vio}(\Sigma,G)$ (line 6).

We next present procedures bPar and localVio.

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

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

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

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

Algorithm repVal

Input: A set Σ of GFDs, coordinator S_c, n processors $S_1, \ldots, S_n$,
a graph G replicated at each processor

Output: Violation set $\mathsf{Vio}(\Sigma, G)$.

1. $\mathsf{bPar}(\Sigma, G)$; /*balance workload in parallel*/

/*executed at coordinator S_c*/

2. send $W_i(\Sigma, G)$ to processor S_i;
3. invoke $\mathsf{localVio}(\Sigma, W_i(\Sigma, G))$ at each processor S_i for $i \in [1, n]$;
4. **if** every processor S_i returns answer $\mathsf{Vio}_i(\Sigma, G)$ **then**
5. $\quad$ $\mathsf{Vio}(\Sigma, G) := \bigcup_{i \in [1,n]} \mathsf{Vio}_i(\Sigma, G)$);
6. **return** $\mathsf{Vio}(\Sigma, G)$;

Procedure $\mathsf{localVio}(\Sigma, W_i(\Sigma, G))$

/*executed at each processor S_i in parallel*/

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

Figure 5.4: Algorithm repVal

(b) For each $\mathsf{PV}(\varphi) = ((z_1, c_Q^1), \ldots, (z_k, c_Q^k))$ and each z_i, it evenly partitions candidates $C(\mu(z_i))$ into m sets, for a predefined number m. More specifically, it derives an m-balanced partition $R_{\mu(z_i)} = \{r_1, \ldots, r_m\}$ of value ranges of a selected attribute of $C(\mu(z_i))$, such that the number of candidates in $C(\mu(z_i))$ whose attribute values fall in each range r_j is even. This is done by using *e.g.,* precomputed equi-depth histogram (*e.g.,* [MZ11]). It then constructs a set M of messages of the form $\langle \mathsf{PV}(\varphi), \bar{r}_z \rangle$, where φ is a GFD, $\bar{r}_z = \langle r_{z_1}, \ldots r_{z_k} \rangle$, and each $r_{z_i} \in R_{\mu(z_i)}$ is a *range* of $C(\mu(z_i))$ for z_i. Removing duplicates, M contains at most m^k messages for φ, where $k \leq 2$ in practice (see

Section 5.4).

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

Example 33: Consider GFD φ_1 of Example 28, where $\mathsf{PV}(\varphi_1) = ((x,1),(y,1))$ (*i.e.,* $k = 2$). Consider graph G including 9 flights flight_1–flight_9. For $n = 3 = m$, procedure bPar balances the estimation $W(\varphi_1, G)$ as follows.

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

(2) It yields a set M of 6 messages $\langle \mathsf{PV}(\varphi_1), (r_{\mathsf{flight}}, r'_{\mathsf{flight}})\rangle$ after removing duplicates (since the two connected components in Q_1 (Fig. 5.2) of φ_1 are isomorphic, $(\mathsf{PV}(\varphi_1), r_i, r_j)$ and $(\mathsf{PV}(\varphi_1), r_j, r_i)$ are duplicates for ranges r_i and r_j).

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

□

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

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

Example 34: For $\langle \mathsf{PV}(\varphi_1), ([\mathsf{flight}_1, \mathsf{flight}_3], [\mathsf{flight}_1, \mathsf{flight}_3])\rangle$, processor S_1 finds 3 candidates $\{\mathsf{flight}_1, \mathsf{flight}_2, \mathsf{flight}_3\}$ in the range $[\mathsf{flight}_1, \mathsf{flight}_3]$, and their 1-hop neighbors. These yield $v_{\bar{z}}[x]$ as $(\mathsf{flight}_i, \mathsf{flight}_j)$ ($i \in [1,3]$, $j \in [1,3]$, and $i < j$ to remove duplicates) and correspondingly, 3 work units encoded with $|G_{\bar{z}}|$, where $|G_{\bar{z}}|$ is the total size of the 1-hop neighbors of flight_i and flight_j in $v_{\bar{z}}[x]$. For example, a unit w_1 is $\langle(\mathsf{flight}_1, \mathsf{flight}_2), 22\rangle$, where $G_{\bar{z}}$ for w_1 is graph G_1 in Fig. 25, which has 22 nodes and edges in total. □

(3) Procedure bPar, at the coordinator S_c, collects a set of messages $\langle v_{\bar{z}}, |G_{\bar{z}}|\rangle$ from all

the processors, denoted by $W(\Sigma, G)$. It encodes the set of work units to be partitioned.

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

Refer to the largest cost incurred at a processor as the *makespan* of the parallel processing. The load balancing problem is "equivalent to" makespan minimization [AMZ03], by setting the capacity of each processor as $\frac{t(|\Sigma|, |G|)}{n}$, via PTIME reductions. The problem is intractable, but approximable.

Proposition 31: *(1) The load balancing problem is* NP*-complete. (2) There is a* 2*-approximation algorithm to find a balanced workload partition in* $O(n|W(\Sigma, G)| + |W(\Sigma, G)| \log |W(\Sigma, G)|)$ *parallel time for given* Σ*, n and* $W(\Sigma, G)$. □

Given $W(\Sigma, G)$, procedure bPar computes a balanced n-partition with a greedy strategy, following an approximation algorithm of [AMZ03] for makespan minimization. (1) It first associates a weight $|G(\bar{z})|$ with each work unit $w = \langle v_{\bar{z}}, |G_{\bar{z}}| \rangle$. It then sorts all the work units, in descending order of the weights. With each processor it associates a load, initially 0. (2) It greedily picks a work unit w with the smallest weight and a processor S_i with the minimum load, assigns w to S_i and updates the load of S_i by adding the weight of w. (3) The process proceeds until all work units are distributed. This yields a 2-approximation algorithm, by approximation-factor preserving reduction to its counterpart of [AMZ03].

Example 35: Suppose that coordinator S_c receives 9 work units $\{w_1, \ldots, w_9\}$ in total, with estimated size $\{22, 22, 26, 26, 30, 30, 24, 28, 28\}$, respectively. The greedy assignment strategy of bPar generates a 3-partition of the work units as $\{\{w_1, w_3, w_9\}, \{w_2, w_4, w_5\}, \{w_6, w_7, w_8\}\}$, with balanced block sizes as 76, 78, 82, respectively. Then S_c assigns the 3 partitions to processors S_1, S_2, S_3, respectively.

□

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

tor S_c at the end of the process.

Example 36: Consider GFD $\varphi_1 = (Q_1[\bar{x}], X_1 \to Y_1)$ (Example 28) and work unit w_1 (Example 34) assigned to processor S_1. Procedure localVio inspects G_1 (Fig. 5.1) for w_1, and finds a match $h_1(\bar{x})$ of Q_1 in G_1, where h_1 is given in Example 6. As shown there, $h_1(\bar{x}) \not\models X_1 \to Y_1$. Thus localVio adds $h_1(\bar{x})$ to $\mathsf{Vio}_1(\Sigma, G)$. Similarly, S_1 processes w_3 and w_9 assigned to it, and finally returns $\mathsf{Vio}_1(\Sigma, G)$ to S_c.

□

Proof of Theorem 29. By the locality of subgraph isomorphism, procedure bPar identifies all work units, and localVio computes all violations. From these the correctness of repVal follows. For the complexity, one can verify the following: (a) procedure bPar estimates $W(\Sigma, G)$ in $O(\frac{|W(\Sigma,G)|}{n})$ parallel time, by using a balanced partition; the partitioning takes $O(n|W(\Sigma, G)| + |W(\Sigma, G)| \log |W(\Sigma, G)|)$ time [AMZ03]; and (b) procedure localVio takes $O(\frac{t(|\Sigma|,|G|)}{n})$ parallel time, via a balanced workload partition. Thus repVal has the complexity stated in Theorem 29 and is parallel scalable. □

5.5.2 Algorithm for Fragmented Graphs

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

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

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

Algorithm disVal works with a coordinator S_c and n processors $S_1, \ldots, S_n$. (1) It first estimates and partitions workload $W(\Sigma, G)$ via a procedure disPar, such that the workload $W_i(\Sigma, G)$ at each S_i is balanced, with minimum communication cost. (2)

Each processor S_i uses a procedure dlovalVio to detect local violation $\mathsf{Vio}_i(\Sigma, G)$, in parallel, with data exchange. (3) Finally, $\mathsf{Vio}(\Sigma, G) = \bigcup_{i \in [1,n]} \mathsf{Vio}(\Sigma_i, G)$.

We next present procedures disPar and dlovalVio.

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

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

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

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

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

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

Proposition 32: *(1) The bi-criteria assignment problem is* NP*-complete. (2) There exists a* 2*-approximation algorithm to find a balanced workload assignment with minimized communication cost in* $O(n|W(\Sigma,G)|^2 \log(|W(\Sigma,G)|))$ *time.* □

Extending a strategy for makespan minimization [ST93], procedure disPar computes an n-partition of $W(\Sigma)$ (after unit grouping) into $W_i(\Sigma,G)$, sent to processor S_i for $i \in [1,n]$.

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

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

Partial detection. We can also ship partial matches instead of data blocks. The idea is to estimate the size of partial matches via *graph simulation* [FWWD14] from pattern $Q[\bar{x}]$ in a GFD φ to F_i. If the number of partial matches is not large, S_i exchanges such matches with other processors in a pipelined fashion, and updates $\mathsf{Vio}_i(\Sigma,F_i)$ as soon as a complete match can be formed from partial ones.

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

Our algorithms also support optimization strategies for skewed graphs and workload reduction.

We verify Theorem 30 by showing that disVal is correct and has the desired complexity, similar to Theorem 29.

5.6 Experimental Study

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

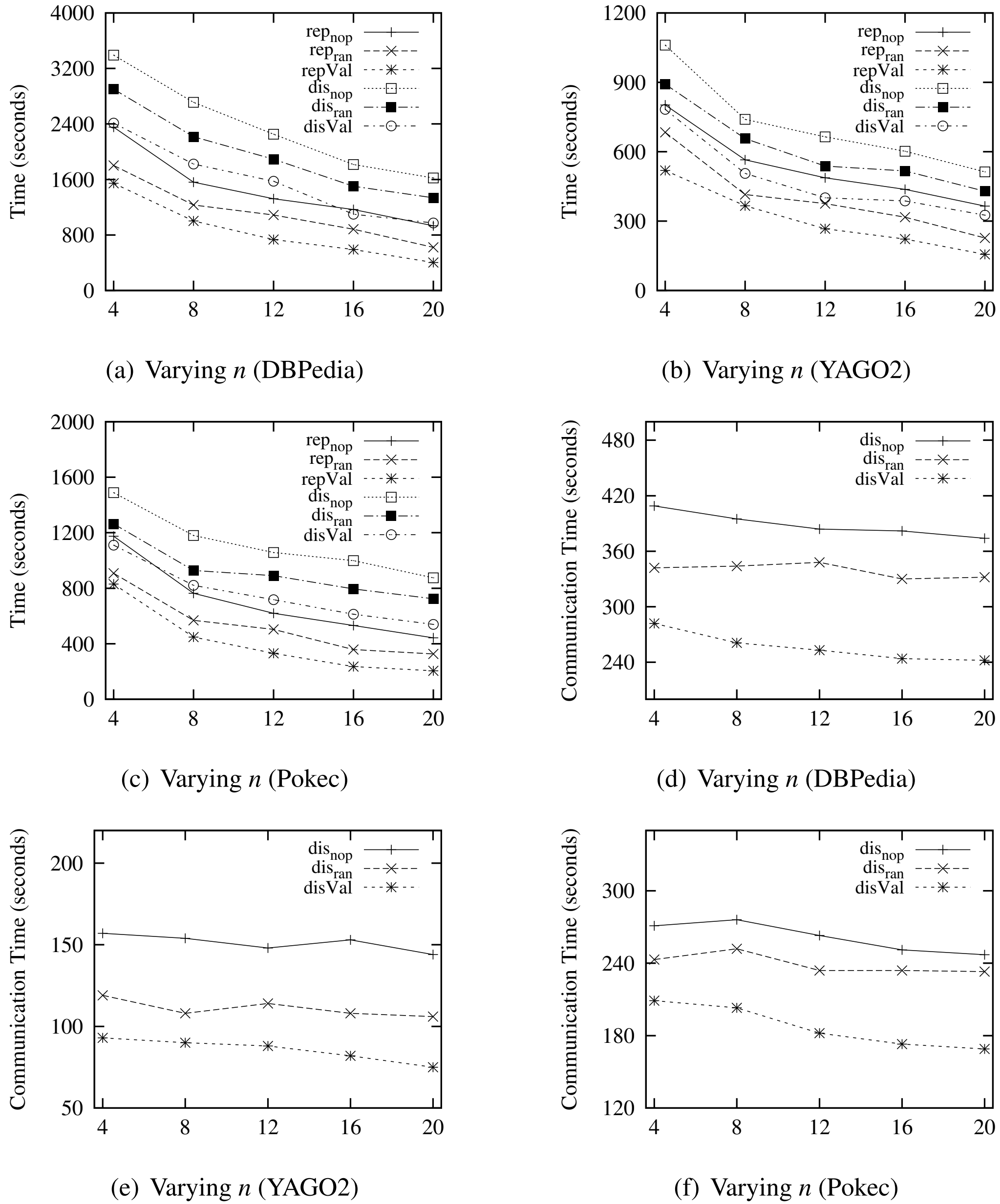
Experimental setting. We used three real-life graphs: (a) DBpedia, a knowledge graph [dbp] with 28 million entities of 200 types and 33.4 million edges of 160 types, (b) YAGO, an extended knowledge base of YAGO [SKW07] with 3.5 million nodes of 13 types and 7.35 million edges of 36 types, (c) Pokec [Pok], a social network with 1.63 million nodes of 269 different types, and 30.6 million edges of 11 types. We removed meaningless nodes and labels for a compact representation. We then inserted new edges by repeatedly dereferencing HTTP URIs over a set of sampled entities to further enlarge DBpedia (resp. YAGO), to 12.3 million (resp. 3.2 million) entities and 32.7 million (resp. 7.1 million) edges.

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

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

Algorithms. We implemented the following, all on GRAPE: (1) sequential algorithm detVio (Section 5.4), (2) parallel algorithm repVal (Fig. 5.4), versus its two variants (a) $\mathsf{rep}_{\mathsf{ran}}$, which randomly assigns work units to processors, and (b) $\mathsf{rep}_{\mathsf{nop}}$, which does not support optimization strategies (multi-query processing [LKDL12] and workload reduction), and (3) parallel algorithm disVal (Section 5.5.2), versus its two variants $\mathsf{dis}_{\mathsf{ran}}$ and $\mathsf{dis}_{\mathsf{nop}}$ similar to their counterparts in (2).

We deployed the algorithms on GRAPE, and used up to 20 instances. Each experi-



(a) Varying n (DBPedia)

(b) Varying n (YAGO2)

(c) Varying n (Pokec)

(d) Varying n (DBPedia)

(e) Varying n (YAGO2)

(f) Varying n (Pokec)

Figure 5.5: Parallel scalability and communication

ment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Parallel scalability. We first evaluated parallel algorithms repVal and disVal, versus their variants. Fixing $|Q|$=5 and $\|\Sigma\|$=50, we varied the number n of processors from 4 to 20. We replicated and fragmented G for repVal and disVal, respectively. Figures 5.5(a), 5.5(b) and 5.5(c) report their performance on real-life DBpedia, YAGO and Pokec, respectively. We find the following. (1) Both repVal and disVal substantially

reduce parallel time when n increases: they are on average 3.7 and 2.4 times faster for n from 4 to 20, respectively. These validate Theorems 29 and 30. (2) Both repVal and disVal outperform their variants: repVal (resp. disVal) is on average 1.9 and 1.4 times (resp. 1.5 and 1.3 times) faster than $\mathsf{rep}_{\mathsf{nop}}$ and $\mathsf{rep}_{\mathsf{ran}}$ (resp. $\mathsf{dis}_{\mathsf{nop}}$ and $\mathsf{dis}_{\mathsf{ran}}$), respectively. These verify the effectiveness of our optimization and load balancing techniques. (3) Algorithm repVal is faster than disVal, since it requires no data exchange by trading with replicated G. (4) Both repVal and disVal work well on large real-life graphs. For example, repVal (resp. disVal) takes 156 (resp. 326) seconds on YAGO with 20 processors. In contrast, sequential algorithm detVio does not terminate on any of the three graphs within 6000 seconds. On average parallel graph replication (not shown) takes 21.3, 89 and 75 seconds for YAGO, DBpedia and Pokec, respectively. The replication is performed once and is reused for all queries.

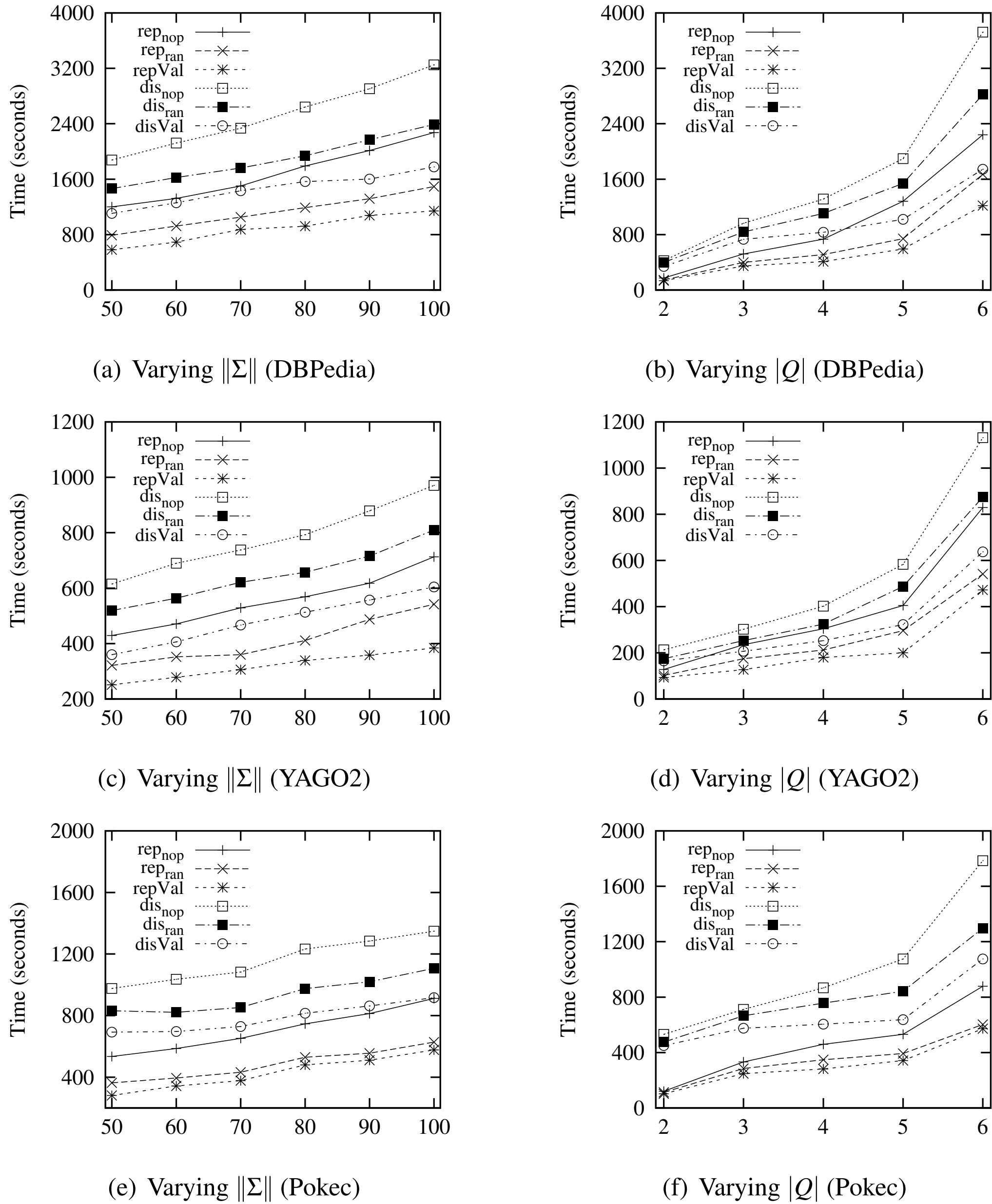
Exp-2: Workload complexity. We next evaluated the impact of the complexity of GFDs on workload estimation and partition, by varying $\|\Sigma\|$, the number of GFDs, and $|Q|$, the average pattern size. We fixed $n = 16$.

Varying $\|\Sigma\|$. Fixing $|Q| = 5$, we varied $\|\Sigma\|$ from 50 to 100. As shown in Figures 5.6(a), 5.6(c) and 5.6(e) on DBpedia, YAGO and Pokec, respectively, (a) all the algorithms take longer time over larger Σ, as expected, and

(b) repVal (resp. disVal) behaves better than $\mathsf{rep}_{\mathsf{ran}}$ and $\mathsf{rep}_{\mathsf{nop}}$ (resp. $\mathsf{dis}_{\mathsf{ran}}$ and $\mathsf{dis}_{\mathsf{nop}}$), by balancing workload and minimizing communication. However, detVio does not terminate within 120 minutes on any of the three graphs when $\|\Sigma\| \geq 80$.

Varying $|Q|$. Fixing $\|\Sigma\| = 50$, we varied $|Q|$ from 2 to 6. As shown in Figures 5.6(b), 5.6(d) and 5.6(f), all the algorithms take longer over larger $|Q|$, due to larger work units. However, repVal (resp. disVal) outperforms $\mathsf{rep}_{\mathsf{nop}}$ and $\mathsf{rep}_{\mathsf{ran}}$ (resp. $\mathsf{dis}_{\mathsf{nop}}$ and $\mathsf{dis}_{\mathsf{ran}}$) in all the cases, for the same reasons given above. Again, detVio does not terminate in 120 minutes when $|Q| \geq 6$ on all the three graphs.

Exp-3: Communication cost. In the same setting as Exp-1, we evaluated the total communication cost (measured as parallel data shipment time) of disVal, $\mathsf{dis}_{\mathsf{ran}}$ and $\mathsf{dis}_{\mathsf{nop}}$ over the three datasets, reported in Figures 5.5(d), 5.5(e) and 5.5(f), respectively. We omit repVal since it does not require data exchange. We find the following: (a) the total amount of data shipped (not shown) is far smaller than the size of the underlying graphs; this confirms our estimate of communication costs (Sections 5.4 and 3.3.2); (b) the communication cost takes from 12% to 24% of the overall error detection cost when n changes from 4 to 20; this is one of the reasons why adding processors



(a) Varying $\|\Sigma\|$ (DBPedia)

(b) Varying $|Q|$ (DBPedia)

(c) Varying $\|\Sigma\|$ (YAGO2)

(d) Varying $|Q|$ (YAGO2)

(e) Varying $\|\Sigma\|$ (Pokec)

(f) Varying $|Q|$ (Pokec)

Figure 5.6: Workload complexity

does not always reduce parallel running time [FWWD14], since using more processors introduce more data exchange among different processors; and (c) although more data is shipped with larger n, the communication time is not very sensitive to n due to parallel shipment.

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

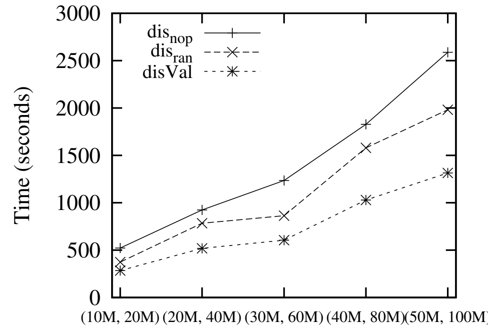


Figure 5.7: Scalability: Varying $|G|$ (synthetic)

Fixing n = 16, we varied $|G|$ from (10M, 20M) to (50M, 100M). As shown in Fig. 5.7, (1) all the algorithms take longer time over larger $|G|$, as expected; (2) error detection is feasible in large graphs: disVal takes 21 minutes when $|G|$ = (50M, 100M); (3) disVal is on average 1.9 and 1.5 times faster than dis_{ran} and dis_{nop}, respectively; this is consistent with the results on real-life graphs; and (4) sequential algorithm detVio does not run to completion when $|G| \geq$ (30M, 60M) within 120 minutes with one processor.

Exp-5: Effectiveness. To demonstrate the effectiveness of GFDs in error detection, we show in Fig. 5.8 three real-life GFDs and error caught by them.

GFD *1* is $(Q_{10}[\bar{x}], \emptyset \to x.\text{val} = c \ \wedge \ y.\text{val} = d)$ for distinct c and d, (*i.e.,* $x.\text{val} = c \ \wedge$ $y.\text{val} = d$ is false, stating that a person x cannot have y as both a child and a parent. It catches inconsistency in *YAGO2* shown in Fig. 5.8.

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

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

We also evaluated the effectiveness of GFDs for error detection with YAGO, by comparing with (a) the extension of CFDs to RDF [HZZ14], referred to as GCFDs, and (b) BigDansing [KIJ$^+$15]. Since the complete set of "true" errors in YAGO is unknown, we sampled a set of entities. For each sampled entity x, we randomly injected noise (with probability 2%, 2690 errors in total) into YAGO as suggested by [ZKS$^+$13]: (a) *attribute inconsistency*, by changing the value of an attribute $x.A$; (b) *type inconsis-*

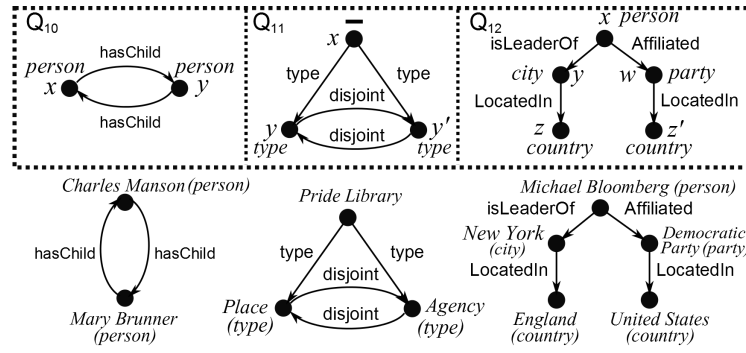


Figure 5.8: Real-life GFDs

tency, by revising the type of x; and (c) *representational inconsistency*, by revising the value of either $x.A$ or $x'.A$ if $x.A{=}x'.A$ and x and x' are of the same type. Denote the set of entities with noise as Vio, we define the *precision* (resp. *recall*) of an error detection method $\mathcal{A}$ as $\frac{|\mathsf{Vio} \cap \mathsf{Vio}(\mathcal{A})|}{|\mathsf{Vio}(\mathcal{A})|}$) (resp. $\frac{|\mathsf{Vio} \cap \mathsf{Vio}(\mathcal{A})|}{|\mathsf{Vio}|}$, where $\mathsf{Vio}(\mathcal{A})$ denotes the inconsistent entity set detected by $\mathcal{A}$.

We constructed (1) a set Σ of 10 GFDs on YAGO with frequent patterns that match a fraction of sampled entities and with constants from the original values before noise injection; and (2) a set of 7 GCFDs over sampled entities following [HZZ14], including *all* GFDs in Σ with conjunctive paths (GCFDs do not allow general graph patterns). (3) We hard-coded the GFDs as user-defined functions for each GFD in Σ, as BigDansing does not support subgraph isomorphism.

We report the running time and accuracy of these methods in Fig. 5.2, with $n = 16$ on YAGO extended with noise. We find that (a) GFDs has higher accuracy (91%) than GCFDs, since it catches inconsistencies with general patterns not expressible by GCFDs; (b) it takes comparable time for GFDs and GCFDs; and (c) BigDansing is 4.6 times slower, because it had to cast subgraph isomorphic testing as relational joins. It reports the same accuracy as our algorithm since it hard-coded the same set Σ of GFDs.

Real-world GFDs. Observe the following about the GFDs depicted in Fig. 5.8.

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

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

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

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

Table 5.2: Running time and accuracy

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

5.7 Related Work

We categorize related work to this chapter as follows.

FDs *on graphs*. Extensions of FDs and CFDs have been studied for RDF [CCP12, ACCP10, YH11, CFP+14, HZZ14, HGPW15]. The definitions of FDs in [CCP12, ACCP10, HGPW15] are based on RDF triple embedding and the coincidence of variable valuations. FDs are extended [YH11] to specify value dependencies on clustered values via, *e.g.,* path patterns; similarly for extensions of CFDs [HZZ14]. A schema matching framework is proposed in [CFP+14], for transformations between RDF and relations. It defines FDs as trees in which each node denotes an attribute in a corresponding relation.

Our work differs from the prior work in the following. (1) We define GFDs with graph patterns to express topological constraints of (property) graphs, beyond RDF. (2) GFDs capture inconsistencies in graph-structured entities identified by patterns. In contrast, the FDs of [CCP12, ACCP10, HGPW15] are value-based regardless of what entities carry the values, and the reasoning techniques of [HGPW15] are based on relational encoding of RDF data. Moreover, these FDs cannot express equality with constants (semantic value binding) as in CFDs, *e.g.,* x.city = "Edi", while GFDs subsume CFDs.

The FDs of [CFP+14] are defined as trees and assume a relational schema. They do not support general topological constraints; similarly for [YH11, HZZ14]. (3) We provide complexity bounds for GFD analyses and parallel scalable algorithms for error detection in graphs, which were not studied by the prior work.

Closer to this work is [FFTD15] on keys for graphs [FFTD15], which differ from GFDs in the following. (1) Keys are defined simply as a graph pattern $Q[x]$, with a designated variable x denoting an entity. In contrast, GFDs have the form $(Q[\bar{x}], X \to Y)$, where $\bar{x}$ is a list of variables, and X and Y are conjunctions of equality atoms with constants and variables in $\bar{x}$. GFDs cannot be expressed as keys, just like that relational FDs are not expressible as keys. Moreover, keys of [FFTD15] are recursively defined to identify entities, while GFDs are an extension of conventional FDs and are not recursively defined. (2) Keys are defined on RDF triples (s, p, o), while GFDs are defined on property graphs, *e.g.,* social networks. (3) Keys are interpreted in terms of three isomorphic mappings: two from subgraphs to Q, and one between the two subgraphs. In contrast, GFDs needs a single isomorphic mapping from a subgraph to Q. In light of the different semantics, algorithms for GFDs and keys are radically

different. (4) We study the satisfiability and implication for GFDs; these classical problems were not studied for keys [FFTD15].

Inconsistency detection has been studied for relations (see [FG12] for a survey), and recently for knowledge bases (linked data) [HZZ14, PS04, Men04, SDNR07, SSW09, PD07]. The methods for knowledge bases employ either rules [HZZ14, Men04, PS04, SDNR07, SSW09], or probabilistic inferences [PD07]. (1) Datalog rules are used [SDNR07] to extract entities and detect inconsistent "facts". SOFIE [SSW09] maintains the consistency of extracted facts by using rules expressed as first-order logic (FO) formulas along with textual patterns, existing ontology and semantic constraints. Pellet [PS04] checks inconsistencies by using inference rules in description logic (*e.g.,* OWL-DL). Dependency rules are used to detect inconsistencies in attribute values in semantic Web [Men04] and RDF [HZZ14]. BigDansing [KIJ$^+$15] supports user-defined rules for repairing relational data. To clean graph-structured entities, it needs to represent graphs as tables and encode isomorphic functions beyond relational query languages. (2) The inference method of [PD07] uses Markov logic to combine FO and probabilistic graphical models, and detects errors by learning and computing joint probability over structures.

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

Parallel algorithms related to GFD validation algorithms are (1) algorithms for detecting errors in distributed data [FGMM10, FLTY14], and (2) algorithms for subgraph enumeration, subgraph isomorphism and SPARQL [GHS14, HRN$^+$15, AFU13, SCC$^+$14, LQLC15, SWW$^+$12, HAR11, RvRH$^+$14, LKDL12].

(1) Algorithms of [FGMM10, FLTY14] (incrementally) detect errors in (horizontally or vertically) partitioned relations based on CFDs. The methods work on relations,

but do not help GFDs that require subgraph isomorphism computation. Indeed, our algorithms are radically different from those of [FGMM10, FLTY14].

(2) Closer to this work are parallel algorithms for subgraph enumeration [Pla13, AFU13, SCC$^+$14, LQLC15]. (a) MapReduce algorithms are proposed via conjunctive multi-way join operations [AFU13] and decomposed edge joins [Pla13]. The strategy is effective for triangle counting [SV11]. (b) To reduce excessive partial answers for general patterns, a MapReduce solution in [LQLC15] decomposes a pattern into twin twigs (single edge or two incident edges), and adopts a left-deep-join strategy to join multiple edges as stars. To cope with skewed nodes, the neighborhoods of high-degree nodes are partitioned, replicated and distributed. Decomposition strategies are used to reduce MapReduce rounds and I/O cost. (c) A BSP framework is developed in [SCC$^+$14] via vertex-centric programming. It adopts an online greedy strategy to assign partial subgraphs to workers that incur minimum overall workload, and optimization strategies to reduce subgraph instances.

(3) A number of parallel algorithms are developed for subgraph isomorphism [SWW$^+$12, RvRH$^+$14] and SPARQL queries [GHS14, LKDL12, HRN$^+$15, HAR11]. Twig decomposition is used to prune the intermediate results and reduce the latency in Trinity memory cloud [SWW$^+$12]. The in-memory algorithm of [RvRH$^+$14] parallelizes a backtracking procedure by (a) evenly distributing partial answers among threads for local expansion, and (b) copying the partial answers to a global storage for balanced distribution in the next round. Hash-based partitioning, query decomposition and load balancing strategies are introduced for parallel SPARQL on RDF [GHS14, HAR11]. Query decomposition and plan generation techniques are studied in [HRN$^+$15], which avoid communication cost by replicating graphs. Optimization techniques for multi-pattern matching are provided in [LKDL12], by extracting common sub-patterns. Many of these techniques leverage RDF schema and SPARQL query semantics, which are not available for GFDs and general property graphs.

This work differs from the prior work in the following. (a) GFD validation in distributed graphs is a bi-criteria optimization problem, to balance workload and minimize communication cost, with combined complexity from subgraph enumeration of *disconnected* patterns and dependency checking in fragmented graphs. It is more challenging than graph queries studied in the prior work. (b) We introduce a workload assignment strategy for the intractable optimization problem, with approximation bounds, instead of treating workload balancing and communication cost minimization

separately [LQLC15, SCC$^+$14]. (c) We warrant parallel scalability, which is not guaranteed by the prior algorithms.

On the other hand, this work can benefit from prior techniques for fast parallel subgraph matching and listing, *e.g.,* query decomposition strategies [LQLC15, SWW$^+$12, HRN$^+$15] and multi-thread in-memory algorithm [RvRH$^+$14], for local error detection at each worker. We have adopted the optimization techniques of [LKDL12], and will incorporate others into GFD tools.

(4) There has also been work on characterizing the effectiveness of parallel algorithms, in terms of communication costs of MapReduce algorithms [AU10], constraints on MapReduce computation/communication cost (MRC [KSV10], MMC [TLX13] and SGC [QYC$^+$14]), and the polynomial fringe property of recursive programs [ABC$^+$11]. We adopt the notion of parallel scalability [KRS88], which measures speedup by parallelization over multiple processors, in terms of both computation and communication costs. It is for generic parallel algorithms not limited to MapReduce. A parallel scalable algorithm guarantees to scale with large graphs by adding processors. However, parallel scalability is beyond reach for certain graph computations [FWWD14]. We show that GFD validation is parallel scalable, by providing such algorithms.

Static analyses. Over relations, the satisfiability and implication problems are known to be in $O(1)$ and linear time for FDs, NP-complete and coNP-complete for CFDs, $O(1)$ time and PSPACE-complete for inclusion dependencies (INDs), respectively. The validation problem is in PTIME for FDs, CFDs and INDs (cf. [AHV95, FG12]). We show that for GFDs on graphs, validation, satisfiability and implication for GFDs are coNP-complete, coNP-complete and NP-complete, respectively. As will be seen in Section 5.3, the complexity of GFDs comes from the interactions between graph patterns (subgraph isomorphism); it is not inherited from CFDs.

5.8 Summary

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

Chapter 6

Conclusion and Future Work

In this chapter we summarise the results of this thesis and propose future work.

6.1 Conclusion

In this thesis, We have studied a new parallel graph computation engine, GRAPE, from its foundations to applications.

Framework and Foundation of GRAPE. We proposed an approach to parallelizing sequential graph algorithms. For a class of graph queries, users can plug in existing sequential algorithms with minor changes. GRAPE parallelization guarantees to terminate with correct answers under a monotonic condition if the sequential algorithms are correct. Moreover, we proved that graph algorithms for existing parallel graph systems can be migrated to GRAPE, without incurring extra cost. We have verified that GRAPE achieves comparable performance to the state-of-the-art graph systems for various query classes, and that (bounded) IncEval reduces the cost of iterative graph computations.

Association Rules Discovery on GRAPE. As an application on GRAPE, we studied association rules with graph patterns, from its syntax, semantics to support and confidence metrics. We have studied DMP and EIP, for mining GPARs and for identifying potential customers with GPARs, respectively, from complexity to parallel (scalable) algorithms. Our experimental study has verified that while DMP and EIP are hard, it is feasible to discover and make practical use of GPARs. We contend that GPARs provide a promising tool for social media marketing, among other applications.

Extending Pattern Matching on GRAPE with Quantifiers. To make GPARs sat-

isfy the need in social marketing, we have proposed quantified matching, by extending traditional graph patterns with counting quantifiers. We have studied important issues in connection with quantified matching, from complexity to algorithms to applications. The novelty of this work consists in quantified patterns (QGPs), quantified graph association rules (QGARs), and algorithms with provable guarantees (*e.g.,* optimal incremental matching and parallel scalable matching). Our experimental study has verified the effectiveness of QGPs and the feasibility of quantified matching in real-life graphs. Quantified graph pattern matching also opens new areas for the applications on GRAPE. With its expressive power, GRAPE is able to resolve more complex problems such as accurately entity identification, customer recommendations.

Functional Dependencies on Graphs. As response to the data quality issue on GRAPE, we studied functional dependencies on graphs. It is a first step towards a dependency theory for graphs. We have proposed GFDs, established complexity bounds for their classical problems, and provided parallel scalable algorithms for their application. Our experimental results have verified the effectiveness of GFD techniques.

6.2 Future Work

There are many problems related to the thesis remain open.

Asynchronised Parallel Model. GRAPE adopts BSP model and imposes many global synchronisation barriers. While simplifies the convergence analysis of GRAPE, it inevitably suffered from stale messages and worker stragglers in some cases. Some workers take much longer time than the average. Hence the system has to wait for the slowest worker. This situation gets worse when failure happens to some workers. Extending GRAPE to support asynchronous parallel processing is possibly utilise the resources in the whole system. Problems include correctness conditions, consistency and a model of cost evaluation need to be addressed.

Graph Mutation Support. It should first be remarked that state-of-the-art graph analysis platforms such as GraphLab, Giraph and Blogel all take as input static graphs, and do not mutate during the computation. This dissertation adopts this setting. However, the social network data is not static and changes constantly. Nonetheless, the use of incremental computation makes it easier than the competitors of GRAPE to support streaming updates during the computation. A simple strategy is to support a sliding

window to accumulate updates and then apply IncEval to incorporate the updates, when the updates are not substantial as commonly found in practice. We are developing a more sophisticated strategy to cope with heavy updates, by supporting concurrency control.

Discovering GPARs with quantifiers. We studied the GPARs discovering problem and expressive power of the GPARs with quantifiers. However, mine the GPARs with counting quantifiers are not easy. As remarked earlier, quantified pattern matching problem is DP-complete for patterns with possibly negative edges. This makes the mining problem hard. Besides, there still unknown whether exists a parallel scalable algorithm to discover the GPARs with quantifiers.

Parallel scalability. As remarked in previous chapters, not all parallel algorithms are guaranteed to have a linear speed up when more processors used. Worse still, there are graph query classes for which there exist no parallel algorithm with this property. A natural question is then how to characterise the effectiveness of parallel algorithms? Several models have been proposed for this purpose, *e.g.,* [FGN13], [KSV10], [QYC+14], [TLX13]. However, the study of this issue is still in its infancy. A characterization remains to be developed for general shared-nothing systems.

Discovering GFDs. To use GFDs to detect inconsistencies in real-life graphs, we assumed GFDs are in place. In fact, it is non-trivial to discover GFDs in real knowledge bases. Worse still, GFD discovery is much harder than its counterparts algorithms for relational FDs [HKPT99] and CFDs [FGLX11], since GFDs are a combination of topological constraints and attribute dependencies its validation analysis is NP-complete. It is also more challenging than graph pattern mining since it has to deal with disconnected patterns and trivial or negative GFDs. Not to mention their intractable satisfiability and implication analyses.

Repairing inconsistency for graph-structured data. As the next step of error detection in knowledge graphs, efficient methods are needed to repair the dirty data. Repairing big data is much harder than detecting errors and introduce many challenges. It is NP-hard for repairing problem even only with relational FDs [BFFR05]. The graph data lack of schemes, which makes it worse. It is more challenging when applied to the critical data such as knowledge base, which requires the fix should be 100% correct.

Bibliography

[ABC$^+$11] Foto N Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.

[ACCP10] Waseem Akhtar, Álvaro Cortés-Calabuig, and Jan Paredaens. Constraints in rdf. In *International Workshop on Semantics in Data and Knowledge Bases*, pages 23–39. Springer, 2010.

[AFU13] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, 2013.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Record*, 22(2):207–216, 1993.

[AJ08] Wilfred Amaldoss and Sanjay Jain. Research note-trading up: A strategic analysis of reference group effects. *Marketing science*, 27(5):932–942, 2008.

[AL04] Marcelo Arenas and Leonid Libkin. A normal form for xml documents. *ACM Transactions on Database Systems (TODS)*, 29(1):195–232, 2004.

[ali] Aliyun. https://www.aliyun.com.

[AMZ03] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *SPAA*, 2003.

[AR06] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[AU10] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[Ave11] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.

[AYLVY09] Sihem Amer-Yahia, Laks VS Lakshmanan, Sergei Vassilvitskii, and Cong Yu. Battling predictability and overconcentration in recommender systems. *IEEE Data Eng. Bull.*, 32(4), 2009.

[AYLY09] Sihem Amer-Yahia, Laks Lakshmanan, and Cong Yu. Socialscope: Enabling information discovery on social content sites. *arXiv preprint arXiv:0909.2058*, 2009.

[BBBG09] Michele Berlingerio, Francesco Bonchi, Björn Bringmann, and Aristides Gionis. Mining graph evolution rules. In *Machine learning and knowledge discovery in databases*, pages 115–130. 2009.

[BCD03] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[BCFK06] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 211–222. VLDB Endowment, 2006.

[BDR13] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 50, 2013.

[BFFR05] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by

value modification. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 143–154. ACM, 2005.

[BG81] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[BIJ02] Hannah Blau, Neil Immerman, and David Jensen. A visual language for querying and updating graphs. *University of Massachusetts Amherst Computer Science Technical Report*, 37:2002, 2002.

[BJG08] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[BLV14] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465. ACM, 2014.

[BM] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite.

[BMC10] M. Bendersky, D. Metzler, and W.B. Croft. Learning concept importance using a weighted dependence model. In *WSDM*, 2010.

[BN08] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 858–863. Springer, 2008.

[BU12] Ravi Bapna and Akhmed Umyarov. Do your online friends make you pay? A randomized field experiment in an online music social network. *NBER working paper*, 2012.

[BW13] Paul Burkhardt and Chris Waring. An nsa big graph experiment. In *presentation at the Carnegie Mellon University SDI/ISTC Seminar, Pittsburgh, Pa*, 2013.

[CCP12] Alvaro Cortés-Calabuig and Jan Paredaens. Semantics of constraints in rdfs. In *AMW*, pages 75–90, 2012.

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *PVLDB*, 8(12):1804–1815, 2015.

[CFP+14] Diego Calvanese, Wolfgang Fischl, Reinhard Pichler, Emanuel Sallinger, and Mantas Šimkus. 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 transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[CK00] Chandra Chekuri and Sanjeev Khanna. A PTAS for the multiple knapsack problem. In *SODA*, pages 213–222, 2000.

[CSCC15] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[CSYP12] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *NSDI*, pages 197–210, 2012.

[dat] Gartner says more than 25 percent of critical data used in large corporations is flawed.
http://www.gartner.com/newsroom/id/492028.

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

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Don14] Xin Dong et al. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.

[DPP01] Agostino Dovier, Carla Piazza, and Alberto Policriti. A fast bisimulation algorithm. In *International Conference on Computer Aided Verification*, pages 79–90. Springer, 2001.

[EASK14] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.

[ECD$^+$04] Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S Weld, and Alexander Yates. Web-scale information extraction in KnowItAll. In *WWW*, 2004.

[FFTD15] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs. *Proceedings of the VLDB Endowment*, 8(12):1590–1601, 2015.

[FG12] Wenfei Fan and Floris Geerts. Foundations of data quality management. *Synthesis Lectures on Data Management*, 4(5):1–217, 2012.

[FGJK08] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6, 2008.

[FGLX11] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

[FGMM10] Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. Detecting inconsistencies in distributed data. In *ICDE*, pages 64–75, 2010.

[FGN13] Wenfei Fan, Floris Geerts, and Frank Neven. Making queries tractable on big data with preprocessing: through the eyes of complexity theory. *Proceedings of the VLDB Endowment*, 6(9):685–696, 2013.

[FHT17] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[FLM$^+$10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275, 2010.

[FLTY14] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. Incremental detection of inconsistencies in distributed data. *TKDE*, 26(6), 2014.

[FLWW12] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 157–168. ACM, 2012.

[fre] Google Freebase. https://developers.google.com/freebase/.

[FT87] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.

[FVT12] Philippe Fournier-Viger and Vincent S Tseng. Mining top-k non-redundant association rules. In *ISMIS*. 2012.

[FWW13] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):18, 2013.

[FWWD14] Wenfei Fan, Xin Wang, Yinghui Wu, and Dong Deng. Distributed graph simulation: Impossibility and possibility. *Proceedings of the VLDB Endowment*, 7(12):1083–1094, 2014.

[FWWX15] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. Association rules with graph patterns. *Proceedings of the VLDB Endowment*, 8(12):1502–1513, 2015.

[FWX16a] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Adding counting quantifiers to graph patterns. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1215–1230. ACM, 2016.

[FWX16b] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1843–1857. ACM, 2016.

[FXW+17a] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[FXW+17b] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 495–510. ACM, 2017.

[GBDS14] Ivana Grujic, Sanja Bogdanovic-Dinic, and Leonid Stoimenov. Collecting and analyzing data from e-government facebook pages. *ICT Innovations*, 2014.

[GBL08] Amit Goyal, Francesco Bonchi, and Laks VS Lakshmanan. Discovering leaders from community actions. In *CIKM*, 2008.

[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.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[GHS14] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient RDF processing. In *WWW*, 2014.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[GLG+12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[Gon12] Neil Zhenqiang Gong et al. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *IMC*, 2012.

[goo] Google inside search. https://www.google.com/intl/es419/insidesearch/features/search/knowledge.html.

[GRA] GRAPE. *http://grapedb.io/*.

[GS09] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *Proceedings of the 18th international conference on World wide web*, pages 381–390. ACM, 2009.

[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.

[GXD+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[Har94] Tim J Harris. A survey of pram simulation techniques. *ACM Computing Surveys (CSUR)*, 26(2):187–206, 1994.

[Har04] Robert Harper. Self-adjusting computation. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 254–255. IEEE, 2004.

[HAR11] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11), 2011.

[HCD+94] Lawrence B Holder, Diane J Cook, Surnjani Djoko, et al. Substucture discovery in the subdue system. In *KDD workshop*, 1994.

[HGPW15] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. Implication and axiomatization of functional and constant constraints. *Ann. Math. Artif. Intell.*, pages 1–29, 2015.

[HHK95] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. Computing simulations on finite and infinite graphs. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 453–462. IEEE, 1995.

[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.

[HLH12] Shasha Huang, Qingguo Li, and Pascal Hitzler. Reasoning with inconsistencies in hybrid MKNF knowledge bases. *Logic Journal of IGPL*, 2012.

[HRN+15] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 2015.

[HVA14] Jiewen Huang, Kartik Venkatraman, and Daniel J Abadi. Query optimization of distributed pattern matching. In *ICDE*, 2014.

[HZZ14] Binbin He, Lei Zou, and Dongyan Zhao. Using conditional functional dependency to discover abnormal data in RDF graphs. In *SWIM*, pages 1–7, 2014.

[IWM00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*. 2000.

[JCZ13] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(01):75–105, 2013.

[Kar11] George Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. 2011.

[KBG12] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

[KBV+09] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[KCY09] Yiping Ke, James Cheng, and Jeffrey Xu Yu. Efficient discovery of frequent correlated subgraph pairs. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 239–248. IEEE, 2009.

[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.

[KK95] George Karypis and Vipin Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[KLCL13] Song-Hyon Kim, Kyong-Ha Lee, Hyebong Choi, and Yoon-Joon Lee. Parallel processing of multiple graph queries using MapReduce. In *DBKDA*, 2013.

[KLKF98] Flip Korn, Alexandros Labrinidis, Yannis Kotidis, and Christos Faloutsos. Ratio rules: A new paradigm for fast, quantifiable data mining. In *VLDB*, 1998.

[Kor08] Richard E Korf. Minimizing disk I/O in two-bit breadth-first search. In *AAAI*, pages 317–324, 2008.

[KRS88] Clyde Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Automata, Languages and Programming*, pages 333–346, 1988.

[KS96] Micheline Kamber and Rajjan Shinghal. Evaluating the interestingness of characteristic rules. In *KDD*, pages 263–266, 1996.

[KS11] Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–234. ACM, 2011.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. Society for Industrial and Applied Mathematics, 2010.

[KWAY13] Arijit Khan, Yinghui Wu, Charu C Aggarwal, and Xifeng Yan. Nema: Fast graph search with label similarity. In *Proceedings of the VLDB Endowment*, volume 6, pages 181–192. VLDB Endowment, 2013.

[LAR00] Weiyang Lin, Sergio A Alvarez, and Carolina Ruiz. Collaborative recommendation via adaptive association rule mining. *Data Mining and Knowledge Discovery*, 6:83–105, 2000.

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[LGK+10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. arxiv preprint. *arXiv preprint arXiv:1006.4990*, 1, 2010.

[LHKL12] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*, volume 6, 2012.

[Lib13] Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

[Lit] Joey Little. Who do you trust? 92advertising. https://www.linkedin.com/pulse/who-do-you-trust-92-consumers-peer-recommendations-over-joey-little.

[liv] Snap. *http://snap.stanford.edu/data/index.html*.

[LKDL12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012.

[LLP10] Clemens Ley, Benedikt Linse, and Olga Poppe. SPARQLog: SPARQL with rules and quantification. *Semantic Web Information Management: A Model-Based Perspective*, page 341, 2010.

[LQLC15] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in MapReduce. *PVLDB*, 8(10):974–985, 2015.

[LSZD13] Vitaliy Liptchinsky, Benjamin Satzger, Rostyslav Zabolotnyi, and Schahram Dustdar. Expressive languages for selecting groups from graph-structured data. In *Proceedings of the 22nd international conference on World Wide Web*, pages 761–770. ACM, 2013.

[LTP07] Stephane Lallich, Olivier Teytaud, and Elie Prudhomme. Association rule interestingness: Measure and statistical validation. In *Quality measures in data mining*, pages 251–275. 2007.

[LZ11] Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, 2011.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[Men04] Leandro Mendoza. A rule-based approach to address semantic accuracy problems on linked data. 2004.

[Mil13] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[MMS14] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 529–542. ACM, 2014.

[mov] Movielens. *http://grouplens.org/datasets/movielens/*.

[MZ11] Hamid Mousavi and Carlo Zaniolo. Fast and accurate computation of equi-depth histograms over data streams. In *EDBT*, 2011.

[MZL12] Seth A Myers, Chenguang Zhu, and Jure Leskovec. Information diffusion and external influence in networks. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 33–41. ACM, 2012.

[Pap03] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[PD07] Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *AAAI*, volume 7, pages 913–918, 2007.

[PH02] Jian Pei and Jiawei Han. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explorations*, 4(1), 2002.

[Pla13] Todd Plantenga. Inexact subgraph isomorphism in MapReduce. *J. Parallel Distrib. Comput.*, 73(2):164–175, 2013.

[PNK+11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25. ACM, 2011.

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

[PS04] Bijan Parsia and Evren Sirin. Pellet: An OWL DL reasoner. In *ISWC-Poster*, volume 18, 2004.

[QYC+14] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2014.

[RFRS14] Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. *ACM SIGPLAN Notices*, 49(10):909–927, 2014.

[RMM15] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2015.

[RPG+13] Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60. IEEE, 2013.

[RR96a] Ganesan Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[RR96b] Ganesan Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1):233–277, 1996.

[RVDB04] Cristóbal Romero, Sebastián Ventura, and Paul De Bra. Knowledge discovery with genetic programming for providing feedback to courseware authors. *UMUAI*, 14(5):425–464, 2004.

[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 *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[RW15] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628, 2015.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Acm Sigmod Record*, volume 25, pages 1–12. ACM, 1996.

[Sah07] Diptikalyan Saha. An incremental bisimulation algorithm. *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pages 204–215, 2007.

[SCC+14] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, 2014.

[SDNR07] Warren Shen, AnHai Doan, Jeffrey F Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[SHJS06] Christoph Schmitz, Andreas Hotho, Robert Jäschke, and Gerd Stumme. Mining association rules in folksonomies. In *Data Science and Classification*, pages 261–270. 2006.

[SK12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.

[SKW07] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[SMGW11] Mauro San Martın, Claudio Gutierrez, and Peter T Wood. Snql: A social networks query and transformation language. *cities*, 5:r5, 2011.

[Smi13] C Smith. Twitter users say they use the site to influence their shopping decisions. *Business Insider Intelligence*, 2013.

[SQC14] Prakash Shelokar, Arnaud Quirin, and Óscar Cordón. Three-objective subgraph mining using multiobjective evolutionary programming. *JCSS*, 80(1):16–26, 2014.

[SSW09] Fabian M Suchanek, Mauro Sozio, and Gerhard Weikum. Sofie: a self-organizing framework for information extraction. In *Proceedings of the 18th international conference on World wide web*, pages 631–640. ACM, 2009.

[ST93] David B Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1-3):461–474, 1993.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[SW13] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[SWL12] Bin Shao, Haixun Wang, and Yatao Li. The trinity graph engine. *Microsoft Research*, page 54, 2012.

[SWW$^+$12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.

[TBC$^+$13] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[TLX13] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 529–540. ACM, 2013.

[tra] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml*.

[Tri89] Phil Trinder. *A functional database*. PhD thesis, University of Oxford, 1989.

[tru] Nielsen global online consumer survey. *http://www.nielsen.com/content/dam/corporate/us/en/newswire/uploads/2009/07/pr_global-study_07709.pdf*.

[UKBM11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val91] Leslie G Valiant. *General purpose parallel architectures*. MIT press, 1991.

[VJG14] João Vinagre, Alípio Mário Jorge, and João Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *International Conference on User Modeling, Adaptation, and Personalization*, pages 459–470. Springer, 2014.

[WXDG13] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[WZZ04] Xindong Wu, Chengqi Zhang, and Shichao Zhang. Efficient mining of both positive and negative association rules. *TOIS*, 22(3), 2004.

[XCG+15] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[XCYH06] Dong Xin, Hong Cheng, Xifeng Yan, and Jiawei Han. Extracting redundancy-aware top-k patterns. In *KDD*, 2006.

[XGFS13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[YCLN14] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[YCX+14] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[YDCL06] Wan-Shiou Yang, Jia-Ben Dia, Hung-Chi Cheng, and Hsing-Tzu Lin. Mining social networks for targeted advertising. In *HICSS*, 2006.

[YH11] Yang Yu and Jeff Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *ISWC*. 2011.

[ZKS+13] Amrapali Zaveri, Dimitris Kontokostas, Mohamed A Sherif, Lorenz Bühmann, Mohamed Morsey, Sören Auer, and Jens Lehmann. User-driven quality evaluation of dbpedia. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 97–104. ACM, 2013.

[ZLL+15] Yang Zhou, Ling Liu, Kisung Lee, Calton Pu, and Qi Zhang. Fast iterative graph computation with resource aware graph parallel abstractions. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 179–190. ACM, 2015.

[ZRM+13] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, Sören Auer, and Pascal Hitzler. Quality assessment methodologies for linked open data. *Submitted to Semantic Web Journal*, 2013.

[ZZ02] Chengqi Zhang and Shichao Zhang. *Association rule mining: models and algorithms*. Springer-Verlag, 2002.