ABSTRACT

Title of dissertation: DECLARATIVE CLEANING, ANALYSIS, AND QUERYING OF GRAPH-STRUCTURED DATA

Walaa Eldin Moustafa, Doctor of Philosophy, 2013

Dissertation directed by: Professor Amol Deshpande, Professor Lise Getoor, Department of Computer Science

Much of today's data including social, biological, sensor, computer, and transportation network data is naturally modeled and represented by graphs. Typically, data describing these networks is observational, and thus noisy and incomplete. Therefore, methods for efficiently managing graph-structured data of this nature are needed, especially with the abundance and increasing sizes of such data.

In my dissertation, I develop declarative methods to perform cleaning, analysis and querying of graph-structured data efficiently. For declarative cleaning of graph-structured data, I identify a set of primitives to support the extraction and inference of the underlying true network from observational data, and describe a framework that enables a network analyst to easily implement and combine new extraction and cleaning techniques. The task specification language is based on Datalog with a set of extensions designed to enable different graph cleaning primitives. For declarative analysis, I introduce 'ego-centric pattern census queries', a new type of graph analysis query that supports searching for structural patterns in

every node's neighborhood and reporting their counts for further analysis. I define an SQL-based declarative language to support this class of queries, and develop a series of efficient query evaluation algorithms for it. Finally, I present an approach for querying large uncertain graphs that supports reasoning about uncertainty of node attributes, uncertainty of edge existence, and a new type of uncertainty, called identity linkage uncertainty, where a group of nodes can potentially refer to the same real-world entity. I define a probabilistic graph model to capture all these types of uncertainties, and to resolve identity linkage merges. I propose 'context-aware path indexing' and 'join-candidate reduction' methods to efficiently enable subgraph matching queries over large uncertain graphs of this type.

DECLARATIVE CLEANING, ANALYSIS AND QUERYING OF GRAPH-STRUCTURED DATA

by

Walaa Eldin M. Moustafa

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Advisory Committee:
Professor Amol Deshpande, Chair/Co-Advisor
Professor Lise Getoor, Co-Advisor
Professor Atif Memon
Professor Ashok Agrawala
Professor Eyad Abed

© Copyright by
Walaa Eldin M. Moustafa
2013

Dedication

To my great parents.

To my wonderful wife, Ethar.

To my lovely kids, Yaseen and Maryam.

Acknowledgments

The Arabic proverb says "Choose the companion before you choose the journey". The PhD has been a long journey and I was fortunate to have the greatest companions during that journey, without whom, I could not reach the end.

I am deeply indebted to my advisors, Prof. Amol Deshpande and Prof. Lise Getoor, who provided me with enduring guidance and support. They inspired me with their wisdom, righteous judgement, hard work, attention to detail and pursuit of perfection. They challenged me to bring forth my best and pursue what is beyond possible. I always found something to learn from them, far beyond what is in the books, or on the boards. Their advice did not know boundaries and they were always there ready to give.

I am greatly thankful to Prof. Atif Memon, Prof. Ashok Agrawala, and Prof. Eyad Abed, who agreed to serve on my dissertation committee, and to Prof. Nick Roussopoulos, who led the Database Design courses that I taught with him. Throughout my PhD career, they all kindly spent informal time with me discussing my research ideas and PhD life. Their words have always been reassuring and encouraging.

If I want to thank my wife, Ethar, words will fall short of expression. She always did before I expected. She always gave before I asked, and she always sacrificed when she could have other options. She cared for everyone in our small family, and preferred us to herself; that is where her name comes from, "إيثار".

I would like to thank all my group mates and collaborators, Hossam, Galileo,

Hui, Lilly, Angelika, Mustafa, Prithvi, Elena, Bhargav, Thodoris, Jayanta, Udayan, Abdul, Ashwin, Steve, Ben, Jay, Bert, Alex, Shobeir, Arti, Souvik, Amit, for all their insightful discussions and sparking ideas we shared together. I will always miss my wonderful colleagues and the brightest teammates I have ever met.

I am lucky to have a group of great friends. M. Abdelkader, Hossam, Mostafa, A. Khalil, M. Raafat, Omar, Tarek, M. Fahmi, Wael, Karim, T. Elsharnouby, Hatem, A. Mansy, T. Elsayed, M. Hussein, Hazem, and M. Youssef are real brothers. We shared a lot, and counted on each other a lot. They have always been my source of strength and reinforcement.

I am grateful to my son and daughter, Yaseen and Maryam, whose smiles gave me the light and the power to achieve.

I am in debt to my parents, and my parents-in-law, who spent endless efforts, all over the years, providing me and our family with all the love, help, and care. They always gave without return, and served when it was their turn to be served.

Thank you all and thank you God for all your countless blessings!

Table of Contents

List of Figures

List of Abbreviations

GraphQL	Graph Query Language
ICA	Iterative Classification Algorithm
PRM	Probabilistic Relational Model
RMN	Random Markov Network
MLN	Markov Logic Network
PSL	Probabilistic Soft Logic
RDF	Resource Description Framework
XML	Extensible Markup Language
PGM	Probabilistic Graphical Model
PEG	Probabilistic Entity Graph

Chapter 1

Introduction

As more data is available from different information sources such as the Web, social media, communication networks, software repositories, citation and collaboration networks, there is an emerging need to query and analyze such data. Much of the data in these domains expresses complex relationships between objects, making it natural to model it as "graphs". Existing relational database management systems are inadequate for querying or analyzing graph data for a variety of reasons:

- **Graph traversal:** The vast majority of graph algorithms and operations rely on traversing the graph structure. Relational models are not well suited for such types of operations because in the relational model, graph traversals are interpreted into joins, resulting in a large number of joins per query. Since joins in relational databases are very expensive, evaluating graph-specific queries over relational databases is usually inefficient.

- **Indexing methods:** Indexing techniques designed for relational data cannot be applied directly to graph-structured data. In graph data, there is a need to index *sub-structures* of a graph such as small paths, subtrees, or subgraphs in order to use them as structural features of the graph containing them. However, relational database indexes are designed to store primary data types such as numbers and strings.

- **Recursion:** Graph queries are naturally recursive. For example, a graph path is defined as being either an edge or an edge connected to a path (by recursion). There are other graph concepts that are recursive too such as reachability, connected components, and transitive closure. Although there are attempts at extending relational databases to handle recursion, these solutions are still unnatural and difficult to use.

At the same time, such data is often noisy and incomplete due to different reasons:

- **Missing information or errors from the source:** Sometimes, the data is missing from the source itself, or machine/human errors take place while entering the data. For example, location information of mobile device users is sometimes inaccurate or missing due to limitations in the technology used for location determination. Another example is in social network data, where user data is usually missing some of the details that users do not wish to disclose.

- **Data extraction errors:** Although much of the data existing on the Web is publicly available, access to the underlying database sources is usually restricted. To extract data directly from web pages, methods such as web crawling and screen-scraping are used. However, those methods suffer from being slow, error-prone, and hard to maintain, leading to various problems in the quality of the extracted data. Furthermore, even after extracting the raw data by those methods, in order to make the data suitable for further analysis, natural language processing techniques are utilized to convert the natural lan-

guage text into other more structured formats like RDF triples or relational tables. Many of those approaches depend on domain knowledge, require human intervention, and are still open research problems, e.g., [1].

- **Data duplication errors:** Real-world objects may have multiple representations or *references* in the data. These references may be of different forms, but in fact they all refer to the same real-world entity. For example, two references to the same person may differ in their spelling, or the way the name is abbreviated. Furthermore, a single user of an online service may have multiple accounts, leading to the impression that there are multiple users, while there is actually one. Before processing the data such duplicate references need to be detected, resolved, and combined into one entry representing the underlying real-world object.

- **Data integration errors:** When integrating data from multiple sources, different sources may disagree on some facts, and hence, conflicts and uncertainties arise.

Given the limitations of standard declarative database management systems for dealing with graph data, and the quality issues occurring with this type of data, in this dissertation, we address the problems of declaratively and efficiently cleaning, analyzing, and querying large graph-structured data. Before discussing the challenges facing these areas of research and the contributions made by this dissertation, we begin by discussing some motivating applications.

1.1 Motivating Applications

There are various applications where cleaning, analyzing, and querying graph data are important operations, such as analyzing call traffic between customers of telephone companies, querying social networks for connections among people, cleaning spam from email archives, etc. Below we discuss the motivating applications for our research using examples in two domains.

1.1.1 Scientific Publication Networks

Consider a scientific publication network extracted from online bibliographic information-archiving websites such as DBLP, CiteSeer, or PubMed. The publication network consists of different node and edge types. Nodes are of type scientist, publication, and conference, representing the three types of objects in this network. Edges are of type *wrote* between scientists and their publications, *appeared in* between publications and the conferences they appeared in, and *cites* among publications indicating that a publication cites another. In the following paragraphs, we discuss examples of cleaning, analyzing and querying tasks that are of interest over such kinds of networks.

1.1.1.1 Graph Data Cleaning

Data cleaning is the process of correcting inaccurate entries and predicting missing information in a database. Common types of graph data inaccuracies include missing nodes or edges, missing node or edge attribute information, and the

existence of multiple nodes that refer to the same real-world object. There has been a significant amount of work in the database, machine learning, and information retrieval communities to deal with these types of inaccuracies. *Attribute prediction* is used for predicting values of node attributes in graph data. *Link prediction* is used for inferring the existence of missing links between nodes. *Entity resolution* is used for detecting multiple references in the data that refer to the same real-world object. Below we discuss some examples of performing those operations on collaboration network data.

- **Attribute Prediction:** Given a partially labeled publication network, we may want to infer the research interests of the scientists from their publication titles, or from the research interests of their collaborators. Other examples include classifying publication topics using both the publication title and the citations.

- **Link Prediction:** We may want to predict potential collaborations between scientists based on the current information. Furthermore, we may be interested in suggesting a candidate set of publications to cite by a new publication using a seed set of citations. In case the observed data set is incomplete, we may want to predict missing edges of different types in the graph.

- **Entity Resolution:** A common problem in online bibliographic archives is the problem of listing the same author as two different authors just because her name is spelled slightly differently on different articles. Performing entity resolution on such entries helps gathering the same author's information in

one place rather than being scattered at different locations. This problem also occurs with conference names, especially when the data is gathered from multiple sources – sometimes the names are abbreviated, and sometimes the conference full names are stated. Figuring out sets of nodes that refer to the same conference is an important step before performing any further analysis.

1.1.1.2 Graph Data Analysis

Generally, there are two main types of graph data analysis (a) global macro-level analysis characterizing properties of the entire graph and its subgraphs, or (b) local micro-level analysis characterizing properties of the nodes in the graph. Macro-level analysis is important for characterizing graph properties and understanding network evolution; examples of such analysis include measuring structural properties like degree distribution, diameter, and graph cohesion [2], and discovery of patterns or *motifs* in the network [3, 4, 5, 6, 7, 8]. Micro-level analysis focuses instead on measuring properties of the nodes in a graph, e.g., degree centrality, second-order degree, local clustering coefficient etc. This is often called *ego-network* analysis, because it looks at the individual nodes and their neighbors in a graph. Although global network analysis is well-understood and efficient computational tools are well-developed, similar tools are not yet available for the harder problem of ego-centric analysis that requires analyzing a very large number of small, largely overlapping graphs. In this dissertation, we focus on the second type of graph analysis. Below we discuss some of its motivating applications in the case of analyzing scientific

publication networks.

- Properties of individual nodes can be studied with respect to their neighborhoods or *ego-networks*, where an ego network of a node is the local subgraph surrounding this node. For example, to obtain information about conferences and their attendees, we need to extract the two-hop neighborhoods around conferences, which are referred to as *two-hop ego-networks*. We can study the role scientists play with respect to their collaborators by extracting the two-hop ego-network around each scientist consisting of publications of each scientist, and the collaborators who worked on those publications as well. Different types of analysis can be performed on those ego-networks including studying their size, density, and the central node's centrality and brokerage scores. We will discuss the details of these measures in Chapter 4.

- Local node neighborhoods contain valuable information that can be used for predicting missing values in graph data. For example, the structure of node neighborhoods affects values node attributes can take (e.g., if a scientist is interested in Computer Science, then with a high probability, her collaborators are interested in Computer Science as well, etc.). Furthermore, similarity between neighborhood structures of pairs of nodes can indicate that the two nodes refer to the same underlying object.

- Summaries of node neighborhoods, especially the occurrence of specific patterns, can be used as a pre-processing step for evaluating graph queries. By summarizing neighborhood information around a node, one can determine be-

forehand if a node can be a candidate for the results of a graph query, before evaluating more complex conditions. For example by summarizing two-hop information around scientists, we can obtain statistics that summarize the set of conferences each scientist participates in. When querying for scientists who participated in specific conferences, each scientist's summary information can be directly looked up instead of traversing the graph to evaluate the conditions.

1.1.1.3 Graph Data Querying

- **Subgraph Pattern Matching Queries over Certain Graphs:** There are various types of queries that can be posed on graph-structured data. Querying graph data ranges from explaining how a set of nodes are connected in a graph, or finding a set of nodes that are most similar to a group of query nodes, or searching graph data for keywords. In this dissertation, we focus on one of the most widely used and studied type of graph queries, which are subgraph pattern matching queries. Subgraph pattern matching queries find subgraphs in a large graph that satisfy a specific criteria expressed by a pattern graph. The criteria can be on the node types and attributes, edge types and attributes, and the structure of the subgraph itself. For example, Figure 1.1 shows two graph patterns that we may be interested in finding matches for in a scientific publication network. The query in Figure 1.1(a) asks for all publications written by a scientist whose last name is "Hellerstein" that were published in a SIGMOD conference, and involved a collaboration with a scientist whose main

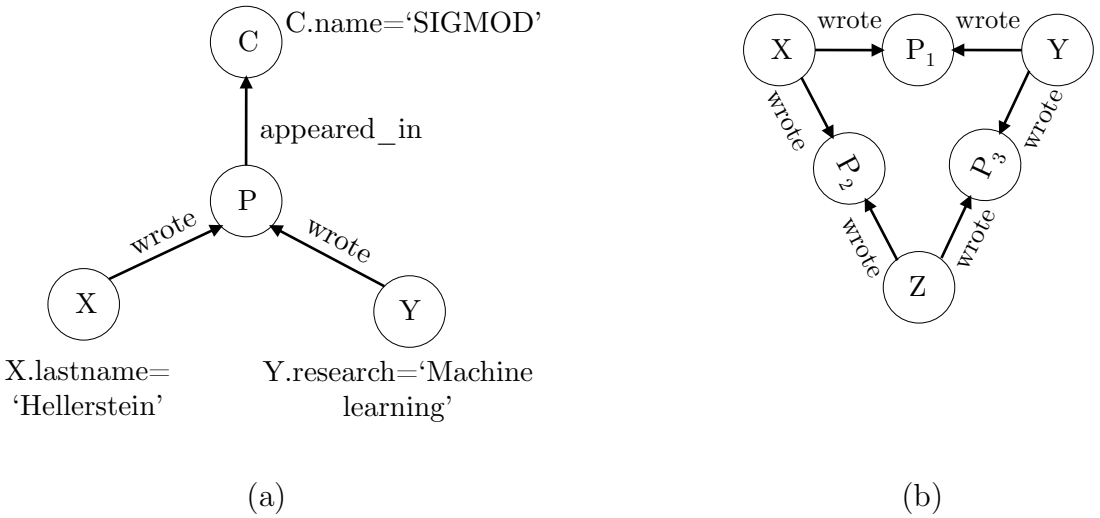


Figure 1.1: Example patterns used to query a scientific collaboration network

area is Machine Learning. The query in Figure 1.1(b) asks for all triplets of scientists where each pair of them has collaborated on a different publication. Answers to that query may indicate that the three scientists may collaborate together on a joint project.

- **Subgraph Pattern Matching Queries over Uncertain Graphs:** As discussed at the beginning of this chapter, graphs extracted from different sources are likely to have different types of noise or errors. Furthermore, the data may have duplicate information, i.e., sets of nodes that refer to the same real world entity, while queries over such data require reasoning at the real-world entity semantics. Therefore, it is useful to express and encode different types of uncertainty in a probabilistic model, and perform *soft* querying over such *uncertain* graphs, taking into consideration that multiple nodes may just be references to the same entity. In that case, only matches whose probability

passes a given user-specified threshold are returned.

In the case of scientific collaboration networks, incorporating uncertainty is useful, especially since some of the attributes are fuzzy by nature. Scientist research interests are usually not very well defined, and there is a fine line that distinguishes between them. The same applies to publication topics. Furthermore, a scientist or a conference may be represented by different nodes due to name variations; thus linking scientists (or conferences) who have similar names with probabilistic edges is a concise way to express uncertainty regarding duplicate information. Therefore, there is a need to query graphs which contain all those types of uncertainty, where only highly probable answers of the queries are returned.

1.1.2 Targeted Marketing

In targeted marketing applications, advertising companies carefully pick some customers to give them free samples or free goods so that they can attract other customers to their advertised products. Consider a travel agency that provides special trips for couples and obtains online social network information of some users to utilize it for advertising.

1.1.2.1 Graph Data Cleaning

Several graph cleaning operations can be applied before the social network information can be used for further querying or analysis. Examples of these operations

are as follows.

- **Attribute Prediction:** One step towards identifying potential customers is to find out some of their attributes if they are missing. For example, it is beneficial to predict attributes such as living place, gender (male/female), etc, that may be missing for some customer. In addition, we may be interested in predicting unobservable attributes such as income-range or preferences for vacation destinations.

- **Link Prediction:** It is also beneficial to predict the nature of relationships between the social network users. It would be interesting to predict links whose type is "engaged to" or "married to".

- **Entity Resolution:** One factor negatively affecting the utility of using online data for marketing is the fact that some users may have multiple accounts, which are actually sometimes created to delude the service into thinking that they are different users so that the same person can get additional benefits or products. An effective way to overcome that is by performing entity resolution on the social network data to discover which users have multiple identities.

1.1.2.2 Graph Data Querying

The company may be interested in searching for all instances of "couples" or "pairs of couples". Therefore, subgraph pattern matching queries in Figures 1.2(a) and 1.2(b) can be issued on this network to search for all instances that have these types of connections among them. If there is uncertainty in the data (e.g., because

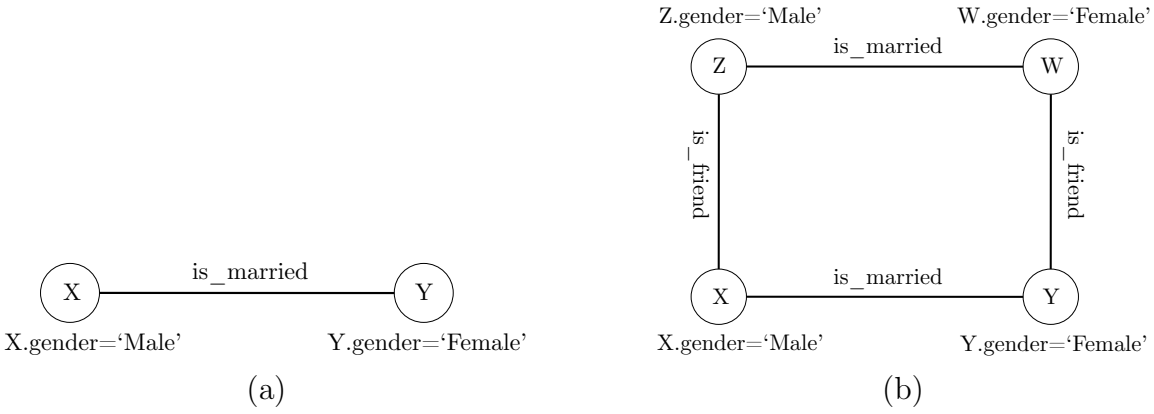


Figure 1.2: Example patterns used to query a social network for targeted marketing

a portion of the network attributes was predicted), then subgraph pattern matching queries over uncertain graph data are useful in determining only highly probable matches to these queries.

1.1.2.3 Graph Data Analysis

The company may want to choose some consumers to get free trips or promotions. These consumers must be chosen wisely to minimize the cost and maximize the benefits of advertising. Simple criteria such as picking consumers with the most friends, or consumers that are connected to many other consumers through short paths, are typically used. However the ability to identify arbitrary structures is desirable in many cases. For example, the travel agency may wish to identify couples that have either the largest number of couples in their combined network, or the largest number of couple pairs, i.e., couples who are friends with couples. The latter structure is depicted in Figure 1.2(b).

1.2 Overview of Dissertation Research

In this section, we give a brief overview of our work in the areas of graph-structured data cleaning, analysis and querying.

1.2.1 Graph Data Cleaning

To perform graph data cleaning tasks, we present the design and architecture of a data management system that enables efficient, declarative cleaning of large-scale information networks. The goal is to provide a declarative framework for common operations required in cleaning and extracting networks, a mechanism for combining them in various ways, and an implementation for efficiently applying them to large observation networks. The challenges for building this system are:

- Network cleaning operations do not depend only on the local information available at the graph nodes or edges but also are heavily dependent on the actual graph structure and typically require traversal of the node neighborhoods and computation of structural features. The declarative language must be able to express both local and structural information in an intuitive way, and the implementation must take into consideration those types of computation.

- Most network cleaning techniques are inherently *iterative*, and require repeated passes over the graph. Therefore, supporting fast iterative graph processing is important.

- Network cleaning often needs to be "collective" (where a decision in one part

of the network affects the information flow in other parts of the network), and therefore, there must be a means to determine the order of application of cleaning decisions, as they will probably affect future ones.

- It is hard to find an open-source or easy-to-modify query engine that has all or most of the required features of such a system. This necessitates building the evaluation engine from scratch and implementing all of the optimizations from the low level upwards.

- The system should be extensible to support different kinds of prediction models such as logistic regression and Bayesian inference.

- Systems supporting graph data cleaning should be scalable to support graph data with increasing sizes within a reasonable period of time.

We propose a declarative approach to efficiently perform a variety of graph cleaning operations. The key to our approach is to *decouple* the graph traversal operations from the modification operations; the traversal operations are typically computationally expensive, especially for large disk-resident graphs. We present a declarative language based on *Datalog*, and show how it can be used to cleanly achieve such decoupling. This decoupling enables us to develop a framework for declarative cleaning over large networks, and facilitates efficient execution, by allowing us to push much of the computation inside a database system. Further, the declarative framework allows us to efficiently incorporate, and propagate through the cleaning task, dynamic updates to the network data. We have implemented a

system called GRDB that supports our declarative framework. Our results illustrate the computational and usability advantages of our system.

1.2.2 Graph Data Analysis

For graph data analysis, we address a new problem in ego-centric analysis, namely counting the number of structural patterns or motifs that occur either in the k-hop neighborhoods of the nodes (for a given k), or in subgraphs defined by intersections or unions of k-hop neighborhoods of pairs of nodes; k-hop neighborhood of a node n is defined to be the incident subgraph on the nodes reachable from n in k hops or less. We refer to this type of query as an *ego-centric pattern census*. To achieve that, we devise a simple yet powerful query language to support this kind of queries. The query language allows specifying predicates to select which nodes to perform the analysis on, and allow specifying the neighborhood sizes to be searched. Furthermore, it supports specifying arbitrary structural patterns to search for in the neighborhoods. We develop efficient algorithms and indexing techniques to evaluate this type of queries. The challenges for designing such a system are as follows:

- It is known that the problem of subgraph pattern matching is NP-complete. Ego-centric pattern census problem is even harder because we need to perform subgraph isomorphism in the local neighborhoods of $|V_G|$ nodes where $|V_G|$ is the number of nodes in the graph.

- Currently existing system for ego-centric graph analysis only study very simple measures, such as the node degree, two-hop reach, number of triangles, or ego-

centric betweenness. It is not easy to extend these systems to support more complex and general analysis operations over the node neighborhoods, because firstly, they are not declarative, and secondly, algorithms for finding subgraph patterns in node neighborhoods require a completely different set of algorithms and methods than those reporting simple neighborhood counts.

1.2.3 Graph Data Querying

1.2.3.1 Querying Certain Graphs

We develop an efficient algorithm for subgraph pattern matching over large graphs. The algorithm finds all instances in of an input pattern query in a large graph. The algorithm is highly scalable and outperforms a current state of the art algorithm [9] by orders of magnitude. The challenges are as follows.

- Given the difficulty of the subgraph isomorphism problem, and the size of today's graphs in different applications, it is challenging to answer subgraph pattern matching queries over large graphs in a timely fashion.

- Current systems such as relational databases are not suitable to support such type of queries because both their query evaluation and indexing methods are designed to deal with relational, tabular data, not data that is laid out as graphs.

- The search space of this problem is prohibitively large. Without optimizations, the search space is in the order of $|V_G|^{|V_Q|}$, where $|V_G|$ is the number of nodes

in the database graph, and $|V_Q|$ is the number of nodes in the query graph. Therefore, effective optimizations have to be devised in order to reduce the search space significantly.

1.2.3.2 Querying Uncertain Graphs

Although graph cleaning approaches can be used to handle uncertainty and noise in graph data before querying or analyzing it, it is not always possible to remove all the uncertainties in the graph using such approaches. In that case, instead of throwing away the uncertainties, a more desirable approach would be to model the uncertain and noise directly by associating probability distributions with graph data and then using a probabilistic model to query the uncertain graph directly. Therefore, we address the problem of reasoning about and operating on uncertain graphs with identity uncertainty and develop techniques for efficiently answering subgraph pattern queries over them. Our graph model combines node attribute uncertainty (i.e., nodes have probability distributions over attribute values), edge existence uncertainty (i.e., edges have a probability of existence), and identity uncertainty (i.e., the probability that a group of references combine to form a single entity). We show that our model defines a probability distribution over possible graphs describing entities, their labels and relations.

We then introduce techniques to find all matches of a subgraph pattern that have a probability above a given threshold. Answering subgraph pattern matching queries is NP-hard on non-probabilistic graphs. It becomes even harder when

adding uncertainty, especially identity uncertainty, which requires reasoning about constraints on sets of nodes that can exist together, and thus makes the problem #P-complete. Nonetheless, we propose and systematically explore a range of novel techniques to prune the search space and effectively perform subgraph pattern matching over large-scale uncertain graphs. The challenges are as follows:

- Usually the observed data is at the reference-level, while when querying the data, users are interested in the entity-level semantics. Sophisticated models have to be developed in order to define the entity semantics and enable reasoning over them.

- Systems for query evaluation on probabilistic relational databases are not suitable for querying uncertain (probabilistic) graph databases, due to the huge search space that will be considered by these systems when dealing with graphs. Furthermore, identity uncertainty introduces a special type of constraints and merge operations that have to be considered during query evaluation. Probabilistic graph databases do not have a built-in support for such type of operations.

- The search space for uncertain graph pattern matching is even larger than that of certain graphs, because in addition to the complexity of solving subgraph isomorphism problem, one has to solve in all the possible worlds of the input database graph. There is an exponential number of possible worlds in terms of the graph number of uncertain nodes, edges, and attributes. This makes exploring all these possible worlds prohibitive, and therefore devising efficient

techniques is a requirement.

1.3 Outline and Contributions

In Chapter 2, we review related work on the problems of graph-structured data cleaning, analysis, and querying (in both certain and uncertain domains).

In Chapter 3, we present our proposed approach for declarative graph data cleaning based on Datalog. We propose a unifying framework to specify different data cleaning tasks, and implement efficient algorithms to execute them. The main contributions are:

1. We identify the commonalities between different graph extraction and cleaning tasks and derive a decoupling that enables efficient integration of these tasks.

2. We propose an interface for specifying network cleaning tasks that makes it easy for the users (network analysts) to experiment with different methods and combinations of features to decide how to best analyze and clean a network. The proposed framework supports defining prediction domains, features, and functions, which allows a declarative interface for the coupled inferences required for network cleaning.

3. We present several extensions to Datalog giving it operational semantics rather than fixed-points semantics, where necessary, to make it more suitable for graph cleaning operations.

4. We develop algorithms for efficiently computing the features, and for incre-

mentally maintaining them in the presence of updates (a requirement given the iterative nature of the algorithms used). The proposed techniques can handle a wide variety of Datalog queries including aggregate and outer-join queries.

5. We present the results of an experimental study over several real and synthetic datasets.

In Chapter 4, we investigate ego-centric pattern census queries, and its closely related problem, subgraph pattern matching over certain graphs. We develop efficient techniques for executing both operations. The key contributions include:

1. We introduce a flexible declarative SQL-based query language for specifying ego-centric pattern census queries. The proposed query language allows users to specify the **neighborhood** size (k), the set of **focal nodes** (or pairs of nodes), and the pattern to be counted.

2. We propose an efficient graph pattern matching algorithm, and show that it outperforms GraphQL [9], a recent graph pattern matching system.

3. We introduce two query evaluation algorithms for the ego-centric census queries, one based on searching from nodes to patterns (*node-driven*) and another based on searching from patterns to nodes (*pattern-driven*).

4. We empirically evaluate the proposed algorithms on a variety of real-world and synthetic data.

In Chapter 5, we study the problem of querying uncertain graphs with identity uncertainty. We present our proposed approach for modeling graphs with attribute uncertainty, edge existence uncertainty, and identity uncertainty. We propose an efficient algorithm to perform subgraph pattern matching queries over such uncertain graphs. The main contributions are:

1. We introduce *probabilistic entity graphs*, a general uncertain graph model that captures label, edge and identity uncertainties.

2. We define the semantics of probabilistic entity graphs as a probability distribution over possible entity graphs.

3. We develop scalable algorithms to answer subgraph pattern matching queries over such uncertain graph data, based on *query path decomposition.context-aware path indexing*, to capture information about the graph paths, their surrounding structures, and their probabilities, enabling efficient retrieval of candidate matches.

4. We propose *reduction by join-candidates*, an algorithm that efficiently prunes candidate answers by progressively propagating structural and probabilistic information between the candidates. This approach reduces the search space size by *multiple orders of magnitude*.

5. We demonstrate that the proposed approaches can evaluate complex queries over graphs with millions of nodes and edges in seconds, outperforming a baseline implementation by *multiple orders of magnitude*.

Chapter 2

Related Work

There has been much work on graph databases, with renewed interest in recent years due to the popularity of social networks and other types of information that can be represented as graphs, and the growth of graph sizes representing these networks. That has led to much research in different areas related to processing, storing, indexing, analyzing, querying, and learning over graph data. In this chapter, we review related work to our proposed methods in declarative graph cleaning, analysis and querying for both certain and uncertain graphs.

2.1 Graph Data Cleaning

We begin by discussing related work to the main three problems we are interested in in the context of data cleaning, i.e., attribute prediction, link prediction, and entity resolution, then we discuss related work to declarative methods in these areas.

- **Attribute prediction:** Missing attributes of nodes in graph data can be predicted based on the values of other attributes of the node or based on the predicted neighbors' attribute values, i.e., collective classification. The underlying assumption in collective classification is that the links between nodes carry important information for inferring the attribute values. In many

cases, there is auto-correlation between the labels of the nodes [10], which means that linked nodes are likely to share the same attribute values, but other, more complex correlations can be modeled and exploited. A summary of various methods used for attribute predication in relational settings can be found in [11]. Generally, approaches for attribute prediction are divided into local formulations and global formulations. Local formulations predict the value of a node attribute based on the node's information and its local neighborhood information. The most common approach for local attribute prediction is the *iterative classification algorithm*, ICA, (e.g., [12, 13, 14]). ICA employs features calculated from the nodes' neighborhoods and can use any general classification function such naive Bayes or logistic regression for the prediction. The algorithm proceeds iteratively until convergence or until a specific number of iterations is reached. Some extensions of ICA such as [15, 16] apply only a subset of the predictions after each iteration, where predictions of higher confidence are chosen in earlier iterations. On the other hand, global attribute prediction formulations depend on defining a global function to optimize. Some formulations are based on Probabilistic Relational Models (PRMs) , e.g., [17], Random Markov Networks (RMNs), e.g., [18], Markov Logic Networks (MLNs), e.g., [19], or Probabilistic Soft Logic (PSL), e.g., [20].

- **Link prediction:** The link prediction problem [21] can be formulated as a classification problem where we associate a binary variable for each pair of

nodes which is true if a link exists between the two nodes and false otherwise. The simplest approach is to predict new links based on similarity measures between pairs of nodes [22] (e.g., number of common neighbors, Jaccard similarity, Adamic/Adar [23], Katz measure [24], random walks with restart [25], recursive similarity [26], etc.). Other link prediction formulations are based on RMNs, e.g., [27], MLNs, e.g., [19], or PSL, e.g., [28].

- **Entity resolution:** Entity resolution is the task of identifying when two nodes in the graph are referring to the same real-world entity. In this case, the nodes should be merged, and their attributes and links should be updated accordingly. Common approaches to entity resolution use a variety of similarity measures, often based on approximate string matching criteria [29, 30, 31]. These work well for correcting typographical errors and other types of noisy reference attributes. More sophisticated approaches make use of domain-specific attribute similarity measures and often learn such mapping functions from resolved data. Other approaches take graph structure and similarity into account [32, 33] and allow dependencies among the resolutions, e.g., *collective* entity resolution [34]. Global entity resolution formulation are based on MLNs, e.g., [35], or PSL, e.g., [20].

Namata et al. [36] propose a hybrid approach to perform joint inference between the different types of tasks, where they use coupled collective classifiers to propagate information among solutions to the problem.

The above approaches are related to the graph cleaning problem in general. There are two lines of research that are specifically related to the *declarative* specification aspect of our research; *declarative entity resolution* and *Datalog-based declarative specification*. We discuss each type of related work below.

Declarative Entity Resolution: In [30], the authors consider the problem of generic entity resolution, where they define entity resolution in terms of two functions, *match* that matches two records and *merge* that merges two records if they match. These two functions are treated as black-boxes, and the authors define classes for their properties, studying efficient algorithms for different classes. Our work is in a similar spirit, in attempting to define black boxes for the prediction problems; however we focus on a declarative specification for the interactions among the predictions, and efficient incremental maintenance.

In other related work, Arasu et al. [37] employ Datalog to solve the problem of collective entity resolution using domain-specific constraints. The constraints are in the form of user-defined soft and hard rules. The system performs the deduplication by satisfying all the hard rules and minimizing the number of violations to the soft rules. Our approach also supports a declarative approach toward collective entity resolution; however our approach is capable of performing a more general set of network inferences.

Datalog: The language that we use to enable declarative graph cleaning is based on *Datalog* [38]. Datalog has drawn the attention of many researchers since the introduction of ideas of integrating databases with logic as a standalone area in 1978,

leading to the emergence of *deductive databases*. Since then, there has been extensive research on Datalog, its semantics [39, 40], evaluation techniques [41, 42], and optimization [41], which led the Datalog query model to be extensively studied and well understood. The popularity of database models such as the relational model, object-oriented databases, and XML had drawn the attention away from deductive databases. However, recently there has been a renewed interest in this language because of its declarative nature, expressive power, and mathematical foundations. In recent studies, Datalog has been the centerpiece in enabling declarative specification in various domains, like network protocol specification [43], sensor networking [44], recommendation in social networks [45], and deduplication [37]. Recently, Seo et al. [46] proposed an approach for using Datalog to express graph analytics. They show an efficient implementation based on semi-naive evaluation. They do not consider issues of graph cleaning such as attribute prediction, link prediction and entity resolution, and their related issues such as incremental maintenance.

2.2 Graph Data Analysis

Analyzing graph-structured data has been a very effective tool for gaining insights regarding the graph properties, graph structure, node and link properties. There are many types of graph analysis queries such as ranking (e.g. PageRank [47], Hubs and Authority [48], and Betweenness Centrality [49]), clustering (e.g. edge betweenness [50] and modularity optimization [51]), similarity using various similarity measures (e.g. Jaccard coefficient, Lada/Adamic measure [23], and Katz

measure [24]), and local structural analysis, such as measuring node degrees, degree of homophily, two-hop reach, clustering coefficient, and ego-centric betweenness [52].

Our work in ego-centric graph data analysis is closely related to several active research topics that are being studied in different communities. As we discussed in Chapter 1, in social network analysis, distinction is often made between socio-centric analysis and ego-centric analysis. The former has seen much work over the last two decades with focus on understanding how networks evolve (see, e.g., [2, 53]), computing and reasoning about global or local properties of the networks, designing visualization tools to help with analysis (e.g., NodeXL [54]) and so on. In ego-centric analysis, instead the focus is typically on understanding how the structure of the neighborhood around a node affects the node or dictates its function. For example, *structural holes* in ego networks are considered indicative of the positional advantage or disadvantage of individuals [55, 56]. Although computational techniques for ego-centric analysis aren't as well-developed yet, there is increasing interest in understanding how to do ego-centric analysis more efficiently and several software packages support reasoning over ego networks (e.g., EgoNet [57]). Another related research area is the study of network motifs [3, 4, 5, 6]. Roughly speaking, network motifs are subgraphs that occur more frequently than expected to appear in a random network. Most real-world networks exhibit a small set of motifs that occur repeatedly in the network, and can be considered building blocks of the network. There is much work on efficiently counting the number of motifs that appear in a given network [58, 59, 60]. Although similar in spirit, our focus on counting motifs (generalized to allow predicates on the node or edge attributes) in all ego networks

requires us to develop new computational techniques to solve the problem.

2.3 Graph Data Querying

2.3.1 Querying Certain Graph Databases

In the area of certain graph databases, several query languages and query evaluation technqiues have been proposed to query and manage graph data including GraphLog [61], GOOD [62], GraphDB [63], GOQL [64], and PQL [65]. Also, there has been much work on algorithms and indexing methods to answer distance and reachability query, e.g., [66, 67, 68], connection subgraph queries, e.g. [69, 70], and clustering large graphs [71, 72].

Subgraph pattern matching is perhaps the most common type of graph query. There is much work on subgraph pattern matching with renewed interest in recent years. Several researchers have proposed exact or approximate methods for searching for patterns in graph databases consisting of several relatively small graphs as well as a single large graph (e.g., [73, 74, 75, 76, 77, 78, 79]). Examples of exact methods include GraphQL [9], GADDI [80], and SPath [81]. GraphQL [9] retrieves instances of subgraph patterns from a large database graph by indexing node neighborhoods, and using that to reduce the search space. GADDI [80] uses a distance index based on the number of discriminating substructures between pairs of nodes. Zhao et al. [81] propose an indexing technique that is based on neighborhood signatures and shortest paths. Other work in this area has focused on variants of the pattern matching problem. Fan et al. [82, 83] allow an edge in the pattern to represent

a short path in the database graph, and the matching is based on the concept of *bounded simulation*. Similarly, Zou et al. [84] propose *distance join* where the query is a pattern along with a distance δ. A match exists iff for two vertices v_i and v_j that are connected by an edge in the pattern, the shortest path between their images v'_i and v'_j is $\leq \delta$.

2.3.2 Querying Uncertain Graph Databases

Although there has been a lot of research studies addressing the problems of representing and querying uncertain and probabilistic data, e.g., [85, 86, 87, 88], the area of uncertain graph data processing is still new and gaining more interest recently. Research in uncertain graph databases has covered different areas such as finding shortest paths, reliable subgraphs, mining frequent patterns, and answering graph queries. For example, in [89] Potamias et al. address the problem of finding k-nearest neighbors in an uncertain graph. They extend the definition of shortest path and random walk in order to define meaningful distance functions that are suitable for uncertain graphs. Jin et al. [90] address a similar problem, which is distance-constraint reachability queries over uncertain graphs. Given an uncertain weighted graph, and a pair of nodes, a distance d and a probability threshold α, the question is to find whether the two nodes can be connected by a d-path with probability α or more. Furthermore, there has been work on extracting reliable subgraphs from uncertain graphs, e.g., Jin et al. [91] find sets of vertices whose induced subgraphs are reliable with some user specified threshold. Hintsanen et al. [92]

study the problem of fast discovery of reliable subnetworks in unreliable networks, where given a probabilistic graph and a pair of nodes, they propose algorithms to extract a subgraph with at most B edges such that the probability of a path existing between the two nodes is maximized. On the other hand, Zou et al. [93] propose efficient algorithms to find top-k maximal cliques in uncertain graphs, where top-k cliques are selected according to their probability. There has also been work on discovering frequent subgraphs in probabilistic graphs, i.e., given a graph database of uncertain graphs, the task is to find subgraphs which exist in at least Φ graphs with probability α or more [94, 95]. In the area of probabilistic graph querying, Chen et al. [96] propose algorithms to extract answers of subgraph queries from continuously changing graph streams. Since the problem is based on subgraph isomorphism which is proven to be NP-complete, the authors propose some approximations to the problem. Furthermore, the solutions are targeted for applications with large number of small graphs (in the order of tens of nodes and edges) rather than a large single graph. In addition, in our proposed work we plan to handle node attribute uncertainty in addition to edge uncertainty, which is the only type of uncertainty handled by that work. Yuan et al. [97] propose efficient algorithms to calculate the support of a subgraph pattern in uncertain graphs using pruning methods and deriving upper- and lower-bounds for the value of support. Our goal is different from that work as we aim to find highly probable matches, not the expected support of the query subgraph in the database graph. Udrea et al. [98] propose precise semantics for probabilistic RDF graphs formed by associating probabilities to triplets, calling them quadruples. They propose algorithms for answering queries consisting of one

quadruple with one variable at most. Huang el al., [99] propose algorithms for query processing over probabilistic RDF graphs with edge uncertainty only. Lian et al. [100] propose efficient algorithms for querying probabilistic RDF graphs with node attribute correlations. All these works do not support all the three types of uncertainty together, and do not support identity uncertainty at all.

Recently, there have been some studies that deal with identity uncertainty in querying data. Ioannou et al. [101] propose query evaluation algorithms for uncertain data with identity uncertainty. Hua et al. [102] propose a method for evaluating aggregate queries over data with identity uncertainty. Both of these approaches are not designed to handle graph data.

Chapter 3

Declarative Graph Data Cleaning

3.1 Introduction

In this chapter, we present our proposed approach for declarative graph data cleaning based on Datalog. We propose a unifying framework to specify different data cleaning tasks, and implement efficient algorithms to execute them.

Outline: We describe our specification language (Section 3.2). In the next sections, we present our declarative framework for specifying network analysis tasks (Section 3.3), and discuss our system implementation (Section 3.4) along with our techniques developed for incremental maintenance (Section 3.5). We then present the results of an experimental study (Section 3.6).

3.2 Specification Language and Data Model

Our specification language for defining inference tasks builds upon Datalog. A Datalog program consists of a set of rules and a set of facts. Facts represent statements that are true, whereas rules allow us to deduce new facts from other true facts that are already known (or deduced), and exist in the knowledge base. A Datalog rule has the following syntax:

$$\mathtt{L}_0 \text{ :- } \mathtt{L}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{L}_n$$

where each of $\texttt{L}_i$ is a *literal* of the form $P_i(X_1, \ldots, X_n)$, or $\sim P_i(X_1, \ldots, X_n)$, where P_i is a predicate symbol, and $X_1, \ldots, X_n$ are terms. For the purposes of our GRDB specification, we consider only definite clauses, in which there are no negations. Also, in some places, we use the shorthand $P(\overline{X})$ where $\overline{X}$ stands for $X_1, \ldots, X_n$. Terms can be variable terms or constant terms. Informally, rules are read as 'if $\texttt{L}_1$, ..., $\texttt{L}_n$ are true, then $\texttt{L}_0$ is true.' $\texttt{L}_0$ is called the rule's LHS or *head*, and $\texttt{L}_1$, ..., $\texttt{L}_n$ are called the rule's RHS, or *body*. Each $\texttt{L}_i$ on the rule's RHS is called a subgoal. A fact is a rule with an empty body and is always true. A fact that has all its terms constant is called a ground fact. In database terminology, each predicate symbol corresponds to a relation name. An *extensional database (EDB)* is the set of relation names corresponding to ground facts. An *intensional database (IDB)* is the set of relation names corresponding to inferred facts. Our graph is stored in an EDB, while rules defining various inference tasks are expressed as IDBs.

We use Datalog as the base language for our graph analysis framework for several reasons.

- Datalog can naturally capture both graph structure and properties of graph elements (i.e., nodes and edges). For example, one may query two-hop neighbors of node `X` using the rule `TwoHops(X,Z):-Edge(X,Y),Edge(Y,Z).`

- Datalog also enables querying the attributes of the queried elements. For example, one may specify the list of nodes with first name "John" using the rule: `John(X):-Node(X,'John', Y)`, where each node has an ID (corresponding to the X variable), a first name ('John' in this case), and a last name (corresponding

to the Y variable).

- Compared to SQL, Datalog is a natural language to answer path-based graph queries because it is a recursive language.

- On the other hand, compared to XPath, Datalog deals with graph edges as first-class citizens, where they can have identifiers and attributes that can be queried. In XPath, an edge is just expressed by the "/" (slash) operator.

- Compared to RDF query languages like SPARQL, Datalog can be naturally extended to handle concepts like feature domains (Section 3.3) and updates.

- Finally, compared to imperative languages, a declarative language like Datalog relieves the user from the burden of specifying how to evaluate the query by pushing this work to the evaluation engine. Furthermore, its algebraic properties allow the system to incrementally compute the changes in query results when the base graph changes, while in imperative languages, it is not as clear how to track dependencies and perform incremental maintenance.

Our data model supports multiple node and edge types where each type has its own set of attributes. Although our approach and framework can be applied to any EDB schema representing a graph structure, for brevity, we will assume just two EDBs, $Node(X, \overline{A})$ and $Edge(X, Y, \overline{B})$, where the $Node$ relation contains a key, X, along with a set of attributes $\overline{A}$ and the $Edge$ relation contains a key (X, Y) along with edge attributes $\overline{B}$. We have the following shorthand: `Node(X)` stands for `Node(X,_,...)`and means that node `X` exists in the EDB. Similarly, `Edge(X, Y)` means

that node `X` points to the node `Y`. `Node(X,Att=V)` stands for `Node(X,_,..,V,..)` and means that node `X` has the value `V` for attribute `Att`, and similarly, `Edge(X, Y, Att=V)` means that edge `(X, Y)` has the value `V` for attribute `Att`.

We extend Datalog with several constructs to enable our analysis framework. Some bear close similarity to existing Datalog extensions (e.g. aggregation), whereas others are more novel.

- **Aggregates:** An aggregate is a term of the form $Agg\langle\overline{Y}\rangle$ where Agg is an aggregation function, and $\overline{Y}$ are the aggregate function arguments. For a rule: $P(\overline{X}, Agg\langle\overline{Y}\rangle) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n})$, where $\overline{X}, \overline{Y} \subseteq \bigcup_i \overline{X_i}$, a set is created for each value of $\overline{X}$, the aggregate operation Agg is applied on each set, and a corresponding fact is added. In essence, this corresponds to the SQL group by operation. In network analysis, aggregates form a central component that enables feature definition to collect graph-wide, or element-based statistics.

- **Update Rules:** We use update rules to express graph updates that result from inference operations. Since updates have side effects, the order in which these side effects should take place must be specified explicitly in the program. Hence, our programs are divided into two parts: (1) the non-update rules (i.e.,query rules) where evaluation order does not matter, and (2) the update rules where it matters. We use following syntax to express updates:

  `[INSERT | DELETE | UPDATE]` $P(\overline{X}) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n})$

  where the predicate name P corresponds to an EDB, where the changes will take place. The semantics are that the rule is evaluated and the results are

then added or deleted from P's EDB for `INSERT` or `DELETE`, and updated (based on their keys) for `UPDATE`.

- `ITERATE` **Construct:** We introduce the `ITERATE` construct as a looping construct to allow updates to be performed iteratively:

  ```
  ITERATE(N) {   Block of Update Rules  }
  ```

  where N is either the number of iterations or $*$. The semantics of the `ITERATE` construct are that it applies the update rules in its body in the specified ordering iteratively, and recomputes (or maintains) the results of any "query" rules, until no change takes place or for at most N iterations, whichever happens first. If $*$ is specified, then the evaluation proceeds indefinitely until no changes take place.

- **Other extensions:** There are other Datalog extensions specific to our framework, like `DOMAIN` **constructs**, and **top K ranking**. We will discuss these extensions as we encounter them.

3.3 Declarative Analysis Framework

While the three network cleaning operations described in Section 2.1 result in different updates to the network, and can be combined in complex ways, network analysis processes can be seen, at a high level, as interleaved application of three basic modules as shown in Figure 3.1(i). In this section, we describe these components in detail, and then present our proposed declarative language constructs for specifying these components.

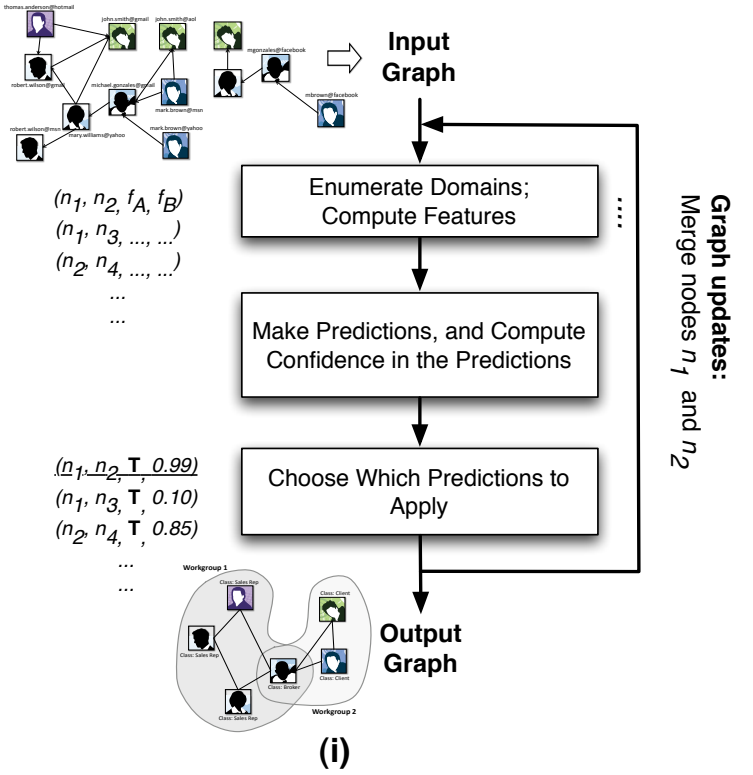
3.3.1 Defining Prediction Domains and Features

In order to define our graph inference tasks, we need to specify the prediction elements, features, and domains. We refer to the relevant objects as *prediction elements*. These are either nodes (in the case of attribute prediction), or pairs of nodes (in the case of link prediction and entity resolution). For our graph inferences, we typically need to compute the values of various "features" of the relevant objects. Depending on the nature of the features, this step is typically the most computationally expensive step in the overall inference process. *Prediction domains* are used to constrain the set of prediction elements. Judicious selection of prediction domains is essential to enable scaling.

Features: We can divide the features broadly into three categories based on their complexity:

Local: Attributes of the prediction elements themselves can be used as features for input to a prediction function. For nodes, these are node attributes; for pairs of nodes, these can be binary features which describe whether the attribute values of the nodes match, or real-valued features which measure the distance between the nodes' attribute values. The key distinguishing characteristic of these features is that they require only local information about the attribute values of the nodes, or pairs of nodes.

Local Structural: These are features that require exploration of a small fixed neighborhood around the prediction element. For node predictions, commonly used features include the **degree** of the node, the count of the number of neighbors




```
DOMAIN ER(X, Y) :- Node(X, Name=V1), Node(Y, Name=V2), dist(V1,V2) < α
{                                                                    (A)
   IntersectionCount(X, Y, COUNT<Z>) :- ...
   Adamic1(X, Y, Z, COUNT<N>) :- ...
   Adamic(X, Y, SUM<1 / log(N)>) :- ...
   Features-ER(X, Y, F1, F2) :- IntersectionCount(X, Y, F1), Adamic(X, Y, F2)
}
DOMAIN LP(X, Y) :-  ..... { ... }
```

```
DEFINE Merge(X, Y)                                                   (B)
{
   INSERT Edge(X, Z) :-  Edge(Y, Z)
   DELETE Edge(Y, Z)
   UPDATE Node(X, A=ANew) :- Node(X,A=AX), Node(Y,A=AY), ANew=(AX+AY)/2
   UPDATE Node(X, B=BNew) :- Node(X,B=BX), Node(X,B=BX), BNew=max(BX,BY)
   DELETE Node(Y)
}
```

```
ITERATE(n)                                                           (C)
{
   Merge(X, Y) :- Features-ER(X, Y, F1, F2, F3, ...), predict-ER(F1, F2, F3, ..) = true,
            confidence-ER(F1, F2, F3, ..) > 0.95
   INSERT Edge(X, Y) :- Features-LP(X, Y, G1, G2, G3, ...), predict-LP(G1, G2, G3, ..) = true,
            confidence-LP(G1, G2, G3, ..) IN TOP 5
}
```

(ii)

Figure 3.1: (i) Illustrative workflow depicting the main steps in an iterative statistical inference task (using the example of *entity resolution*); (ii) An example Datalog program that specifies an interleaved execution of an ER task and an LP task.

within c hops with a specific property (where c is a constant), etc. Another commonly used feature is the **clustering coefficient**. The clustering coefficient $C(x)$ of a node x in a graph is a measure of how close the node and its neighbors are to forming a clique; more precisely, it is the ratio of edges observed over the number of possible edges. In addition, for pairs of nodes, we often measure some sort of neighborhood similarity. Common neighborhood similarity measures include: **No. of common neighbors**; **Jaccard coefficient**, which is the No. of common neighbors normalized by the size of the union of the neighbors of the nodes; and, a related measure, introduced by Adamic and Adar [23], which gives more weight to rare features (those that are not shared by many other entities).

Global Structural: Examples of features that depend on the global structure of the graph include the **Katz score**, **betweenness centrality**, and **PageRank**. The Katz score for a pair of nodes is computed based on the number of different paths between the two nodes, with the shorter paths given higher weight than the longer paths. See Table 3.1 for the formal definition. The betweenness centrality of a node is determined by the number of shortest paths that contain that node; it is the frequency with which a node appears along the shortest path between other pairs. PageRank captures the probability that a random walk from one node will end up at the other node.

The features can be specified using Datalog in a straightforward manner. In Table 3.1, we show how the features discussed above can be expressed concisely and declaratively using this language. For example, to count the number of common

neighbors for nodes `X` and `Y`, we simply find the set of nodes `Z` such that `Edge(X, Z)` and `Edge(Y, Z)` are true, and we then count them using an aggregate rule. More complex features often require multiple rules (e.g., the Jaccard coefficient for two nodes is computed by finding the sizes of the intersection and the union of the neighborhoods of the two nodes separately). Local structural features can be computed without using any recursive evaluation. However, computation of global structural features requires use of recursion. For example, to compute the Katz score for a pair of nodes, we need to enumerate all paths between the two nodes.

Domains: While features may be defined for all prediction elements, often we want to restrict our attention to only a subset of the elements to make analysis tractable. We refer to such a subset of elements as the *prediction domain*. Prediction domain constructs are used to enumerate the elements for which predictions are made and feature values need to be computed. For attribute prediction, the prediction is over attribute values of the nodes and we can use the domain construct to restrict our attention to some subset of the nodes. This allows us, for example, to predict attribute values only for nodes with missing attribute values, or to predict attribute values only for nodes which have some percentage of neighboring values observed (not missing). Judicious use of prediction domains is especially important for tasks such as link prediction and entity resolution, where the prediction takes place over pairs of nodes. For a reasonably-sized network, it is infeasible to check every possible prediction element, and we must be able to limit the possible node pairs that are considered.

Degree: $Degree(x) = \|\Gamma(x)\|$	`Degree(X, COUNT<Y>) :- Edge(X, Y)`
No. of neighbors w/ Att = 'A'	`NumNeighbors(X, COUNT<Y>) :- Edge(X, Y), Node(Y, Att='A')`

Clustering Coefficient	`NeighborCluster(X, COUNT<Y,Z>) :- Edge(X,Y), Edge(X,Z), Edge(Y,Z)`
	`Degree(X, COUNT<Y>) :- Edge(X, Y)`
	`ClusteringCoeff(X, C) :- NeighborCluster(X,N), Degree(X,D), C=2*N/D*(D-1)`

Number of common neighbors	`IntersectionCount(X, Y, COUNT<Z>) :- Edge(X, Z), Edge(Y, Z)`

Jaccard's coefficient $Jaccard(x,y) = \frac{\|\Gamma(x)\cap\Gamma(y)\|}{\|\Gamma(x)\cup\Gamma(y)\|}$	`Degree(X, COUNT<Z>) :- Edge(X, Z)`
	`IntersectionCount(X, Y, COUNT<Z>) :- Edge(X, Z), Edge(Y, Z)`
	`UnionCount(X, Y, D) :- Degree(X,D1), Degree(Y,D2), D=D1+D2-D3 IntersectionCount(X, Y, D3)`
	`Jaccard(X, Y, J) :- IntersectionCount(X, Y, N), UnionCount(X, Y, D), J=N/D`

Adamic measure $Adamic(x,y) = \sum_{z\in\Gamma(x)\cap\Gamma(y)} \frac{1}{\log\Gamma(z)}$	`Degree(X, COUNT<Z>) :- Edge(X, Z)`
	`Adamic1(X, Y, Z, N) :- Edge(X, Z), Edge(Y, Z), Degree(Z,N)`
	`Adamic(X, Y, SUM<1/log(N)>) :- Edge(X, Z), Edge(Y, Z), Adamic1(X, Y, Z, N)`

Similarity based on a func. $f(v1, v2)$	`Similarity(X, Y, S) :- Node(X, Att=V1), Node(Y, Att=V1), S=f(V1, V2)`

Katz measure $Katz(x,y) = \sum_{l=1}^{\infty} \beta^{-l}.\|paths(x,y)^{<l>}\|$	`Path(X, Y, 1) :- Edge(X, Y)`
	`Path(X, Y, L1) :- Edge(X, Z), Path(Z, Y, L), L1=L+1`
	`Path_count(X, Y, L, COUNT<1>) :- Path(X, Y, L)`
	`Katz1(X, Y, L, K) :- Path_count(X, Y, L, N), K=N * power(`β`, -L)`
	`Katz(X, Y, SUM<K>) :- Katz1(X, Y, L, K)`

Table 3.1: Common relational features and their Datalog representation. We use $\Gamma(x)$ to indicate the set of neighbors of node x.

We use the keyword `DOMAIN` for defining a domain for features. The general syntax for specifying a domain is:

```
DOMAIN D(X1, X2, ...)  :-   ...
    {
       ⟨ List of features to be computed ⟩
    }
```

For example, during entity resolution, we may want to restrict ourselves to pairs

of nodes that are sufficiently close to each other based on the *string similarity distance* between their names. This can be specified as shown in Figure 3.1(ii)(A). Although it may seem that this domain requires listing all the pairs of nodes and filtering them, our framework supports efficient methods for avoiding that. The last rule (with head `Features-ER`) combines all the features into a single predicate using which we can do inference. Note that although we have focused on unary or binary prediction domains thus far, our framework allows for using n-ary domains; this may be needed for situations where we want to make predictions for groups of three or more entities.

3.3.2 Iterative Inference and Updating

The next step in the analysis process is to perform the required inferences and updates. For each prediction element, the prediction is made by applying a *user supplied function* over the features computed in the previous step and returning a *prediction* and a *confidence* (or *score*) value. This function can either be a user defined function or a function that is the output of some machine learning system; in the context of GrDB, we treat it as a black box. A key observation we make here is that, at this point, the *prediction can be done independently for each domain element in parallel*.

For attribute prediction, commonly used prediction functions include classifiers like naïve Bayes, logistic regression, and decision trees. Similarly, for link prediction, the problem of deciding whether to add an edge between a pair of nodes is often

treated as a *binary* classification problem, and the functions listed above can be used as well. In some cases, especially for entity resolution, a similarity function might be used instead to compute a similarity score for a pair of nodes, and then a thresholding mechanism may be used to decide which nodes to merge or which edges to add.

The next step depends on the nature of the inference task. In some cases, we may just make one pass and commit all of the predictions made. In other cases, we may only choose to commit a subset of the predictions, and may want to iteratively recompute the features and perform inference on the updated graph. The updates include attribute value changes (for attribute prediction), edge insertions/deletions (for link prediction), and node merges (for entity resolution), and we must recompute the values of the features in response to these updates. Such iterative application often results in more accurate predictions and robust behavior. The most common approach to choosing which predictions to commit is to choose either the top k of the predictions (by score) or all predictions with confidence above a given threshold. In general, for each individual inference task, the user must specify:

- *Prediction* function to be used and the predicate containing the features. The prediction function is written as a user-defined function (UDF) **Predict**: $\overline{FT} \rightarrow P$, where $\overline{FT}$ is the feature vector and P is the set of possible predictions.

- *Confidence* or *score* function to be used to choose a subset of the predictions to commit. This is also typically written as a UDF **Confidence**: $\overline{FT} \rightarrow [0, 1]$

(or more generally, **Score**: $\overline{FT} \rightarrow \Re$).

- *Prediction Confidence Cut-off:* In addition, the user must specify how to choose the subset of the predictions to be committed. A cut-off value for the confidence is provided by defining a predicate over the confidence function. A predicate can take the form of a minimum given threshold (e.g. `confidence(`$\overline{FT}$`)` $> C$), or can be expressed by picking the top K predictions. We define a Datalog extension for this purpose (`confidence(`$\overline{FT}$`) IN TOP K`).

- *Graph Update Operations* to be performed as a result of the inference. These are expressed as Datalog update rules.

- *Number of Iterations* used when updates are executed iteratively so that only high confidence predictions are applied in each iteration. As we described earlier, update rules are enclosed in an `ITERATE` block to achieve this control.

As an example, an entity resolution task where we only commit high-confidence predictions can be specified as (Figure 3.1(ii)(C)):

```
Merge(X, Y) :- Features-ER(X, Y, F1, F2, F3, ...),
               predict-ER(F1, F2, F3, ..)  = true,
               confidence-ER(F1, F2, F3, ..)  > 0.95
```

Here `Features-ER` contains all the features that are needed for inference. `predict-ER` and `confidence-ER` are the prediction and confidence functions respectively. To differentiate between functions and predicates in our Datalog programs, we use upper case initials for predicates and lower case initials for functions. `Merge(X, Y)` indicates that the graph update operation to be performed is a merge (corresponding to

entity resolution or duplicate elimination). Other examples include `INSERT Edge(X, Y)`, indicating edge addition between nodes `X` and `Y` (for link prediction, see Figure 3.1(ii)(C)), and `UPDATE Node(X,Att=V)`, indicating that the attribute value of `Att` should be changed to `V` for node `X`, (for classification or attribute prediction).

Note that update operations corresponding to link prediction and attribute prediction are simple (i.e.,a single rule). However, the `Merge` operation can be composite, i.e., defined in terms of other operations. An example of `Merge` definition is shown in Figure 3.1(ii)(B). This allows the user specify exactly how to update the attribute values for the new node that is created. The semantics of composite updates is that the update rules inside them are executed in order; however, there is no need to recompute the features after each single update rule. Features are recomputed only after the entire composite block is executed.

Finally, the user may specify an interleaving of two or more different inference tasks. For example, the syntax for specifying an interleaving of entity resolution and link prediction is as shown in Figure 3.1(ii)(C). Here for the second inference task, we specify that only the `TOP 5` of the predictions (based on the `confidence-LP` function results) be committed at end of each iteration. Note that `DOMAIN` blocks do not necessarily map to prediction tasks in the program according to a one-to-one relationship. The case in the above program is just for illustration. Two prediction tasks can use a set of features that have the same domain, and accordingly, all the features can be defined inside the same `DOMAIN` block. Furthermore, `DOMAIN` blocks may be defined for features that are not part of any prediction, but used as an input for computing other features in the program (e.g., to compute Jaccard which

requires a domain of pairs of nodes, we define degree using a domain of nodes).

3.4 Implementation

To implement our framework, we built a deductive database system on top of the Java Edition of the Berkeley DB key/value store. There are a number of reasons we built our own system rather than using an existing deductive or relational DBMS. Our initial system prototype was written on top of the H2 relational database system. In doing that implementation, we realized that this approach would not provide us with the fine-grained control over execution policies and, more importantly, storage policies, that we need to scale to large information networks with peculiar access and storage patterns. We investigated the use an existing deductive database system, but lack of an efficient open-source implementation that could be readily modified to implement our extensions hampered our efforts. We decided that implementing our own system, built using the solid indexing and storage foundations provided by Berkeley DB key/value store, would be the most flexible option for investigating new approaches for managing large-scale information networks.

Our implementation of a graph data analysis system involved two key components. First, we implemented a full fledged non-transactional relational database system that has *a query parser*, *a rule-based query optimizer*, *a relational expression converter* for converting Datalog rules to canonical relational expressions, and *a plan executor*; the plan executor contains various database operators like *scan*, *index lookup*, *group by*, different types of *join operators*, *view materialization and mainte-*

nance routine, and *a top-k operator*. We omit further details of this component due to space constraints.

Second, we implemented the necessary special logic to enable our framework, like incremental maintenance of various types of views, the `DOMAIN`, and the `ITERATE` constructs. Next, we discuss some of these components in detail.

3.4.1 Rule-Based Query Optimizer

In our system, we use a simple rule-based query optimizer that converts the relational expression, corresponding to each rule, to an execution plan. When handling joins, merge-joins are used whenever possible, then hash-join if the left and right tables (maps) fit in memory, and then index-based joins, and nested loop joins otherwise. We are currently developing a more sophisticated query optimizer that takes into account the special structure of the information networks and the Datalog rules to construct query plans.

3.4.2 The Merge-Join Operator

The implementation of a merge-join operator is crucial to our implementation for three reasons:

- Berkeley DB implements its key/value store as sorted maps where the sequential access of the map efficiently yields the tuples in their key order. Utilizing this property by using merge-joins can boost the performance, with little added query optimization cost.

- Many of the rules we saw in the previous sections are expressed in terms of other simpler rules, where each rule calculates a portion of a feature value, which are combined to yield a final value. For example, Jaccard coefficient is computed by first calculating the values in its numerator and denominator and dividing them. When evaluating the Jaccard rule, it is much faster to go over the numerator list and denominator list in one pass together and divide them, instead of joining them with nested loops or other methods.

- The same rationale applies to rules that combine multiple features for a given task in one view; instead of doing a multiway join for all the features, a single pass over them using merge join can give the desired results quickly.

To achieve maximum utilization of the merge-join operator, we introduce the notion of primary keys for IDBs, where rules defining IDBs define their keys as well, and where the user can precede a head variable by the # symbol to indicate it is part of the key. When the results are materialized, they can be accessed using their key.

3.4.3 Top-K Ranking by Confidence

Whenever ranking is required for the predictions, the view involving confidence computation is materialized along with the confidence score values obtained from the confidence function, sorted using the sorting operator, and the top K (or K percent) tuples are selected.

3.4.4 INSERT, DELETE and UPDATE Rules

As we mentioned earlier, Datalog does not have constructs for updating the base relations. However, since we need to change the graph to reflect the predictions made based on the features, we need a way to specify these changes in our language. Evaluation of `INSERT`, `DELETE` and `UPDATE` rules is identical to the evaluation of regular rules, where the right hand side of the rule is evaluated normally, but with the addition of applying the appropriate update to the base tables.

3.4.5 Iterative Inference

As in fixed-point Datalog semantics, where two approaches exist for evaluating programs: naive and semi-naive, in our `ITERATE`-based operational semantics, we support two evaluation techniques: naive and incremental. In naive, the views are recomputed in each iteration from scratch, regardless of the changes that happened to the base tables. In incremental evaluation, we take advantage of the small changes that take place in each iteration and maintain views incrementally, where they are updated by a small delta that is computed from the changes to the base tables. We discuss our incremental maintenance approach in more detail in Section 3.5.

3.5 Incremental Maintenance

In our implementation, we materialize the result of every Datalog rule in the system, and we treat these results as materialized views over the base relations. As the base relations change in response to the predictions made during analysis, we

need to maintain these views. View maintenance has been an active area of research in database systems for a long time. Some papers discuss its algebraic derivations (e.g. [103]), while others discuss efficient maintenance [104, 105]. Noisy graph analysis requires defining feature rules (which contain aggregate constructs), and `DOMAIN` rules, and we have developed a framework for maintaining views corresponding to these rules. Our contributions for this part of the system are:

- Previous work on deductive database aggregate queries (e.g., [40]) focuses more on mathematical properties of the aggregate programs and less on efficient implementation. On the other hand, relational (as opposed to deductive) systems have a rich literature on efficient implementation of incremental maintenance of materialized views. We extend the approach of Gupta et al. [104] to our deductive framework, for efficient materialized view maintenance without extra overhead over relational systems.

- We define an approach for efficiently maintaining `DOMAIN`-based views, which require left-outer joins. Our technique utilizes the fact that the two input tables are joined on their key attributes. Previous approaches for outer-join view maintenance like [105] are over-complicated for our requirements, or have been shown to be incorrect ([104]) (as discussed in [105]).

- We support cascaded view maintenance to handle the case where an incrementally maintained (aggregate) view is used to define another aggregate, non-aggregate, or `DOMAIN` view.

We note that, although our system can handle recursive feature rules, we currently do not maintain the corresponding views incrementally. We plan to support that in future work.

3.5.1 Feature Definition Views

Feature rules may result in aggregate or non-aggregate views. As described in Appendix 3.4, non-recursive, non-aggregate rules correspond to SPJ views in relational systems. We employ the change table technique [104] for incremental maintenance of SPJ and aggregate views. In the change table approach, changes to a view are derived and grouped together in one table. This approach is more efficient than tuple based approaches, which identify tuples to be deleted from the old view, and then identify new tuples to be added. Rather, when using the change table approach, a `REFRESH` operator is employed to update the old version of the table given the change table. For details of the `REFRESH` operator in case of SPJ and aggregate queries, the reader is referred to [104].

To keep track of insertions and deletions on base relations, we create a change table for base relations. The change table contains the changed tuples, appended by one more field, which we refer to as the *count field*; this field is set to -1 in case of deletion from the base table, and to 1 in case of insertion. We use the notation $P(\overline{X})[C]$ to express a predicate P that corresponds to a table P with fields $\overline{X}$ and the count field C.

To derive the change table for an SPJ view defined as:

$$P(\overline{X}) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n}),$$

we apply the following set of rules:

$$\Delta P(\overline{X})[C] :- \Delta P_1(\overline{X_1})[C_1], P_2(\overline{X_2})[C_2], \ldots, P_n(\overline{X_n})[C_n], C = \prod C_i$$

$$\Delta P(\overline{X})[C] :- P_1^{\mu}(\overline{X_1})[C_1], \Delta P_2(\overline{X_2})[C_2], \ldots, P_n(\overline{X_n})[C_n], C = \prod C_i$$

$\ldots$

$$\Delta P(\overline{X})[C] :- P_1^{\mu}(\overline{X_1})[C_1], P_2^{\mu}(\overline{X_2})[C_2], \ldots, \Delta P_n(\overline{X_n})[C_n], C = \prod C_i$$

where P_i^{μ} is the version of the table P_i after the delta has been applied. For brevity, from now on, when the suffixes $[C]$ and $[C_i]$ do not appear, it means that their relationship is $C = \prod C_i$.

Therefore, in our incremental maintenance framework, after the first iteration, we convert SPJ rules to the above set of rules and evaluate them using the same underlying query evaluation system.

For aggregate view maintenance, given the rule:

$$P(\overline{X}, Agg\langle Y\rangle) :- P_1(\overline{X_1}), \ldots, P_n(\overline{X_n}),$$

we find the change table using the set of rules

$$\acute{P}(\overline{X}, Y) :- \Delta P_1(\overline{X_1}), P_2(\overline{X_2}), \ldots, P_n(\overline{X_n})$$

$$\acute{P}(\overline{X}, Y) :- P_1^{\mu}(\overline{X_1}), \Delta P_2(\overline{X_2}), \ldots, P_n(\overline{X_n})$$

$\ldots$

$$\acute{P}(\overline{X}, Y) :- P_1^{\mu}(\overline{X_1}), P_2^{\mu}(\overline{X_2}), \ldots, \Delta P_n(\overline{X_n})$$

$$\hat{P}(\overline{X}, F(Y))[1] :- \acute{P}(\overline{X}, Y)[1] \qquad (1)$$

$$\hat{P}(\overline{X}, G(Y))[-1] :- \acute{P}(\overline{X}, Y)[-1] \qquad (2)$$

$$\Delta P(\overline{X}, \acute{Agg}(V))[SUM\langle C\rangle] :- \hat{P}(\overline{X}, V)[C]$$

Agg	$\acute{Agg}$	$F(Y)$	$G(Y)$
COUNT	SUM	1	-1
SUM	SUM	Y	$-Y$

Table 3.2: Corresponding values of $\acute{Agg}, F, G$ for COUNT and SUM aggregate functions

where $\acute{P}$ and $\hat{P}$ are intermediate tables that are used for computing ΔP, and $\acute{Agg}, F, G$ are functions that depend on the original aggregate function Agg. In Table 3.2, we show their values for the aggregates COUNT and SUM. Note according to the values in the table, rules (1) and (2) can be expressed in one rule and be efficiently evaluated using the following rules for the COUNT and SUM aggregates resp.

$\hat{P}(\overline{X}, C)[C] :- \acute{P}(\overline{X}, Y)[C]$ For COUNT

$\hat{P}(\overline{X}, Y \times C)[C] :- \acute{P}(\overline{X}, Y)[C]$ For SUM

Again we refer the reader to [104] for the details of the REFRESH operator that updates old versions of aggregate rule evaluation using the change table derived above. In our implementation, we implement the REFRESH operation using the merge-join operator, which makes the REFRESH operation linear in the size of both the change table and the old version of the view.

3.5.2 DOMAIN Views

In GRDB, we expect the use of DOMAINs to be common for two reasons. First, it enables us to restrict the number of objects (nodes, edges, or pairs of nodes) for which features are computed. Second, it is required to keep track of the objects whose values are required. For instance, we may use the number of neighbors as a feature in some task. In a naive evaluation, the nodes without neighbors will be

dropped; we must carefully keep track of such nodes.

A `DOMAIN` rule of the form:

$$\texttt{DOMAIN}\ L(\overline{X}) \texttt{ :- Subgoals of } L\ \{$$

$$P_1(\overline{X},\overline{Y}) \texttt{ :- Subgoals of } P_1$$

$$P_2(\overline{X},\overline{Y}) \texttt{ :- Subgoals of } P_2$$

$$\}$$

is translated to the following set of rules in our system:

$$L(\overline{X}) \texttt{ : -Subgoals of } L$$

$$\acute{P}_1(\overline{X},\overline{Y}) \texttt{ :- } L, \texttt{Subgoals of } P_1$$

$$\acute{P}_2(\overline{X},\overline{Y}) \texttt{ :- } L, \texttt{Subgoals of } P_2$$

$$P_1(\overline{X},\overline{Y}) \texttt{ :- } L(\overline{X}) >> \acute{P}_1(\overline{X},\overline{Y})$$

$$P_2(\overline{X},\overline{Y}) \texttt{ :- } L(\overline{X}) >> \acute{P}_2(\overline{X},\overline{Y})$$

where $>>$, a new Datalog operator that we define, denotes left outer-join, which associates non-appearing keys in the right relation with zeros instead of nulls[1]. Therefore, to maintain a `DOMAIN` view, we need to maintain all the rules in the rewritten version. Among them are outer-join rules. To maintain the outer-joins incrementally, we make use of the fact that the left and right relations are joined on their primary keys.

Assume we are maintaining the following rule:

$$P(\overline{X},\overline{Y}) \texttt{ :- } L(\overline{X}) >> R(\overline{X},\overline{Y})$$

where $\overline{X}$ is the primary key for both the left (outer) table L, and the right (inner)

[1]Generally speaking, this operator associates a default value that depends on the attribute sought; however, we use zeros for clarity.

table R. We use the symbol $\overline{\mathbb{Z}}$ to denote the set of zeros our left-outer join operator produces in place of nulls.

Deriving the change when the left relation L changes: There are four possibilities to consider depending on whether a tuple is being inserted or deleted, and whether a corresponding tuple (with the same key) is present in R. Say a tuple $(\overline{X})$ is deleted from L. If the corresponding tuple $(\overline{X}, \overline{Y})$ exists in R, then, the tuple $(\overline{X}, \overline{Y})$ is deleted from the view; otherwise, $(\overline{X}, \overline{\mathbb{Z}})$ is deleted. On the other hand, say we are inserting a tuple $(\overline{X})$ into L. If the corresponding tuple $(\overline{X}, \overline{Y})$ exists in R, then, the tuple $(\overline{X}, \overline{Y})$ is inserted into the view; otherwise, $(\overline{X}, \overline{\mathbb{Z}})$ is inserted.

Based on this discussion, the change table for an outer-join view when the outer relation changes can be derived as follows:

$$\Delta P(\overline{X}, \overline{Y}) :- \Delta L(\overline{X}) >> R(\overline{X}, \overline{Y})$$

Deriving the change when the right relation R changes: If only updates on $\overline{Y}$ take place on the right relation R, then the view will be updated in the same way updates took place on R. However, maintaining outer-join views becomes trickier when inserting or deleting tuples from the right relation. In this case, the following holds:

- If a tuple $(\overline{X}, \overline{Y})$ is inserted into R that did not exist before, then the tuple $(\overline{X}, \overline{\mathbb{Z}})$ that existed in the view should be removed and replaced with $(\overline{X}, \overline{Y})$.
- If a tuple $(\overline{X}, \overline{Y})$ is deleted from R, then the tuple $(\overline{X}, \overline{Y})$ that existed in the view should be removed and replaced with $(\overline{X}, \overline{\mathbb{Z}})$.

We use the following algorithm to adjust the change table ΔR so that normal SPJ

change table derivation can be applied to the case when an inner table of a left-outer join is changed.

```
1 for each key X̄ in ΔR do
2     S = Sum of the count field values of all tuples with key X̄;
3     if (S < 0) then
4         Adjust R by inserting (X̄, Z̄) to it with a count value of 1;
5     if (S > 0) then
6         Adjust R by inserting (X̄, Z̄) to it with a count value of −1;
```

We keep track of these changes so that they are rolled back after computing the outer-join view change table. Thus, the rule for maintaining left outer-join views when the inner relation changes is:

$$\Delta P(\overline{X}, \overline{Y}) :- L(\overline{X}), \Delta R^{adjusted}(\overline{X}, \overline{Y})$$

Deriving the change when both the R and L relations change simultaneously: Based on the discussion above, the rules for deriving the change table for a left outer-join view defined as:

$$P(\overline{X}, \overline{Y}) :- L(\overline{X}) >> R(\overline{X}, \overline{Y})$$

can be found by:

$$\Delta P(\overline{X}, \overline{Y}) :- \Delta L(\overline{X}) >> R(\overline{X}, \overline{Y})$$

$$\Delta P(\overline{X}, \overline{Y}) :- L^{\mu}(\overline{X}), \Delta R^{adjusted}(\overline{X}, \overline{Y})$$

For refreshing the left-outer join view, we use the `REFRESH` operator defined in [104], with the same parameters as used in the SPJ case.

3.5.3 Cascaded View Maintenance

A common situation in our framework is that views are not only based on the base tables, but also based on other views that are themselves incrementally maintained (a situation we refer to by *cascaded view maintenance*). Handling this situation, however, requires a special procedure when using the output of an incrementally maintained aggregate view to incrementally maintain other views. By looking closely at the nature of the count field associated with aggregate maintenance, we find that it does not express a tuple-based count; rather it represents a group-based count, where it indicates how many times the aggregation group occurs in the relation. These two incompatible representations make the use of aggregate change tables infeasible in maintaining other views that expect tuple-level counts. To overcome this, while refreshing an aggregate view, we change the form of the aggregate change table to reflect tuple based changes, rather than group-based. This is performed by keeping track of the tuples actually deleted and inserted into the aggregate table as a result of the refresh. It is clear that switching between the group-based change table to the tuple-based change table does not impose additional overhead, because it is piggybacked on the `REFRESH` operator, and at the same time it enables use of aggregate change tables in future non-aggregate view maintenance.

3.6 Experimental Evaluation

In this section we discuss our experimental evaluation, where we report experimental results from three sets of experiments. First, we evaluate our framework

over a wide range of synthetically-generated graphs that emulate the topological properties and attributes of real networks. We also compare the performance of the recompute (RECOMP) approach with the incremental approach (INCR) under different conditions and experimental configurations. We then compare the performance of our feature construction sub-module with that of the well-established Apache Derby Java DBMS. Finally, we describe a real-world classification problem that we solved using GrDB and report the results. We run all of our experiments on servers with two 3.4 GHz dual-core CPUs, 8 GB of RAM at 400 MHz, and a 7200 RPM 80 GB hard drive. We present our results for all experiments in Figure 3.2.

3.6.1 Synthetic Data Experiments

To perform experiments on synthetic data, we develop a data generator that first creates a synthetic network with specific properties (e.g. size or edge density), and then adds noise to it to generate an input graph for the system. The goal of the network analysis is to try to reconstruct the original graph.

3.6.1.1 Synthetic Data Generator

Our synthetic data generator creates a noisy network with ambiguous references which need to be assigned to entities, missing labels which need to be classified, and missing and spurious edges whose existence must be predicted. We create the network topology using two widely used graph generation models, preferential attachment [53] and the forest fire [106], to create networks whose topological prop-

erties (e.g., degree distribution, diameter) match those commonly found in various real world networks. We use the forest fire model, with the parameters defined in [106], to generate the network topology used for a majority of our experiments. Because it is difficult to control link density directly in the forest fire model, for the set of experiments varying link density, we use the preferential attachment model where the model parameters directly control the average degree of the nodes. We then generate attributes for the three types of inferences we perform on the network. The first set is for use with attribute prediction and includes the labels and attributes where the node labels exhibit high autocorrelation (i.e., nodes which share an edge are likely to have the same label). The second set of attributes are binary features used for link prediction where nodes with a similar set of attributes are likely to share an edge. The last set of attributes are used for entity resolution and represent attributes that imply, non-uniquely, the entity it refers to. The final phase of the synthetic data generator adds noise to the topology and attributes of the network. We add noise by first creating copies of the original set of nodes where the copies initially have the same sets of attributes and "equivalent" edges as the original. Next, we randomly add noise to the attribute values of the nodes by randomly flipping the value of a random subset of the binary attributes. We also remove label values from all the nodes. Additionally, we add noise to the edges by randomly removing and adding edges between pairs of nodes.

Using this approach, we generate synthetic input networks of varying properties for our experiments. Each network is designed to mimic a communication network between online social network user accounts. The Nodes EDB has the

attributes (NodeID, AccountInfo, Country, Label, Committed). The AccountInfo attribute is an encoding of some information regarding the corresponding account. The Country attribute represents which country this user lists as her home. Label is the attribute to be predicted in attribute prediction experiments, where it can take one of two values, "A" and "B". We discuss Committed attribute in the next subsection. The Edges EDB has three attributes (SourceNodeID, DistNodeID, Type). The Type attribute denotes whether the relationship is a *communication* or *friendship* link.

We vary the network size by increasing both the number of nodes and edges, and the network density by holding the number of nodes constant and increasing the number of edges using the *preferential attachment model* [53].

3.6.1.2 Experiment Details

We evaluated our framework on attribute prediction, link prediction, and entity resolution. Our attribute prediction model is based on a variant of the iterative classification model (ICA) [15]. The ICA model requires a classifier to be run during each iteration. We use a logistic regression classifier; the features we used included a subset of the node attributes and the number of neighbors for each class label (the relational features). For link prediction, we also use a logistic regression classifier over pairs of nodes; the features here were the number of common neighbors and also the local attribute similarities, where the similarity function computes the percent of common attributes between the pair of nodes. In this experiment, we predict

friendship links using *communication* links. In this context, we set the *prediction domain* to be those pairs of nodes that have at least one communication link. For entity resolution, the relational feature we use in our experiments is the Jaccard's coefficient feature defined in Table 3.1 and define the domain as pairs of nodes that match on a subset of the attributes. A complete listing of the programs is as follows.

For attribute prediction, we predict the Label attribute for each user. For link prediction, we predict friendship links based on communication links. The task of entity resolution is to merge user accounts referring to the same user to a single node.

3.6.1.3 Attribute Prediction

The program used for attribute prediction is as follows (recall that we precede a variable by # symbol to indicate that this variable is part of the key).

```
      DOMAIN Uncommitted(#X):-Node(X,Committed='no') {
  ANeighbors(#X,Count<Y>):- Edge(X,Y),
                        Node(Y,Label='A')
  BNeighbors(#X,Count<Y>):- Edge(X,Y),
                        Node(Y,Label='B')
  Features-AP(#X,A,B,I):- ANeighbors(X,A),
                      BNeighbors(X,B),
                      Node(X,AccountInfo=I)
}
ITERATE(10) {
  UPDATE Node(X,_,_,P,'yes'):- Features-AP(X,A,B,I),
                          P = predict-AP(X,A,B,I),
                          confidence-AP(X,A,B,I)
                          IN TOP 1%
}
```

The program specifies that we need to compute the number of neighbors with A and B labels in each iteration, apply the top predictions, and mark those predic-

tions as *committed* (i.e., Committed='yes'). Since our domain is the *uncommitted* nodes (i.e., Committed='no'), we do not change these predictions in later iterations. Initially, before any iteration, all the nodes are marked as uncommitted.

3.6.1.4 Link Prediction

The program used for link prediction is as follows. The `Bin` IDB is used to define the domain for the link prediction elements, which are node pairs that have *communication* edges between them.

```
      DOMAIN Bin(#X,#Y) :- Edge(X,Y,'Communication') {
  IntersectionCount(#X,#Y,Count<Z>)):- Edge(X,Z),
                                  Edge(Y,Z),X!=Y
  Similarity(#X,#Y,S):-Node(X, AccountInfo=IX),
                  Node(Y, AccountInfo=IY),
                  S=sim(IX,IY)
  LabelSimilarity(#X,#Y,S):-Node(X, Label=LX),
                      Node(Y, Label=LY),
                      S=sim(LX,LY)
  Features-LP(#X,#Y,F1,F2,F3):-IntersectionCount(X,Y,F1),
                         Similarity(X,Y,F2),
                         LabelSimilarity(X,Y,F3)
}
ITERATE(10) {
  INSERT Edge(X,Y,'Friend'):-Features-LP(X,Y,F1,F2,F3),
                        predict-LP(F1,F2,F3)=true,
                        confidence-LP(F1,F2,F3)
                        IN TOP 1%
}
```

Here, `sim` is a string similarity function. The above program computes three features; the number of common neighbors between pairs of nodes, the similarity between their local info, and the similarity between their label. The `DOMAIN` is the pairs of nodes having a communication link between them.

3.6.1.5 Entity Resolution

The ER program is as follows:

```
 DOMAIN AllNodes(#X) :- Node(X) {
  Degree(#X,Count<Y>):-Edge(X,Y)
}
DOMAIN Bin(#X,#Y):- Node(X, Country=C1),
                Node(X, Country=C2), C1=C2 {
  IntersectionCount(#X,#Y,Count<Z>):- Edge(X,Z),
                                Edge(Y,Z),X!=Y
  UnionCount(#X,#Y,U):- Degree(X,DX), Degree(Y,DY),
                   IntersectionCount(X,Y,C),
                   U=DX+DY-C
  Jaccard(#X,#Y,J):- IntersectionCount(X,Y,C),
                 UnionCount(X,Y,U), J=C/U
}
ITERATE(10) {
  Merge(X,Y) :- Jaccard(X,Y,J), predict-ER(J)=true,
             confidence-ER(J) IN TOP 1%
}
DEFINE Merge(X, Y){
  INSERT Edge(X, Z, _) :- Edge(Y, Z, _)
  DELETE Edge(Y, Z, _)
  DELETE Node(Y, _)
}
```

The program calculates Jaccard coefficient of nodes having the same Country attribute, and applying the Merge operation to node pairs satisfying the user-defined function `predict-ER`.

3.6.1.6 Varying Network Properties

We begin by exploring the effectiveness of our system at applying attribute prediction, link prediction, and entity resolution as the input graph size varies. We increase the number of nodes in the noisy input graph from $27,014$ nodes with $247,366$ edges to $136,396$ nodes with $1,322,506$ edges (corresponding original net-

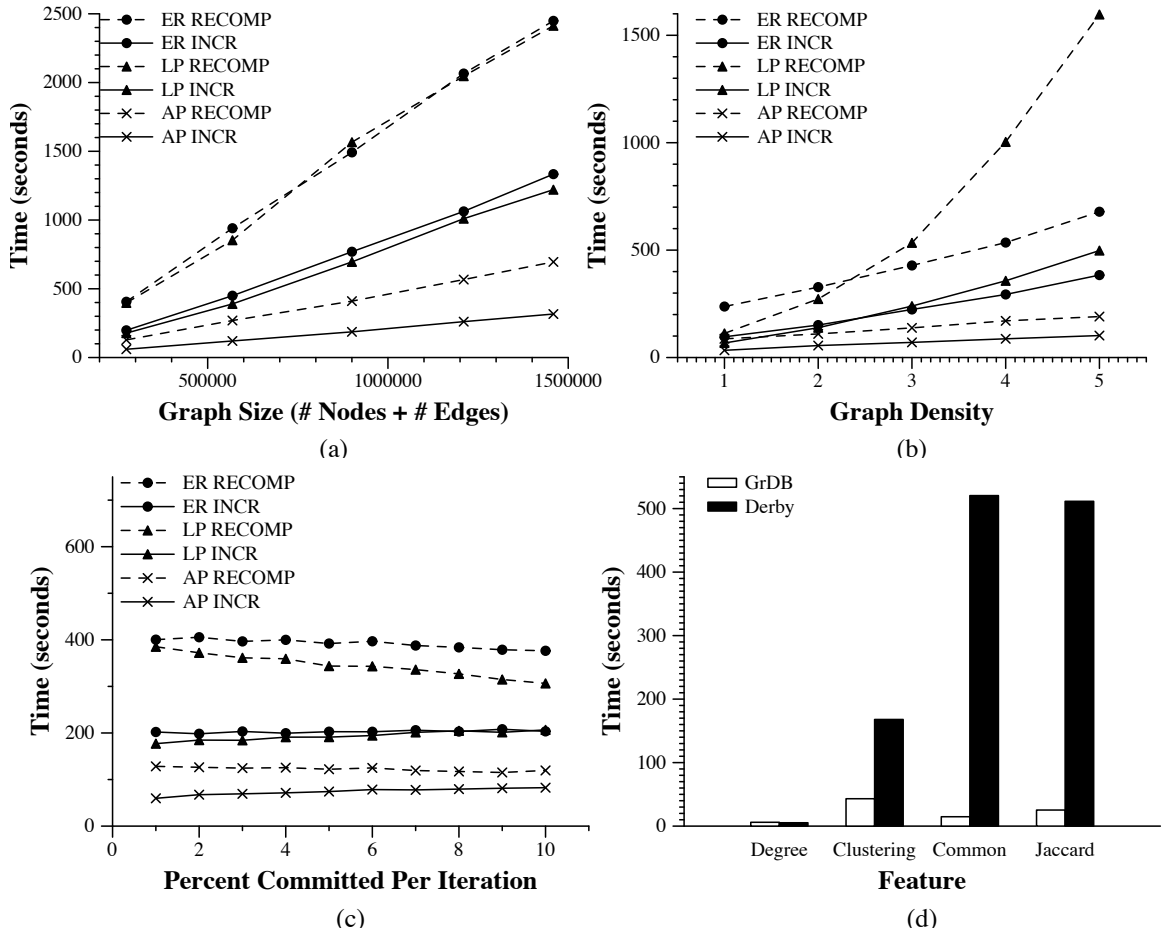


Figure 3.2: Runtime performance, in seconds, of attribute prediction (AP), link prediction (LP), and entity resolution (ER) using graphs of (a) varying sizes and (b) varying densities, and (c) by changing the percentage of predictions committed per iteration. (d) Comparison of feature construction time with Derby.

works contained 20,000 to 100,000 nodes). The results are shown in Figure 3.2(a). As expected, the run time increases with the network size. Further, we can see that INCR greatly improves over RECOMP, by a factor of 2 on average for all the tasks.

We next explore how link density affects performance. Increasing the link density increases the number of edges that need to be considered during feature and domain computation. With attribute prediction, for example, the increased density

results in a larger number of neighboring nodes whose labels must be counted. In our experiment, we added noise to a $20,000$ node network of various network densities. We show the results in Figure 3.2(b). Similar to the previous experiments with AP and ER, INCR is outperforming than RECOMP by a factor of 2 on average. However, in LP, the performance is greatly affected by increasing the density, especially in RECOMP, while in INCR the rate of increase is much less. This is a natural result, as LP is an edge-oriented problem, and hence is even more sensitive to the increase in the number of edges.

3.6.1.7 Varying Update Size

Next, we look at the effect of varying the number of predictions committed in each iteration. We use a graph with $27,014$ nodes and $247,366$ edges for this set of experiments. The results are shown in Figure 3.2(c). While INCR continues to outperform RECOMP, we observe that INCR's time increases as the percent committed is increased. This is to be expected; when the changes become large enough, the overhead of keeping track of the deltas and refreshing the old views will approach the overhead of recomputing everything. We also observe, although we may expect RECOMP to be independent of the percent committed at each iteration, its time actually decreases with the percent increase; this is because, in each iteration, fewer elements are considered for prediction, resulting in less overall running time.

3.6.2 Comparison with Derby

We also ran many micro-benchmarking experiments, focusing on different pieces of the system. We report the result for one such experiment: here we compare the performance of our feature construction sub-module with the performance of Apache Derby Java DBMS, by comparing the execution time of computing the following features over a noisy $20,000$ node graph: (a) the degrees, and (b) clustering coefficients for all nodes; (c) common neighbors between pairs of nodes, for all pairs that share the value of a common attribute; and (d) Jaccard coefficients for the same set of pairs.

For Derby, we express these features using SQL, and measure Derby's execution time. The comparison results are shown in Figure 3.2(d). While computation of a simple feature like degree is comparable in both systems, our implementation outperforms Derby by factors of 4, 35, and 20, respectively, for the other more computationally expensive features.

3.6.3 Real-world Experiment

We crawled the PubMed online dataset – a citation network in the medical domain. The size of the network is $50,634$ papers and it has $115,323$ citation edges. We used GrDB to infer the category (class label) of each paper. The dataset has four categories describing the topic of the paper (Cognition, Learning, Perception, and Thinking). We used 2-fold cross validation, where we trained a classifier using about half of the network, and tested it on the other half. We used logistic regression

for the prediction function, which we supply with the presence of words of the paper abstract (we use a bag of 1678 words) and the count of citations of each category. When committing the top 10% of the predictions and iterating for 10 iterations, INCR takes 28 minutes (average over the two folds) to finish, while RECOMP takes 42 minutes. Note that the increased overhead in these experiments over that of the synthetic data is due to the large number of local features (i.e., words of the abstract) which logistic regression has to reason about for each node. Using this approach leads to 84% accuracy on predicting the paper categories.

Chapter 4

Graph Data Querying and Analysis

4.1 Introduction

In the previous chapter, we proposed a unifying framework to perform declarative graph cleaning over noisy graphs. The input for such a framework is a noisy observed graph, and the output is an inferred graph that has the missing information derived, and duplicate nodes detected and merged. In this chapter, we present approaches for querying and analyzing such cleaned graphs. The contributions are two-fold. First, we define ego-centric pattern census queries, a new type of graph analysis query which searches for patterns in local node neighborhoods and reports their counts, and we show efficient techniques for evaluating ego-centric pattern census queries over large graphs. Second, we propose a scalable algorithm for evaluating subgraph pattern matching queries, as efficient evaluation of ego-centric pattern census queries relies on the existence of highly efficient evaluation methods for subgraph pattern matching. Before presenting both techniques, we discuss further motivating applications of the problem of ego-centric pattern census.

Targeted Marketing: Viral marketing is proving to be an effective tool for product advertisement. Selected consumers are given the product with the hope that they will like the product and recommend it to their friends. These consumers must be

chosen wisely to minimize the cost and maximize the benefits of advertising. Simple criteria such as picking consumers with the most friends, or consumers that are connected to many other consumers through short paths, are typically used. However the ability to identify richer structures is desirable in many cases. For example, a travel agency may wish to identify couples that have either the largest number of couples in their combined network, or the largest number of couple pairs, i.e., couples who are friends with couples. The latter structure is depicted in Figure 4.1(a).

Node Classification: In collective classification [11], a node's neighborhood is used to predict the node's own class label. For example, in a collaboration network, a scientist who collaborates mostly with scientists from a specific field (e.g., databases or software engineering) is likely to be from the same field. In a family relationship network (with "is married to" and "is parent of" relationships), for each child we may wish to count the number of relatives up to 3 hops away who are smokers (or obese), with their parents also being smokers (or obese). This could be a measure of the risk of being a smoker (or obese) for the child, and can be used for predicting that risk. The pattern of this query is depicted in Figure 4.1(b).

Structural Balance: In social balance theory, signed networks are networks that have positive and negative signs on their links, denoting whether a link expresses a positive tie (e.g., friendship) or a negative tie (e.g., foe) [107]. In signed networks, triangles with an odd number of negative links (one or three) are considered unstable. In such networks, we can measure the amount of instability in each node's ego network by counting the number of unstable triangles in its k-hop neighborhood.

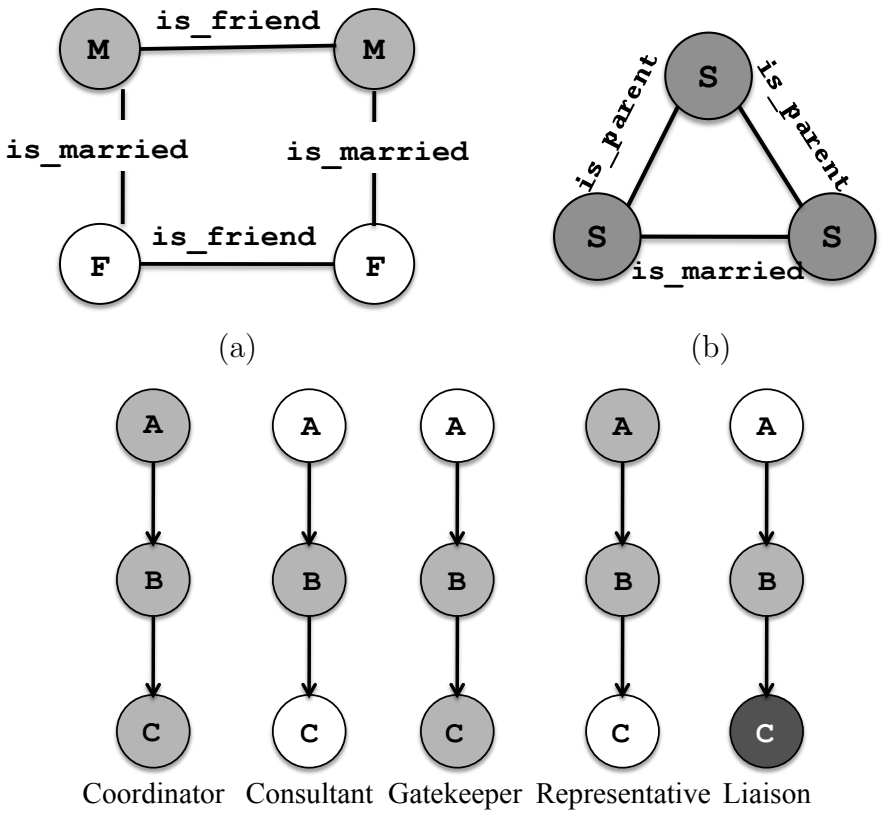


Figure 4.1: (a) Pattern that captures two couples that are friends with each other – such a pattern may be useful in a targeted marketing application; (b) Example pattern used in the node classification application; (c) Different brokerage patterns – the colors denote organizations, and the function of the broker (the middle node) depends on the organizations that the three nodes belong to (e.g., B is a coordinator if all three are in the same organization).

Brokerage Analysis: In organizational theory, management scientists are interested in the roles different individuals play, both within an organization, and across organizations. For example, in a transaction network, the middle node in a directed triad (e.g., node B in the triad $A \rightarrow B \rightarrow C$) is called a *coordinator* if all three are in the same organization, or a *gatekeeper* if A is in a different organization than B and C. Figure 4.1(c) shows the different brokerage types. A brokerage score of a

given type can then be computed for a node x by counting the number of patterns of that type that x participates in.

Graph Indexing: Finally, counts of specific structural patterns in every node's k-hop neighborhood in a large graph are regarded as *node signatures* and are often used for subgraph pattern matching to prune the search space [80]. Our algorithms can be used to efficiently build sophisticated signatures, that can be used when searching for large, complex subgraphs.

The problem of ego-centric pattern census combines elements from micro-level ego-centric network analysis, subgraph pattern matching, and motif counting. The goal of subgraph pattern matching is to find all matches of a given query graph in the database graph. While the general problem of pattern matching is NP-complete, there is much work on designing efficient algorithms and index data structures to answer those queries [9, 80, 81, 108]. Motif counting, on the other hand, is the problem of counting specified structural patterns of small sizes in the network [3, 4, 58, 5, 6, 7, 8]. Motif discovery typically does not take node or edge attributes into account, but rather depends solely on the pattern structure. It is commonly performed on naturally occurring graphs like biological networks [3], or computational graphs such as software graphs [8] or computer network graphs [109]. In these works, motif profiles (i.e., counts of different motif structures in the graph) have proven to be a strong indicator of a network's function. Local motif counting (i.e., counting motifs that a node is part of) has also been recently used as a tool to classify a node's role. For example, Kerrebroeck et al. [110] count the number of

loops a node is part of, and use it as a measure to quantify the node's importance in the network. Prüzlj [111] proposes the notion of *local graphlet degree distribution* as a means for network comparison.

Several ego-centric measures can be expressed as ego-centric pattern census queries with very simple input patterns, and can be seen as special cases of ego-centric pattern census. For example, an ego-centric measure like node degree corresponds to searching for a pattern of a *node* in a 1-hop neighborhood, and clustering coefficient (and its k-clustering coefficient version [112]) can be expressed in terms of counting *edge* patterns in 1-neighborhood (and k-neighborhood, respectively). Similarly, Jaccard coefficient of a pair of nodes can be computed by counting a *node* pattern in the nodes' 1-neighborhood intersection and union.

Outline: The rest of chapter is organized as follows. In Section 4.2 we present the data model and our language specification. In Section 4.3, we propose an efficient pattern matching algorithm that is used as part of the algorithms in Section 5.5 that we propose for evaluating pattern census queries. In Section 4.5 we discuss experimental performance evaluation through synthetic datasets and workloads. In addition, we use our language to design a link prediction experiment over DBLP and report its results.

4.2 Data Model and Language Specification

We begin with a brief discussion of the graph data model, and some basic definitions. We denote the database graph by $G = (V_G, E_G)$, where V_G and E_G

Table 4.1: Examples of patterns and pattern census queries

Row #	Pattern	Query
1	`PATTERN SINGLE_NODE {A;}`	`SELECT ID, COUNT(SINGLE_NODE, SUBGRAPH(ID, 2)) FROM NODES`
2	`PATTERN SINGLE_EDGE {A-B;}`	`SELECT N1.ID, N2.ID, COUNT(SINGLE_EDGE, SUBGRAPH-INTERSECTION(N1.ID, N2.ID, 1)) FROM NODES AS N1, NODES AS N2`
3	`PATTERN SQUARE { A-B; B-C; C-D; D-A; }`	`SELECT ID, COUNT(SQUARE, SUBGRAPH(ID, 2)) FROM NODES`
4	`PATTERN TRIAD { A->B; B->C; A!->C; [A.LABEL=B.LABEL]; [B.LABEL=C.LABEL]; SUBPATTERN COORDINATOR {B} }`	`SELECT ID, COUNTSP(COORDINATOR, TRIAD, SUBGRAPH(ID, 0)) FROM NODES`

denote the sets of nodes and edges respectively. The database graph may be directed or undirected, and both the nodes and the edges can have arbitrary sets of attributes (stored as *attribute-value* pairs). Similarly we denote the pattern graph by P. A pattern graph may be associated with a set of predicates on the underlying attributes.

Two graphs $P = (V_P, E_P)$ and $M = (V_M, E_M)$ are said to be isomorphic if there is a bijective mapping $\mu : V_P \rightarrow V_M$ such that $(v, v') \in E_P$ if and only if $(\mu(v), \mu(v')) \in E_M$. We say that a pattern graph P *matches* a database graph G if there is a subgraph M of G that is isomorphic to P under a mapping μ, and the predicate in P are satisfied under the same assignment μ. We say that M is a match of P in G.

Next, we introduce our language for specifying pattern census queries. Our pattern specification language is designed to be general and flexible. The language is based on SQL, but our algorithms actually operate on an disk-resident adjacency-list graph representation, and our system can be easily extended to support a visual

pattern specification language as well.

The pattern census SQL queries are written against a logical representation of the graph as two relations: `NODES(ID, NATTR1, ...)` and `EDGES(ID1, ID2, EATTR1, ...)`, where `ID` is the node identifier. Attribute references in queries are interpreted dynamically, and hence the list of attributes does not have to be pre-specified.

For a pattern census query, we need to be able to specify three things:

Search Neighborhoods: We need to specify the neighborhoods in which to do pattern census. We currently support specifying three types of search neighborhoods:

- `SUBGRAPH(N, k)`: This specifies an k-hop neighborhood around the node, i.e., the incident subgraph on the nodes that are reachable from N in k hops or less.
- `SUBGRAPH-INTERSECTION(N1, N2, k)`: Given two nodes N1 and N2 and a radius k, this specifies the intersection of the k-hop neighborhoods of N1 and N2.
- `SUBGRAPH-UNION(N1, N2, k)`: Similar to above except we take union instead of intersection.

Focal Nodes: We need to be able to specify for which nodes or for which pairs of nodes to conduct the pattern census. We use standard SQL constructs for this purpose, i.e., the user can specify predicates that should be satisfied by the nodes

or pairs of nodes. Predicates are given in the `WHERE` clause of the `SQL` statement.

Pattern: Our pattern specification language (see Table 4.1) allows the user to specify the nodes in the pattern, the connections (edges) between the nodes, and predicates on either the node attributes or the edge attributes. The structural pattern (nodes and edges) is specified using variables (e.g., A, B, C) that can be bound to any node in the graph. The user can also specify the direction of each edge if desired, and can specify that a particular edge should not exist. Table 4.1 (1-3) shows three simple patterns and SQL queries that count the number of patterns in different types of neighborhoods. Table 4.1 (4) shows a somewhat more complex directed pattern, that also specifies that a particular edge (from A to C) should not exist and requires all three nodes to have the same label (this pattern corresponds to a *coordinator* in brokerage analysis).

We also allow the user to specify one or more **subpatterns** in the pattern, where each subpattern is specified as a subset of the nodes in the pattern. This allows the user to precisely dictate the types of matches that should be counted. Consider the example shown in Table 4.1 (4). Here we specify a single subpattern containing the middle node in the triad, and the census is done in the 0-hop neighborhood around each node (which contains just that node). In other words, this query counts the number of triads in which B is the coordinator. It is not possible to do this type of census without the subpattern construct (if we simply count the number of triads in the 1-hop neighborhood around each node, we would also count the triads for which B is not a coordinator).

4.3 Subgraph Pattern Matching

A key component of both of our proposed query evaluation algorithms is a pattern matching algorithm that is used to find all matches for the given pattern in the graph. We adapt the algorithm proposed recently by He and Singh [9] (denoted GQL henceforth), by incorporating additional novel pruning steps that lead to orders-of-magnitude performance improvements over that prior work. Our algorithm consists of four steps: (1) enumerate candidate matches for each pattern node, (2) initialize candidate neighbor sets for each candidate node, (3) simultaneously prune candidate nodes and their candidate neighbors, and (4) extract pattern matches directly from the pruned set of candidates and the candidate neighbors. Although similar in spirit to GQL, our algorithm differs from it in subtle but significant ways. Our algorithm is centered around the idea of explicitly maintaining *candidate neighbors* with each candidate node. This not only results in more efficient pruning of the search space, but also results in orders of magnitude improvements in the final stage of extracting patterns. Our algorithm is also much simpler. In the description that follows, we assume both the pattern and database graphs have an explicit attribute called *label* drawn from a finite label space; the unlabeled case is equivalent to both the database and pattern graphs having the same label for all nodes. Our algorithms are applicable to both directed and undirected graphs; however we focus on undirected graphs here for simplicity. Table 4.2 lists the notation used in the following discussion.

Table 4.2: Notation used in the chapter

Notation	Explanation
$G = (V_G, E_G)$	Database graph
n, n'	Database graph nodes
$P = (V_P, E_P)$	Pattern graph
v, v'	Pattern graph nodes
$\mathcal{M}$	Set of matches of P in G
$V_\sigma(G)$	Focal nodes, i.e., result of the SQL node selection predicates
M	A pattern match (i.e., a subgraph of G isomorphic to P)
m, m'	Nodes in a pattern match M
$N(x)$	Immediate neighbors of node x
$N^l(x)$	Neighbors of node x with label l
$N_k(x)$	Neighbors of node x in radius k
$S(n, k)$	k-hop neighborhood subgraph of node n
$\mu(v, M)$	Image of v in a match M. M is not stated when it is clear from the context.
$\mu^{-1}(m, M)$	The node in P which m matches. M is not stated when it is clear from the context.

4.3.1 Enumerating Candidates of Each Pattern Node

The first step of this algorithm is to enumerate the candidate database graph nodes for each pattern node. We utilize *node profiles* [9, 81] for this purpose. A node profile is a compact representation of a node's neighborhood that contains the number of neighbors for each label. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_L\}$ denote the L vertex labels. Then, the profile $P(n)$ of a node n is the vector: $\langle |N^{l_1}(n)|, |N^{l_2}(n)|, \ldots, |N^{l_L}(n)| \rangle$, where $N^{l_i}(n)$ denotes the set of neighbors of n having label l_i. A database graph node n is a candidate for a pattern graph node v if and only if $P(v)$ is contained in $P(n)$, i.e., for each label $l_i \in \mathcal{L}$ in $N(v)$, $|N^{l_i}(n)| \geq |N^{l_i}(v)|$. To make this filtering process fast, each database node profile is calculated once and stored along with the graph as an index. The result of this step is a set of database node candidates $C(v)$ for each pattern node $v \in V_P$.

4.3.2 Initializing the Candidate Neighbor Sets

Let v be a pattern node and v' be one of its neighbors in the pattern graph. For each node $n \in C(v)$ that is a candidate for v, we maintain a set of candidate neighbors with respect to v', denoted by $CN(n, v, v')$, i.e., neighbors of n that are a possible match to v'. We initialize each such set by finding the neighbors of n that have the same label as v', i.e., $CN(n, v, v') = C(v') \cap N(n)$.

4.3.3 Simultaneously Pruning the Candidates and Their Neighbors

Consider a pattern node v and a candidate node $n \in C(v)$. For every neighbor v' of v in the pattern graph, we must have that $CN(n, v, v')$ is non-empty. We use this observation to prune the candidate sets. We make passes over the candidate sets; in each pass, we remove those nodes from the candidate sets that do not satisfy this condition, and we then prune the candidate neighbor sets by identifying nodes n' such that $n' \in CN(n, v, v')$ but $n' \notin C(v')$. It is not hard to prove that the number of iterations is bounded by the number of nodes in the pattern graph (we omit the proof because of space constraints). Our approach is much simpler to implement than the approach based on semi-perfect matchings proposed by He et al. [9], but does not prune as aggressively for some types of query patterns; however, as we show in the experimental study, overall the performance of our approach is superior to theirs.

4.3.4 Extracting the Set of Matches from Candidate Sets

The output of the previous step is the set of candidates for each pattern node, along with their candidate neighbors. To find the final set of matches, we process these sets of candidates in a forward manner. For this purpose, we first choose an order of the pattern nodes such that each prefix of the order forms a connected component. Let $v_1, \cdots, v_{|V_P|}$ be that order. At step i, we produce the set of matches for the pattern subgraph consisting of $v_1, \cdots, v_i$ (and all edges between them). In step $i+1$, we grow the matches by adding possible matching nodes to v_{i+1}. Let $v_{j_1}, \cdots, v_{j_l}$, where $j_1 < j_2 < \cdots < j_l < i+1$, be the pattern nodes that are connected to v_{i+1} that appear before v_{i+1} in the chosen order. Then we find the possible matches for m_{i+1} efficiently by taking an intersection of candidate neighbor sets: $CN(n_{j_1}, v_{j_1}, v_{i+1})$, $CN(n_{j_2}, v_{j_2}, v_{i+1})$, ..., $CN(n_{j_l}, v_{j_l}, v_{i+1})$, and removing nodes that already appear in $n_1, \cdots, n_i$, if any. Since the candidate neighbor sets are typically small, this step can be done very efficiently (as opposed to prior work [9] where this check requires scanning over comparatively large candidate sets). If the intersection of the candidate neighbor sets is empty, then the corresponding partial match is discarded. In the experimental section, we show that utilizing candidate neighbors leads to orders of magnitude savings in finding pattern matches. Our proposed pattern matching algorithm is listed in Algorithm 1.

Input : Database graph $G = (V_G, E_G)$; pattern $P = (V_P, E_P)$; A permutation of the pattern nodes $v_1, v_2, ..v_{|V_P|}$ s.t. each prefix is a connected component of P

Output: Matches of P in G

```
1  for v ∈ V_P do
2      C(v) ← {};
3      for n ∈ V_G s.t. l(n) = l(v) do
4          if profile(v) ⊑ profile(n) then
5              C(v) ← C(v) ∪ n;
6              for v' ∈ N(v) do CN(n, v, v') ← C(v') ∩ N(n);
7  repeat
8      for v ∈ V_P, n ∈ C(v), v' ∈ N(v) do
9          if CN(n, v, v') = {} then C(v) ← C(v) − n;
10     for v ∈ V_P, n ∈ C(v), v' ∈ N(v), n' ∈ CN(n, v, v') do
11         if n' ∉ C(v') then
12             CN(n, v, v') ← CN(n, v, v') − n';
13 until no change in C and CN ;
   /* Let M_i denote the matches of pattern subgraph v_1, ..., v_i.      */
14 for n ∈ C(v_1), n' ∈ CN(n, v_1, v_2) do
15     M_2 ← M_2 ∪ (n, n');
16 for i = 2 to |V_P| − 1 do
17     for (n_1, ..., n_i) ∈ M_i do
18         for n_{i+1} ∈ ⋂_{v_j ∈ N(v_{i+1}), j<i+1} CN(n_j, v_j, v_{i+1}) do
19             if n_{i+1} not in (n_1, ..., n_i) then
20                 M_{i+1} ← M_{i+1} ∪ (n_1, ..., n_{i+1});
21 return M_{|V_P|};
```

Algorithm 1: Subgraph Pattern Matching Algorithm

4.4 Ego-centric Pattern Census Query Evaluation Algorithms

Next, we develop a suite of algorithms to solve the ego-centric pattern census problem. In this section, we present algorithms for evaluating queries of type:

```
SELECT ID, COUNT(PATTERN, SUBGRAPH(ID, k))
FROM NODES WHERE (PREDICATE)
```

We defer the discussion of how to handle queries involving subpatterns and pairwise

intersection/union search neighborhoods to the appendix where algorithm pseudo-codes are also presented.

We investigate two broad methods for answering such queries: *node-driven*, and *pattern-driven*, that can be seen as duals of each other. In node-driven algorithms, we start from the nodes and search for pattern matches in their neighborhoods, whereas in pattern-driven algorithms, we start from the pattern matches and look for the nodes whose neighborhoods contain those pattern matches. We assume the existence of a function `pattern-match(G,P)` which returns the set of all matches of the pattern P in the graph G. Furthermore, we refer to the set of database graph nodes selected as a result of applying the node restriction predicates as $V_\sigma(G)$.

4.4.1 Node-driven Algorithms

Perhaps the simplest node-driven algorithm, which we use as a baseline, works by extracting the k-hop subgraph around each node $n \in V_\sigma(G)$, denoted $S(n, k)$, and then performing pattern matching on that subgraph. This baseline algorithm (called **ND-BAS**), however, suffers from repeated and overlapping computations, especially for $k \geq 2$, and is computationally infeasible in practice. Next we propose two node-driven methods: pivot indexing and differential counting.

4.4.1.1 Pivot Indexing (**ND-PVOT**)

The pivot indexing algorithm starts with finding all pattern matches in the database graph, denoted by $\mathcal{M}$, and then counts the number of matches in each node's neighborhood. We use the `pattern-match` algorithm to find $\mathcal{M}$. Then, for each node $n \in V_\sigma(G)$, and for each pattern match $M \in \mathcal{M}$, we check if the nodes in M are entirely contained in $S(n,k)$. However, the naive way to do this requires $O(|V_\sigma(G)| * |\mathcal{M}| * |V_P|)$ checks, which makes this base algorithm impractical. We next introduce two optimizations to reduce the running time significantly.

Pattern Indexing: To avoid checking if every match $M \in \mathcal{M}$ is contained in $S(n,k)$ for every n, we index $\mathcal{M}$ so that the relevant subset of $\mathcal{M}$ can be retrieved when needed. For this purpose, we first designate a node v in the pattern graph as the *pivot* node, and build a pattern match index (denoted PMI_v) on $\mathcal{M}$ using the nodes corresponding to the pivot node in the matches. Let $PMI_v(n')$ denote the list of matches returned by the index for node n' (i.e., the list of pattern matches in which n' is the image of the pattern node v).

Now, to count the pattern matches in $S(n,k)$, we traverse the neighborhood of every node $n \in V_\sigma(G)$ in a breath first fashion starting with n until we reach the maximum depth k. For each node n' visited in this process, we retrieve $PMI_v(n')$ and for each match $M \in PMI_v(n')$, we check if V_M is completely contained within $N_k(n)$. Next we discuss how to efficiently reduce these containment checks further.

Avoiding Containment Checks: Let max_v denote the distance between v and the node farthest from it in the pattern graph. Let $d(n,n')$ denote the shortest

distance between n and n'. Then, if $d(n, n') + max_v \leq k$, any pattern match in $M \in PMI_v(n')$ must be completely contained in $S(n, k)$.

Thus, for any node $n' \in V_G$ for which $d(n, n') \leq k - max_v$, we can avoid checking whether each $M \in PMI_v(n')$ is entirely contained in $S(n, k)$ and instead we simply add $|PMI_v(n')|$ to the overall pattern match count for n. On the other hand, if $d(n, n') + max_v > k$, we need to explicitly check whether all nodes in $PMI_v(n')$ are in $S(n, k)$. Specifically, let v' denote a node in the pattern graph such that $d(v, v') + d(n, n') > k$. Then, we must explicitly check whether the corresponding node in M is within k hops from n. However, if $d(v, v') + d(n, n') \leq k$, then this check can be avoided. Note that both $d(v, v')$ and $d(n, n')$ are easily computed (the former can be pre-computed once for the pattern graph, whereas the latter is available since we are using breadth first search).

Pivot Selection: Finally, the choice of the pivot node v becomes critical for the performance of this algorithm. However, it is easy to see that choosing the node with the minimum value of max_v is optimal with respect to the number of database nodes for which we have to do explicit checks, i.e.,

$$v = argmin_{x \in V_P}\{d(x, argmax_{y \in V_P}\{d(x, y)\})\}$$

The pseudocode is listed in Algorithm 2.

Handling Subpatterns: In this algorithm, handling subpatterns is straightforward. As before, pattern matching is performed using the entire pattern graph; however, the pivot is selected from the set of subpattern nodes $V_{SP} \subseteq V_P$, and

Input : Database graph G; pattern P; set of nodes $V_\sigma(G)$; neighborhood radius k

Output: The number of matches of P within k hops of each node of $V_\sigma(G)$

1 $v \leftarrow argmin_{x \in V_P}\{d(x, argmax_{y \in V_P}\{d(x, y)\})\}$;
2 $max_v \leftarrow d(v, argmax_{y \in V_P}\{d(x, y)\})$;
3 **for** $u \in V_P$ **do**
4 **for** $i \leftarrow 1$ **to** max_v **do**
5 **if** $d(v, u) \geq i$ **then** $distant[i] \leftarrow distant[i] \cup u$
6 $\mathcal{M} \leftarrow$`pattern-match(`G, P`)`;
7 $PMI_v \leftarrow$`build-pmi-index(`$\mathcal{M}, v$`)`;
8 **for** $n \in V_\sigma(G), n' \in N_k(n)$ **do**
9 **if** $max_v + d(n, n') \leq v$ **then**
10 $counts[n] \leftarrow counts[n] + |PMI_v[n']|$;
11 **else**
12 **for** $M \in PMI_v[n']$ **do**
13 **if** $\mu(distant[k - d(n, n') + 1], M) \subseteq N_k(n)$ **then**
 $counts[n] \leftarrow counts[n] + 1$;
14 **return** *counts*;

Algorithm 2: Pivot Indexing Algorithm

the distance checks are only done for the database graph nodes that match the subpattern nodes.

Handling Pairwise INTERSECTION and UNION: In the case of intersection and union, the outer loop (line 9) iterates over pairs of nodes $(n_1, n_2) \in V_\sigma^2(G)$, where $V_\sigma^2(G)$ is the set of selected pairs, and the $N_k(n)$ is replaced with the set of nodes in $N_k(n_1) \cap N_k(n_2)$ and $N_k(n_1) \cup N_k(n_2)$ for intersection and union, respectively (lines 10 and 15). The distance $d(n, n')$ is replaced with $max(d(n_1, n'), d(n_2, n'))$ and $min(d(n_1, n'), d(n_2, n'))$, respectively.

4.4.1.2 Differential Counting (**ND-DIFF**)

The second node-driven approach that we investigate is based on the idea of exploiting shared neighborhoods – Zhang et al. [80] use a similar idea for building a pairwise signature index in their proposed approach for subgraph pattern matching. Let $\mathcal{M}[n]$ denote the set of pattern matches contained in $S(n, k)$. Given two nodes, n and n', and $\mathcal{M}[n]$, we can construct $\mathcal{M}[n']$ by: (1) removing all matches $M \in \mathcal{M}[n]$ for which at least one node in M is present in $N_k(n) - N_k(n')$, and (2) by finding additional matches that contain nodes present in $N_k(n') - N_k(n)$ and that are fully contained in $S(n', k)$.

As above, we start with finding all pattern matches $\mathcal{M}$ using the `pattern-match` algorithm. We then build a modified pattern match index that indexes $\mathcal{M}$ using all the nodes in the match (instead of just the pivot node). In other words, $PMI[n]$ contains all pattern matches that contain n. We start with an arbitrary database graph node n and compute $\mathcal{M}[n]$ using $PMI[n]$ (using a technique very similar to the above algorithm). We then pick an arbitrary neighbor n' of n and compute $\mathcal{M}[n']$ using $\mathcal{M}[n]$. The detailed algorithm is listed in Algorithm 3.

Differential counting is appropriate for finding node-centric counts of compact structures such as nodes or edges, but more complex patterns will likely have parts that fall in unshared areas, making differential counting less effective in such cases. Furthermore, picking a random neighbor does not always guarantee that the shared neighborhood is large enough (we experimented with a heuristic based on shingle ordering [113], but the results were essentially the same and hence we don't report

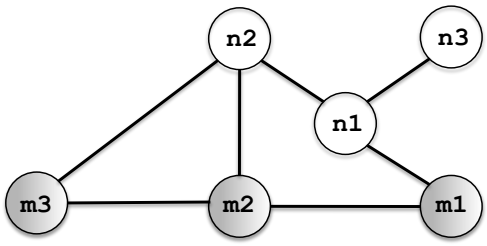
Input : Database graph G; pattern P; set of nodes $V_\sigma(G)$; neighborhood radius k

Output: The number of matches of P within k hops of each node of $V_\sigma(G)$

```
1  M ← pattern-match(G, P);
   /* Index the matches on all the pattern nodes                    */
2  PMI ← build-pmi-index(M, V_P);
3  S ← V_σ(G);
4  current ← Next element from S;
5  M_current ← {};
6  while S is not empty do
7      S ← S − current;
8      if prev = NULL then
9          N_1 ← N_k(current);
10         N_2 ← {}; M_current ← {};
11     else
12         N_1 ← N_k(current) − N_k(prev);
13         N_2 ← N_k(prev) − N_k(current);
14     for n ∈ N_1, M ∈ PMI[n] do
15         if V_M ⊆ N_k(current) then
16             M_current ← M_current ∪ M;
17     for n ∈ N_2 do  M_current ← M_current − PMI[n];
18     counts[current] ← |M_current|;
19     if there exists n s.t. n ∈ S ∩ N(current) then
20         prev ← current;  current ← n;
21     else
22         prev ← NULL;  current ← Next elem. from S;
23 return counts;
```

Algorithm 3: Differential Counting Algorithm

those here). In addition, if there is a selection predicate that specifies a subset of nodes to do pattern census for, then sharing opportunities may be rarer (especially with selective predicates). In our experimental evaluation, pivot indexing technique always outperformed differential counting.



(a)

#	Queue	Head
1	$m_1(0,1,2), m_2(1,0,1),\ m_3(2,1,0)$	m_1
2	$m_2(1,0,1), m_3(2,1,0), n_1(1,2,3)$	m_2
3	$m_3(2,1,0), n_1(1,2,3), n_2(2,1,1)$	m_3
4	$n_1(1,2,3), n_2(2,1,1)$	n_1
5	$n_2(2,1,1), n_3(2,3,4)$	n_2
6	$n_3(2,3,4), n_1(1,2,2)$	n_3
7	$n_1(1,2,2)$	n_1
8	$n_3(2,3,3)$	n_3
9	ϕ	–

(b)

#	Queue	Head
1	$m_2(1,0,1), m_1(0,1,2), m_3(2,1,0)$	m_2
2	$m_1(0,1,2), m_3(2,1,0), n_2(2,1,2)$	m_1
3	$m_3(2,1,0), n_2(2,1,2), n_1(1,2,3)$	m_3
4	$n_2(1,2,2), n_1(1,2,3)$	n_2
5	$n_1(1,2,2)$	n_1
6	$n_3(2,3,3)$	n_3
7	ϕ	–

(c)

Figure 4.2: (a) Example used to illustrate the advantage of best-first traversal order. (b) and (c) Simultaneous node expansions around the pattern match $\{m_1, m_2, m_3\}$ using breadth-first and best-first approaches, respectively.

4.4.2 Pattern-driven Algorithms

The second class of algorithms that we propose start with the pattern matches and look for nodes that contain the pattern match within their neighborhoods. This can be seen as dual to the node-driven algorithms in that, here we process each pattern match once, but may process the database nodes multiple times, whereas in node-driven algorithms, we process each node once, but may process each pattern

match multiple times.

The baseline pattern-driven algorithm (called **PT-BAS**) processes the pattern matches in the database graph independently one at a time. As before, let $S(n, k)$ denote n's k-hop neighborhood subgraph. For each pattern match $M = (V_M, E_M)$, for each node $m_i \in V_M$, we traverse $S(m_i, k)$ in a breadth-first fashion, and for each node in $S(m_i, k)$, we compute its distance from m_i. We then find the node $m_{min} \in V_M$ with the least number of k-hop neighbors, and for each of its k-hop neighbors, we check whether that neighbor is reachable within k hops from every other node in V_M.

Next we discuss a series of optimizations that improve upon this baseline algorithm.

4.4.2.1 Simultaneous Traversal

In the baseline algorithm, an edge may be traversed multiple times if it is shared among the neighborhoods of the nodes in V_M (this will often be the case). We reduce the number of such edge traversals by traversing the neighborhoods of all nodes in V_M *simultaneously*, using a breadth-first algorithm whose queue is initialized with V_M. In each step, we remove and process one node from the queue. With each visited node n, for each pattern match node $m \in V_M$, we maintain $PMD_m[n]$, the current upper bound on the distance between n and m. When a node n is visited, we update the distance vector for its neighbor n' according to the relation: $PMD_m[n'] = min(PMD_m[n] + 1, PMD_m[n'])$ for each $m \in M$. If at least

one of the distances is updated, then n' is pushed on the queue. The algorithm terminates when the queue is empty. Initially $PMD_m[m] = 0$ for each $m \in V_M$ and is equal to ∞ (or $k+1$) otherwise.

4.4.2.2 Distance Shortcuts

We can save some initial PMD computation steps by utilizing the fact that the pattern P is isomorphic to any pattern match $M \in \mathcal{M}$. We find the distances between every pair of nodes v, v' in the pattern, and reuse these to initialize PMD for the nodes in V_M for each match M. Specifically, for $m, m' \in V_M$, we set $PMD_m[m'] = d(\mu^{-1}(m), \mu^{-1}(m'))$ if $d(\mu^{-1}(m), \mu^{-1}(m')) \leq k$, and initialize it to $k+1$ otherwise (recall that $\mu^{-1}(m)$ denotes the pattern node $\in P$ that matches the node $m \in M$).

4.4.2.3 Best-first Ordering

Depending on the order in which the nodes are visited, unnecessary traversals can still occur despite the above two optimizations. Here we present a heuristic approach to further minimize the unnecessary computation by choosing which node to visit next. Specifically, we choose the node with the minimum $score(n) = \Sigma_{m \in V_M} PMD_m[n]$ in the queue to visit next. The intuition behind this heuristic is that the node with lowest $score()$ value is the node that is closest (of the remaining nodes) to all the pattern match nodes combined, and likely more influential in determining the distances from the pattern match nodes.

As an example, consider the graph in Figure 4.2(a). In this graph, the pattern match nodes are m_1, m_2 and m_3, and $k = 3$. Figure 4.2(b) shows the operation of the simultaneous breadth-first traversal approach. Initially, the traversal queue is initialized with the three M nodes, m_1, m_2, m_3, along with their PMD values for (m_1, m_2, m_2). At each step, the node n_h at the head of the queue is removed and its neighbors are inserted into the queue along with their PMD values if they do not exist, or their PMD values are updated if they already exist. In Figure 4.2(b), we observe that in step 4, n_1 is examined before n_2, which is examined in step 5. As a result, when n_2 is examined, it updates the PMD of n_1, causing it to be reinserted, and subsequently causing n_3 to be reinserted too. Figure 4.2(c) shows the operation of the algorithm by employing the best-first order. It can be seen that the reinsertions of nodes n_1 and n_3 have been eliminated, and each node is visited exactly once. The details of the algorithm are provided in Algorithm 4.

Although the best-first approach reduces the number of traversals, it comes with an additional cost of having to maintain a priority queue, which requires $O(\log |Q|)$ time for insertion and deletion, where $|Q|$ is the queue size. However, in our implementation, we eliminate the cost of maintaining a heap-based priority queue by observing that the range of possible scores is pre-defined and small. Specifically, $score(n) \leq (k+1)|V_P|$ (since $PMD_m[n] \leq k+1$). Hence, we use an array-based priority queue where we store the nodes with score equal to i at position i, leading to a complexity of $O(1)$ for both insertions and deletions.

4.4.2.4 Center-based Expansion

Best-first ordering is aimed at reducing the number of node reinsertions into the queue; a node reinsertion may cause its neighbors to be reinserted and hence is an expensive operation. However, best-first ordering does not entirely eliminate node reinsertions. Our next optimization is based on the idea of identifying a set of *important* nodes and making sure they are not reinserted into the queue. Let $\mathcal{C} \in V_G$ denote the set of nodes (called *centers*) that are picked apriori for this purpose. We pre-compute the distances $d(c, n)\ \forall\ c \in \mathcal{C}, n \in V_G$. At query time, we insert these nodes along with their scores (computed at query time) to the traversal queue as part of the queue initialization, i.e., $PMD_m[c] = d(c, m)$ for all $c \in \mathcal{C}$ and $m \in V_M$. Now once these nodes are visited (and their neighbors processed), they will never be reinserted into the queue. Further, we can use the *triangle inequality* to get tighter upper bounds on the distances for other nodes. For any $m, n', c \in V_G$, we have that $d(m, n') \leq d(m, c) + d(c, n')$. So when we visit a node n whose neighbor n' is not yet initialized, we can set the PMD values of n' as:

$$PMD_m[n'] = min(PMD_m[n] + 1, min_{c \in \mathcal{C}}(d(m, c) + d(c, n')))$$

Our final task is to choose a set of centers apriori. Many network centrality measures have been proposed in the social network analysis literature to reflect various notions of importance in social networks [114] that we can use for this purpose, including *page rank*, *betweenness centrality*, *closeness centrality*, to name a few. In our implementation, we pick $\mathcal{C}$ to be the set of nodes with the highest

degree centrality, i.e., the nodes with the highest degrees, primarily due to its low computation cost compared to other centrality measures.

4.4.2.5 Pattern Match Clustering

The algorithm presented so far processes each pattern match independently. Since many pattern matches may be close together, and in fact may overlap, processing groups of them together could potentially lead to more savings. However, the trade-off here is a larger number of distance computations – for a pattern match M that is processed in isolation, we compute distances of all nodes in M to all nodes that are within k hops of at least one node in M. If we were to process multiple pattern matches together, a larger set of distances has to be computed (for every node in a pattern match, we have to compute distances to all database nodes that are within k hops of a node in any pattern match).

We use the center distance index along with the K-means clustering algorithm to group pattern matches together. For each match M, we construct a feature vector:

$$F(M) = \langle d(c_1, m_1), d(c_1, m_2), \ldots, d(c_{|\mathcal{C}|}, m_{|V_P|}) \rangle$$

After computing these feature vectors for all the matches, we use the K-means clustering algorithm [115] to cluster the matches into K clusters. (We discuss the issues in choosing K in the next section.) We then process each cluster independently by simultaneously expanding around all pattern matches in the cluster.

Incorporating this final optimization gives us our proposed pattern-driven al-

gorithm (called **PT-OPT**). The details of the algorithm are listed in Algorithm 4.

(We omit the pattern clustering optimization for simplicity.)

Input : Database graph G; pattern P; set of nodes $V_\sigma(G)$; neighborhood radius k; set of centers $\mathcal{C}$

Output: The number of matches of P within k hops of each node of $V_\sigma(G)$

```
1  M=pattern-match(G, P);
2  for M ∈ M do
3      for m ∈ V_M, m' ∈ V_M do
4          if d(μ^-1(m), μ^-1(m')) ≤ k then
5              PMD_m[m'] ← d(μ^-1(m), μ^-1(m'));
6          else
7              PMD_m[m'] ← k + 1;
8      Q ← V_M;
9      while Q is not empty do
10         n ← argmin_{q∈Q} Σ_{m∈V_M} PMD_m[q];
11         dequeue(Q, n);
12         near ← TRUE;   far ← TRUE;
13         for m ∈ V_M do
14             if PMD_m[n] > k then near ← FALSE;
15             if PMD_m[n] < k then far ← FALSE;
16         if near then N[M] ← N[M] ∪ n;
17         if not far then
18             for n' ∈ N(n) do
19                 noChange ← TRUE;
20                 for m ∈ N(M) do
21                     if PMD_m[n'] = NULL then
22                         noChange ← FALSE ;
23                         PMD_m[n'] ←
                           min(PMD_m[n] + 1, min_{c∈C}(d(m, c) + d(c, n')));
24                     if PMD_m[n'] > PMD_m[n] + 1 then
25                         noChange ← FALSE ;
26                         PMD_m[n'] ← PMD_m[n] + 1;
27             if not noChange then enqueue(Q, n');
28     N[M] ← N[M] ∩ V_σ(G);
29     for n ∈ N[M] do  counts[n] ← counts[n] + 1;
30 return counts;
```

Algorithm 4: Pattern-driven Algorithm

Handling Subpatterns: To handle subpatterns, we use $\mu(V_{SP}, M)$ instead of V_M in the algorithm. In other words, for each match M, we only consider its subgraph incident on the nodes in V_M that match nodes in the subpattern.

Handling Pairwise INTERSECTION and UNION: To handle INTERSECTION, we note that all the pairs in $\mathcal{N}[M]$ already have the pattern in the intersection of their neighborhood. Therefore, instead of adding the match M to each node in $\mathcal{N}[M]$, we add it to each node pair in $\mathcal{N}[M] \times \mathcal{N}[M]$. For UNION, for each match M, we partition the set $N(M)$ into all possible size 2 partitions P_1, P_2. We denote nodes reachable from P_1 and P_2 by $\mathcal{N}[P_1]$ and $\mathcal{N}[P_2]$, respectively. The match M is added for each pair of nodes $(n_1, n_2) \in \mathcal{N}[P_1] \times \mathcal{N}[P_2]$. Because of the requirement to partition the pattern in different ways, and perform the computation in on every partitioning way, pattern-driven UNION evaluation is only useful for very simple and selective patterns.

4.5 Experimental Evaluation

In this section, we present the results of a comprehensive experimental evaluation using our prototype implementation, which is written in Java on top of the disk-based graph database engine Neo4j [116]. We begin with comparing our graph pattern matching algorithm with the prior approach by He et al. [9], and demonstrate that our approach of using candidate neighbor sets results in orders of magnitude savings. We then compare the performance of our node-driven and pattern-driven algorithms, and we study the effect of the various optimizations pro-

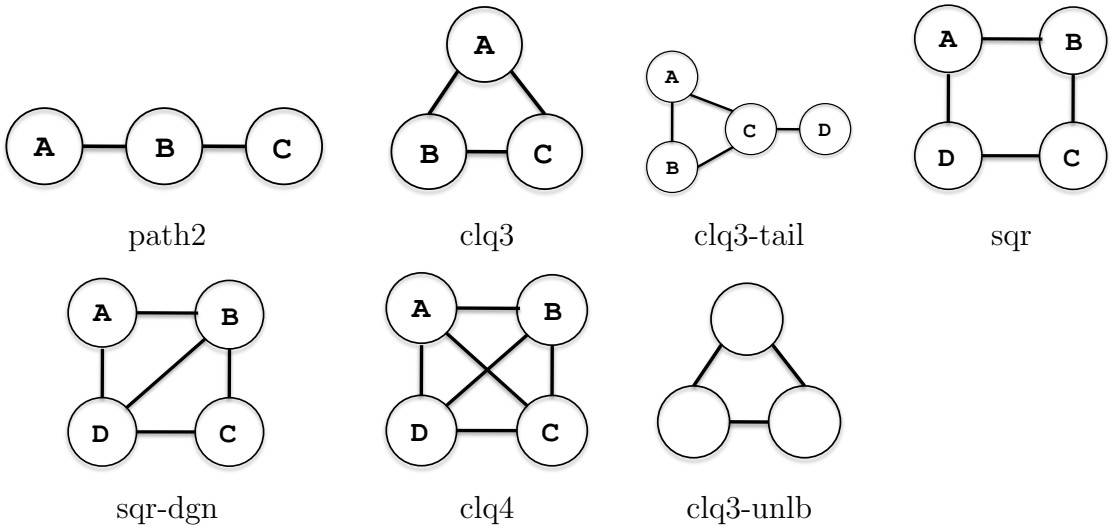


Figure 4.3: Query patterns used in the synthetic dataset experiments – the letters inside the circles indicate the label of the node.

posed for pattern-driven algorithms in detail. Furthermore, we discuss a real-world experiment, where we solve a link prediction problem over DBLP through our framework and report its results.

For the first set of experiments, we use synthetic database graphs generated according to the preferential attachment model [53]. For labeled graphs, the labels are generated randomly. The graph sizes vary from 20K nodes to 1M nodes, with the number of edges 5× the number of nodes in all graphs. The patterns used in the experiments are shown in Figure 4.3. All experiments were performed on identical Linux machines with 2.2 GHz quad-core processor, 8 GB of RAM, and a 750 GB 7200 RPM disk drive.

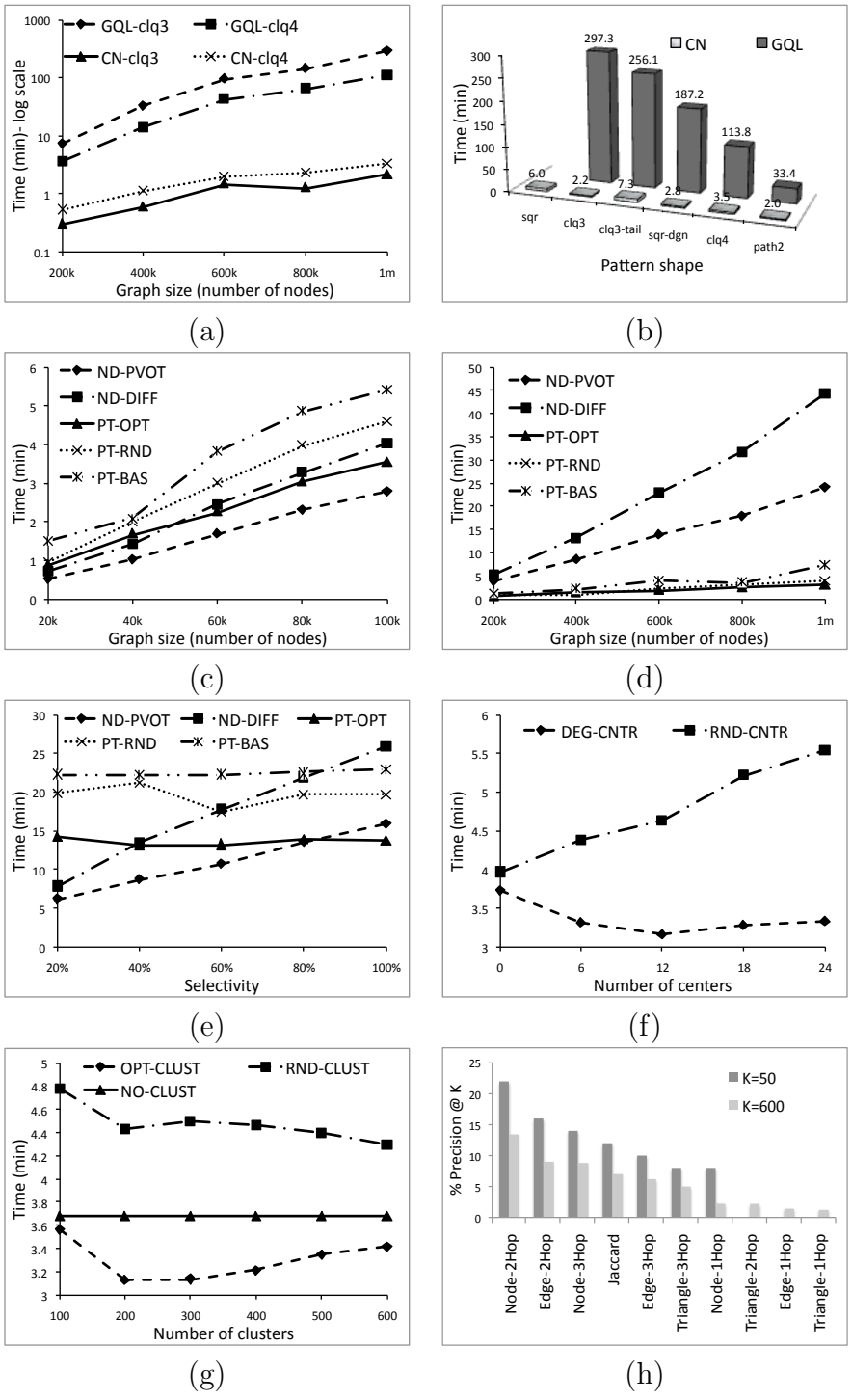


Figure 4.4: (a) Comparison with GQL for different graph sizes and (b) for different patterns; (c) Pattern census: varying graph size (unlabeled graphs); (d) Pattern census: varying graph size (labeled graphs); (e) Pattern census: varying node selectivity; (f) Effect of centers on the pattern-driven algorithm; (g) Effect of clustering on the pattern-driven algorithm; (h) Precision @50 and @600 of DBLP link prediction using different structures and hop lengths.

4.5.1 Experiments Using Synthetic Datasets

We begin with comparing the performance of our pattern matching algorithm (called CN) with GraphQL system [9], also written in Java (called GQL). For this purpose, we use the executable binaries that we obtained from the authors of the system.

4.5.1.1 Comparison with GQL for Different Graph Sizes

Figure 4.4(a) shows the results (in log scale) of comparing CN with GQL for varying graph sizes (from 200K nodes/1M edges to 1M nodes/5M edges), with labels drawn randomly from a set of 4 labels, for two query patterns: clq3 and clq4 (Figure 4.3). As we can see, our algorithm is orders of magnitude better in almost all cases (with speedups ranging from 10 to 140). Our detailed study (omitted because of space constraints) indicates that the speedups are attributable, in large part, to the use of candidate neighbor sets.

4.5.1.2 Comparison with GQL for Different Patterns

Here we compare CN and GQL using a 1M node graph (with 5M edges), for the the labeled query patterns shown in Figure 4.3. The results are shown in Figure 4.4(b). GQL takes 37 hours for calculating matches for sqr (480 times the runtime of CN); therefore, we do not show that point on the graph. The results confirm that our algorithm outperforms GQL by orders of magnitude.

For the next three experiments (3, 4 and 5), we compare the following pattern census

query evaluation algorithms:

Node-driven baseline (ND-BAS): In this algorithm, we extract $S(n, k)$ for each node and use the pattern matching algorithm to count the number of matches.

Node-driven differential counting method (ND-DIFF): This method, based on the GADDI index method in [80], traverses the nodes in the graph in some order, and computes the pattern matches for one node by utilizing the pattern matches for the prior node in the sequence.

Node-driven pivot method (ND-PVOT): Our proposed pivot indexing node-driven algorithm.

Pattern-driven baseline (PT-BAS): The baseline algorithm presented in Section 4.4.1.

Optimized pattern-driven algorithm (PT-OPT): The proposed pattern-driven algorithm with all the proposed optimizations. Unless otherwise is stated, the number of clusters is set to be the number of matches divided by 4, and we use 12 centers. The number of K-means iterations is 10.

Random-first pattern-driven algorithm (PT-RND): The proposed pattern-driven algorithm with all the proposed optimizations except best-first traversal. Instead, we choose the next node to process from the queue randomly.

4.5.1.3 Varying Graph Size – Unlabeled Graphs

Here we compare the performance of the 6 algorithms in evaluating the query (with $k = 2$):

```
SELECT ID, COUNT(clq3-unlb, SUBGRAPH(ID, 2))
FROM NODES
```

We vary the graph size from 20K nodes to 100K nodes. The results are shown in Figure 4.4(c). We do not plot the running time of ND-BAS – for 20K nodes, the runtime of ND-BAS is 116 minutes, which is 218 times higher than our best performing algorithm (ND-PVOT). We see that ND-PVOT outperforms not only the other node-driven algorithms, but also the pattern-driven algorithms. This is because the query pattern (unlabeled triangle) is not very selective, i.e., the number of matches is quite high, and hence the approaches based on searching from patterns do not perform as well. We observed consistent behavior for other non-selective query patterns.

4.5.1.4 Varying Graph Size – Labeled Graphs

Here we use graphs with node labels randomly chosen from a set of 4 labels, and vary the graph size from 200K nodes to 1M nodes. We use a similar query as above ($k = 2$) but use a labeled triangle pattern (clq3) instead. As we can see (Figure 4.4(d)), PT-OPT significantly outperforms the other pattern-based algorithm, including PT-RND, illustrating the importance of the best-first order in reducing the overall runtime. Pattern-driven algorithms generally outperform node-driven algorithms because the query pattern is more selective in this case.

4.5.1.5 Varying Focal Node Selectivity

Next, we vary the selectivity of the focal nodes specified in the query, controlled by the `WHERE` clause. We use an unlabeled 500K database graph. The query is:

```
SELECT ID, COUNT(clq3-unlb, SUBGRAPH(ID, 2))
FROM NODES WHERE RND()<R
```

where we vary R from 20% to 100%. As shown in Figure 4.4(e), performance of pattern-driven algorithms is not affected by the focal nodes' selectivity, because those algorithms start from the pattern matches and examine their neighborhood irrespective of whether the nodes in the neighborhood are selected or not. On the other hand, running time of the node-driven methods increases linearly with the selectivity, and eventually becomes worse than pattern-driven methods.

4.5.1.6 Effect of the Number of Centers on Pattern-driven Algorithm

Next we examine the effect of both the number of centers and how they are chosen, using a labeled graph of 1M nodes and 5M million edges, and 4 labels. The query is:

```
SELECT ID, COUNT(clq3, SUBGRAPH(ID, 2))
FROM NODES
```

We compare the proposed way of choosing centers, i.e., using nodes with the highest degree (DEG-CNTR) versus using randomly chosen centers (RND-CNTR). For both methods, we vary the number of centers from 0, which corresponds to not using centers, to 24 centers. Note that the number of centers affects both (1) the clustering

quality and (2) distance initializations in the pattern match neighborhoods (PMD). The purpose of this experiment is to study (2) in isolation of (1) since using too few centers clearly degrades the clustering quality and the overall performance. Therefore, we isolate the effect of (1) in this experiment by fixing the number of centers that are used for clustering regardless of the number of centers used for PMD. The results are shown in Figure 4.4(f). We can see that using the high-degree nodes as centers greatly helps the query performance, whereas with random centers the performance worsens with increasing number of centers. On the other hand, looking at the performance of DEG-CNTR as the number of centers increases, we observe that the performance initially improves, but as the number of centers becomes too large, the overheads of using centers start dominating.

4.5.1.7 Effect of Pattern Clustering

Finally we study the effect of the pattern clustering optimization on the performance of our proposed pattern-driven algorithm using a labeled graph of 1M nodes and 5M edges, and 4 labels. The query is:

```
SELECT ID, COUNT(clq3, SUBGRAPH(ID, 2)) FROM NODES
```

We compare the performance of three alternatives: (1) no clustering (NO-CLUST), (2) random clustering (RND-CLUST), and (3) the proposed K-means approach that is based on using the centers (OPT-CLUST). We also vary the number of clusters from 100 to 600 to show the effect of changing the number of clusters on the performance. Note that this parameter has no effect on NO-CLUST.

The results are shown in Figure 4.4(g). We observe that OPT-CLUST significantly outperforms both RND-CLUST and NO-CLUST, illustrating both the benefits of clustering and the need to choose the cluster carefully. Furthermore, we can see that there is a trade-off in setting the number of clusters – with too large a number of clusters (600), there is no significant advantage to using clusters since the matches are largely processed independently, but the performance also degrades with too few clusters (100). This is because in the latter case, there are too many matches in each cluster and the resulting redundant distance computations outweigh the benefits of clustering.

4.5.2 Real-world Experiment

In this experiment, we utilize our language to compare the predictive power of different structures in predicting future scientific collaborations (this is an example of a *link prediction* task). We collected publication data from SIGMOD, VLDB and ICDE conferences from 2001 to 2010. Given the co-authorship information from years 2001 to 2005, we predict collaborations in the period from 2006 to 2010. For this purpose we defined 9 pairwise measures using our language. For each pair of authors, we measure the number of nodes, edges and triangles in their common 1, 2, and 3 hop neighborhoods. In other words, we use a query of the form:

```
SELECT N1.ID, N2.ID,
COUNT(struct, SUBGRAPH-INTERSECTION(N1.ID, N2.ID, r))
FROM NODES AS N1, NODES AS N2, EDGES AS E
```

where *struct* represents a node, edge, or triangle pattern, and k is 1, 2 or 3, resulting in 9 total configurations. In the prediction step, for each configuration, we pick the top K pairs in terms of their common structures (i.e., the pairs of authors with the highest counts for the corresponding pattern), and then measure the precision at K defined as the number of correct predictions divided by K. Figure 4.4(h) shows the precision of each of the nine configurations at $K = 50$ and $K = 600$. In addition to the nine measures, the figure shows the performance of Jaccard coefficient, a similarity measure that is regarded as a good predictor and commonly used in link prediction [22]. We also measured the precision of the random predictor (which selects random K pairs of nodes) and it yielded a zero precision at both $K = 50$ and $K = 600$. For our measures, common nodes within 2 hops has the strongest prediction power, almost twice that of Jaccard coefficient. Several other measures also outperform Jaccard coefficient. This simple experiment illustrates the power of our framework in enabling social network analysis.

Chapter 5

Uncertain Graph Data Querying

5.1 Introduction

Although graph cleaning approaches proposed in Chapter 3 can be used to address the issues of uncertainty and noise in graph data before querying or analyzing it, uncertainty and noise can also be directly be modeled by associating probability distributions with graph data, and then, the probabilistic graph data can be directly queried. Therefore, in this chapter, we study the problem of querying uncertain graphs with identity uncertainty. We present our proposed approach for modeling graphs with attribute uncertainty, edge existence uncertainty, and identity uncertainty. Furthermore, we propose an efficient algorithm to perform subgraph pattern matching queries over such uncertain graphs. Although some earlier work, e.g., [98, 99, 100, 117], has addressed the issues of uncertainty in graphs, none of that work has considered the three types of uncertainty together, and none of them has addressed the problem of modeling and querying graphs with identity linkage uncertainty. Before proceeding to the formal model, problem statement and proposed solutions, we discuss a detailed motivating example.

5.2 Motivating Example

Consider a case where we want to build a system to help organizations find experts in different domains and at different levels. We achieve that by integrating information about experts and their affiliations from multiple sources. In this example, we assume three sources: an online professional network (e.g., LinkedIn), an online social network (e.g., Facebook), and personal webpages or blogs. We consider the experts' names, their affiliations (specifically, `Academia` (a), `Research Lab` (r), or `Industry` (i)), and relationships between experts. Figure 5.1 illustrates a small example, where we omit names for clarity. We use the term *reference* to denote the *observed objects*, which in this example are strings encoding names, while we use the term *entity* to refer to *real-world objects*, that is, the experts in our case. A real-world object may thus correspond to a collection of references, as names may be abbreviated, misspelled, etc. In Figure 5.1(a), nodes represent references, letters inside nodes represent reference IDs, and letters outside nodes represent *labels*, that is, affiliations, along with their probabilities in parentheses. Consider node r_1, extracted from a personal webpage. Suppose that a text analysis method suggests that the name is "Gerald Maya" and the affiliation is `industry` with probability 0.75 and a `Research Lab` with probability 0.25. Nodes r_2 and r_3 are extracted from an online professional network, with name "Becky Castor" and an `Academia` affiliation, and the name "Christopher Tucker" and a `Research Lab` affiliation, respectively. Finally, node r_4 is extracted from an online social network, with the name "Chris Tucker" and an `Industry` affiliation. Furthermore, relationships between the indi-

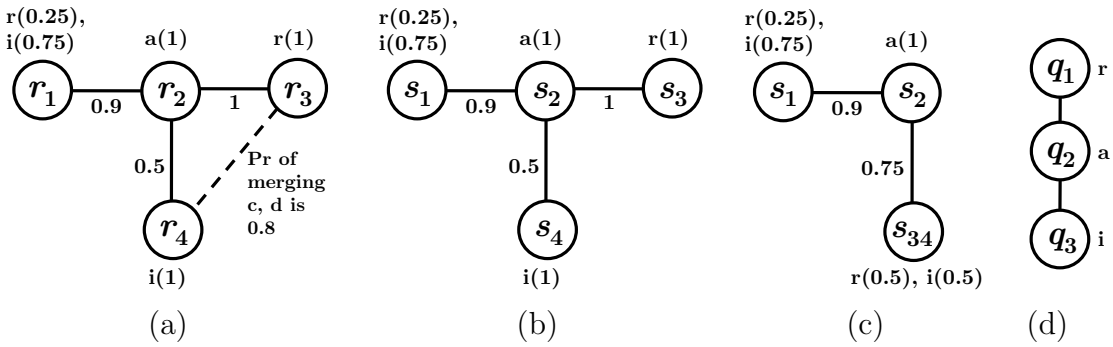


Figure 5.1: (a) Reference-level network, (b), (c) the two possible entity graphs, (d) a query graph

viduals are extracted (represented as edges in the figure) and are associated with probabilities that reflect the likelihood of the relationship's existence. These probabilities can be calculated based on whatever information or signals is available from these online resources, such as the number of common connections or shared attributes or the history of communication. Since "Christopher Tucker" and "Chris Tucker" seem to be the same person based on name similarity, we put them together in the same *reference set* to indicate that these two references may refer to the same entity (depicted as a dashed line in the figure). To quantify identity uncertainty, which is the uncertainty of having multiple references referring to the same real-world entity, we assign this reference set a probability of 0.8, denoting the likelihood that the elements in the set correspond to a single real-world entity.

Figures 5.1(b) and (c) illustrate the two possible sets of entities with their labels and relations for the example reference network shown in Figure 5.1(a), where the letters inside the nodes represent entity IDs. Figure 5.1(b) depicts the entity graph in which r_3 and r_4 remain unmerged, i.e., assumed to be separate real-world entities, ($pr = 0.2$), and Figure 5.1(c) depicts the case where they are merged, i.e.,

assumed to be the same real-world entities, ($pr = 0.8$) to form a new node s_{34} with its own label and edge probability distributions. Going from a set of references to an entity requires merging the information associated with the references, that is, their labels and the relationships they participate in. In this example, we simply average the probability distributions. Since r_3 has label r and r_4 has label i, we assign a label distribution of $r(0.5), i(0.5)$ to entity s_{34}. Similarly, s_{34} has an edge to s_2 with $pr = 0.75$ (average of r_3's edge with $pr = 1$ and r_4's edge with $pr = 0.5$).

Clearly, we want to specify queries to our information system at the level of entities rather than references. In this work, we focus on subgraph pattern matching queries, perhaps the most widely used and studied class of queries over graphs. Figure 5.1(d) depicts a query which asks for all paths of length 2 over nodes labeled (r, a, i). In addition to the query graph, a query specifies a minimum threshold α, which we set to 0.25 in this example, to indicate that only matches with probability larger than α should be returned. In this simple case, we can answer our query by examining all possible matches. In the entity graph in Figure 5.1(b), with r_3 and r_4 unmerged, the nodes (s_3, s_2, s_4) form a path with the required labels. The probability of that path is computed by multiplying together the three node label probabilities (1, 1, 1), the two edges probabilities (1, 0.5), and the probability that the nodes r_3 and r_4 are *not* merged (0.2); the resulting probability of the match is 0.1, which is below our cutoff of 0.25. There are two more potential matches, (s_1, s_2, s_4) and (s_3, s_2, s_1), but neither of them satisfies the minimum threshold constraint. In the second entity graph in Figure 5.1(c), there are two potential matches for the query: (s_1, s_2, s_{34}) and (s_{34}, s_2, s_1). The probability of (s_1, s_2, s_{34}) being a

match to the query is 0.084, which does not meet our threshold, whereas the probability of (s_{34}, s_2, s_1) is 0.253. Therefore, (s_{34}, s_2, s_1) is the only answer to our query. Clearly, such an exhaustive approach is infeasible in practice for larger graphs. In the remainder of this chapter, we therefore develop a scalable approach to answer subgraph pattern matching queries in this setting.

5.3 Uncertain Graph Modeling

We now discuss our formal model for the types of uncertainties arising in situations as described in the example above, where we are given information about *references*, or mentions of objects, but are interested in queries about *entities*, or the objects themselves. We introduce *probabilistic entity graphs*, which define a probability distribution over graphs describing entities, their labels, and links between them. The key challenge here is that references induce constraints on which entity nodes can co-occur in the same graph, as each graph structure corresponds to one possible way of assigning references to existing entities. To deal with these dependencies, we represent our probability distribution as a *probabilistic graphical model* (PGM) [118]. After a quick summary of the necessary basics, we introduce the notion of a *probabilistic graph description* (PGD), and show how the PGD in turn defines a probabilistic entity graph. We first focus on the basic case, where distributions over labels and links are all independent, and then show how additional dependencies can directly be introduced.

A PGM $\mathcal{P} = \langle \mathcal{V}, \mathcal{F} \rangle$ defines a joint probability distribution over its random

Notation	Definition
Σ	Set of labels
R	Set of references
S	Set of sets of references
r	Reference in R
s	A set in S representing a potential real-world entity
$r.x$	Random variable representing the reference's label
(r_1, r_2)	Edge in $R \times R$
$(r_1, r_2).x$	Random variable representing the edge's existence
$s.x, s.n$	Random variables representing the existence of an entity (used interchangeably in the contexts of PGD and PEG, respectively)
$s.l$	Random variable representing the entity's label
$(s_1, s_2).e$	Random variable representing the existence of edge between entities s_1 and s_2
$S.\boldsymbol{n}$	Shorthand for $s_1.n = n_1, \ldots, s_{\|S\|}.n = n_{\|S\|}$
$S^r = \{s_1, \ldots, s_k\}$	Subset of S that contains all sets that contain r, i.e., $\{s \in S \| r \in s\}$
v	Entity graph node
e	Entity graph edge
$v.n, v.l$	Entity graph random variables for node's existence, and node's label, respectively

Table 5.1: Notations used in Section 5.3

variables $\mathcal{V}$ via its set of factors $\mathcal{F}$. Each factor f is defined over a subset $\mathcal{V}_f$ of $\mathcal{V}$ and represents a dependency between those random variables. Given a complete joint assignment $\mathbf{v} \in Dom(\mathcal{V})$ to the variables in $\mathcal{V}$, the joint distribution is defined by $Pr(\mathbf{v}) = \frac{1}{\mathcal{Z}} \prod_{f \in \mathcal{F}} f(\mathbf{v}_f)$, where $\mathbf{v}_f$ denotes the assignments restricted to the arguments $\mathcal{V}_f$ of f and $\mathcal{Z} = \sum_{\mathbf{v}' \in Dom(\mathcal{V})} \prod_{f \in \mathcal{F}} f(\mathbf{v}'_f)$ is a normalization constant referred to as the *partition function*. The independencies in the distribution defined by a PGM are represented graphically in its *Markov network*, which contains one node for each random variable, and an edge between a pair of random variables if and only if the two variables co-occur in some factor. Each connected component in the Markov network corresponds to a part of the model that is *independent* from the rest. We can thus compute the normalized probability for each connected component

separately and multiply them together to obtain the full joint distribution.

As a first step towards our probabilistic model, we now introduce random variables for labels of references ($r.x$), existence of edges between pairs of references ($e.x$), and existence of an entity corresponding to a set of references ($s.x$). We further specify a probability distribution over each such random variable.

Definition 1. ***Probabilistic Graph Description:*** *A probabilistic graph description (PGD) is a tuple* $D = (R, S, \Sigma, P, m^{\Sigma}, m^{\{T,F\}})$*, where*

- R *is a set of references.*
- S *is a set of subsets of* R *including at least all singleton subsets.*
- Σ *is a set of labels.*
- P *is a set of probability distributions containing (1) for each* $r \in R$*, a probability distribution* $p^r(r.\boldsymbol{x})$ *over a random variable* $r.x$ *with values from* Σ*, (2) for each* $(r_1, r_2) \in R \times R$*, a probability distribution* $p^{(r_1,r_2)}((r_1, r_2).\boldsymbol{x})$ *over a random variable* $(r_1, r_2).x$ *with values from* $\{T, F\}$*, and (3) for each* $s \in S$*, a probability distribution* $p^s(s.\boldsymbol{x})$ *over a random variable* $s.x$ *with values from* $\{T, F\}$*.*
- *The merge functions* m^{Σ} *and* $m^{\{T,F\}}$ *transform a set of probability distributions over random variables with values in* Σ *and* $\{T, F\}$*, respectively, into a single such distribution.*

For example, in Figure 5.1(a), $R = \{r_1, \ldots, r_4\}$, $S = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_3, r_4\}\}$, $\Sigma = \{a, r, i\}$, P is the set of probability distributions over the reference labels, re-

lations, and identity uncertainties, and finally, both m^{Σ} and $m^{\{T,F\}}$ are probability distribution averaging functions.

A PGD thus specifies the set of observed references R together with their possible labels as well as probabilities for the existence of edges between two references. Each set in S corresponds to a potential entity and contains all references to that entity. The PGD specifies independent probability distributions for the existence of such entities. The merge functions are used to compute new probability distributions after merging two or more references into a single entity. Different merge functions are appropriate in different settings. The simplest merge function is *average*, where we simply average the input probability distributions (this is the merge function we use in our experimental evaluation). Another example of a merge function for $m^{T,F}$ is *disjunct*; where the output probability distribution is the disjunction of the input distributions.

In the next step of our model construction, the probabilistic entity graph combines these independent probability distributions into a graphical model that encodes the dependencies between entities induced by shared references and combines the distributions over labels and edges using the merge functions provided by the PGD.

Definition 2. ***Probabilistic Entity Graph:*** *For a given PGD D, the probabilistic entity graph (PEG) U is a graphical model with set of random variables $\mathcal{V} = \{s.n|s \in S\} \cup \{s.l|s \in S\} \cup \{e.e|e \in S \times S\}$ and set of factors $\mathcal{F}$ defined as follows. For each*

$r \in R$ with $S^r = \{s_1, \ldots, s_k\} = \{s \in S | r \in s\}$, $\mathcal{F}$ contains a node existence factor

$$f^N(s_1.n = v_1, \ldots, s_k.n = v_k) = \begin{cases} p^s(s_i.x = T) & \text{if } v_i = T \text{ and } v_j = F \text{ for all } j \neq i \\ 0 & \text{otherwise.} \end{cases}$$

For each $s \in S$, $\mathcal{F}$ contains a node label factor

$$Pr(s.\boldsymbol{l}) = \left[m^{\Sigma}(\{p^r | r \in s\})\right](s.\boldsymbol{l}) \tag{5.1}$$

For each $(s_1, s_2) \in S \times S$, $\mathcal{F}$ contains an edge existence factor

$$Pr((s_1, s_2).\boldsymbol{e}) = \left[m^{\{T,F\}}(\{p^{(r_1,r_2)} | r_i \in s_i\})\right]((s_1, s_2).\boldsymbol{e}) \tag{5.2}$$

Identity uncertainty is modeled by the node existence factors ($f^N(s_1.n = v_1, \ldots, s_k.n = v_k)$), which ensure that all assignments where two entity nodes share a reference have zero probability. The node label factors ($Pr(s.\boldsymbol{l})$) are probability distributions obtained by aggregating the label probability distributions of all references in the underlying set s via the node label merge function. In the same way, the edge existence factors ($Pr((s_1, s_2).\boldsymbol{e})$) are probability distributions obtained by aggregating the edge existence probability distributions of all pairs of references from the underlying sets via the edge existence merge function.

Exploiting Independence: Writing out the probability distribution defined by

the PEG, we have

$$Pr(S.\boldsymbol{n}, S.\boldsymbol{l}, (S \times S).\boldsymbol{e}) = \frac{1}{\mathcal{Z}} \cdot \prod_{r \in R} f^N(S^r.\boldsymbol{n}) \cdot \prod_{s \in S} Pr(s.\boldsymbol{l}) \cdot \prod_{(s_1,s_2) \in S \times S} Pr((s_1, s_2).\boldsymbol{e}) \tag{5.3}$$

We use shorthand notation for assignments to sets of random variables, e.g., $S.\boldsymbol{n}$ for $s_1.n = n_1, \ldots, s_{|S|}.n = n_{|S|}$. The partition function $\mathcal{Z}$ is the sum of the factor product over all variable assignments. As all node label and edge existence factors are probability distributions independent of all other factors, Equation 5.3 is equivalent to

$$Pr(S.\boldsymbol{n}, S.\boldsymbol{l}, (S \times S).\boldsymbol{e}) = Pr(S.\boldsymbol{n}) \cdot \prod_{s \in S} Pr(s.\boldsymbol{l}) \cdot \prod_{(s_1,s_2) \in S \times S} Pr((s_1, s_2).\boldsymbol{e}) \tag{5.4}$$

where $Pr(S.\boldsymbol{n})$ is the normalized product of all node existence factors, that is, the partition function $\mathcal{Z}_n$ is with respect to those factors only:

$$Pr(S.\boldsymbol{n}) = \frac{1}{\mathcal{Z}_n} \prod_{r \in R} f^N(S^r.\boldsymbol{n}) \tag{5.5}$$

It is often possible to further decompose this function, taking into account the independencies encoded in the Markov network. Let $\mathcal{C}(S.n)$ be the partitioning of the set of random variables $S.n$ induced by the connected components of the Markov network, that is, each element of $\mathcal{C}(S.n)$ contains all random variables participating

in one such component. We can then rewrite the above equation as

$$Pr(S.\boldsymbol{n}) = \prod_{S_i.n \in \mathcal{C}(S.n)} \frac{1}{\mathcal{Z}_{n_i}} \prod_{r \in R \wedge S^r \subseteq S_i} f^N(S^r.\boldsymbol{n})$$
$$= \prod_{S_i.n \in \mathcal{C}(S.n)} Pr(S_i.\boldsymbol{n}) \tag{5.6}$$

where the partition function $\mathcal{Z}_{n_i}$ normalizes over all assignments for random variables in $S_i.n$.

Distribution over Graphs. Clearly, not all assignments to random variables in the model above directly correspond to legal graphs. We now show how to obtain the final distribution over labeled graphs. The set of possible world graphs $PW(U)$ of a PEG U consists of those graphs $W = (V, E, l(.))$ where V is a set of entity nodes corresponding to reference sets from S (merged into a single entity), $E \subseteq V \times V$ is a set of edges between them, and the label function $l : V \to \Sigma$ labels these nodes with elements of Σ. Slightly abusing notation, we often identify a graph node $v \in V$ with the corresponding set of references $s \in S$, and use both notations interchangeably. This allows us to treat V as a subset of S and thus simplify notation. Each possible world graph W induces a partial value assignment $(S.\boldsymbol{n}^W, V.\boldsymbol{l}^W, (V \times V).\boldsymbol{e}^W)$ to the random variables in the graphical model as follows. For each $s \in V$, we have $s.n^W = T$, and for each $s \in S \setminus V$, we have $s.n^W = F$, that is, values of node existence variables mirror the (non-)existence of nodes in W. For each $s \in V$, we have $s.l^W = l(s)$, that is, for all existing nodes, values of node label random variables mirror the labels in W, and all other node label random

variables remain unassigned. For all $(s_1, s_2) \in E$, we have $(s_1, s_2).e^W = T$, and for all $(s_1, s_2) \in (V \times V) \setminus E$, we have $(s_1, s_2).e^W = F$, that is, for all pairs of existing nodes, edge existence variables mirror the (non-)existence of edges in the graph, and all other edge existence random variables remain unassigned. The probability of W is now obtained based on Equation 5.4 by marginalizing over all unassigned variables. As those all appear in independent factors only, we get

$$\begin{aligned} Pr((V, E, l(.))) =& Pr(S.\boldsymbol{n}^W) \cdot \prod_{v \in V} Pr(v.l = l(v)) \\ & \cdot \prod_{(s_1,s_2) \in E} Pr((s_1, s_2).e = T) \cdot \prod_{(s_1,s_2) \in (V \times V) \setminus E} Pr((s_1, s_2).e = F) \end{aligned} \tag{5.7}$$

As every full assignment to the variables in the graphical model contributes to exactly one graph's probability, this defines a probability distribution over possible world graphs.

5.4 Subgraph Pattern Matching

We now define the task of subgraph pattern matching over uncertain graphs. Our discussion assumes undirected graphs, but our approaches are equally applicable to directed graphs. We start by defining a match of a query Q in a graph G where there is no uncertainty, and we then define probabilistic subgraph pattern matching. A query graph $Q = (V_Q, E_Q)$ is a graph where each node $v \in V_Q$ is labeled with a label $l_Q(v) \in \Sigma$.

Definition 3. ***Match:*** *Given a labeled graph $G = (V_G, E_G, l_G(.))$ and a query graph $Q = (V_Q, E_Q, l_Q(.))$, a subgraph $M = (V_M, E_M)$ of G is a match of Q in G if and only if there is a bijective mapping $\psi : V_Q \rightarrow V_M$ such that (i) $\forall u \in V_Q : l_Q(u) = l_G(\psi(u))$ and (ii) $(\psi(u), \psi(v)) \in E_M$ if and only if $(u, v) \in E_Q$.*

Definition 4. ***Probabilistic Match:*** *Given a PEG U and a query graph Q, a graph M is a probabilistic match of Q in U if and only if M is a match of Q in at least one legal possible world graph G of U, that is, one where no two nodes share a reference. The probability of the match M is the sum of the probabilities of all possible world graphs of U where M is a match:*

$$Pr(M) = \sum_{G \in PW(U) \wedge M \subseteq G} Pr(G) \tag{5.8}$$

Definition 5. ***Probabilistic Subgraph Pattern Matching:*** *Given a PEG U, a query graph Q, and a probability threshold α, find all matches of Q in U whose probability $Pr(M)$ is greater than or equal to α.*

Naively, this problem could be solved by performing subgraph pattern matching over each possible world graph and for each match found, summing the probabilities of possible worlds it appears in. Clearly, this approach is computationally infeasible. In the remainder of this section, we show how to (a) find all matches by performing subgraph matching on a single graph only, and (b) calculate the probability of a given match directly, without need to explicitly consider all possible worlds it appears in. This provides the basis for the algorithms discussed in Section 5.5, which further speed up probabilistic subgraph pattern matching.

Finding Matches. For a given PEG U, let G_U be the graph that has a node for each $s \in S$, labeled with the set of labels $L(s)$ that are associated with s with non-zero probability, that is, $L(s) = \{l' | l' \in \Sigma \wedge Pr(s.l = l') > 0\}$, and an edge between two nodes s_1 and s_2 if and only if $Pr((s_1, s_2).e = T) > 0$. We generalize the notion of match to this case by requiring the query node label to be in the set of labels of the matched node. Clearly, if M is a match in a legal possible world of U, it is a match in G_U. However, while all matches M in G_U are a match in some possible world of U, this world might not be legal. This is the case if and only if the match includes two nodes that share a reference. We therefore further extend the matching procedure on G_U to not return matches where two nodes share a reference. This ensures that the matches on G_U are exactly the probabilistic matches on U. For the discussions to follow, we use the term probabilistic entity graph to denote G_U as well, as it is the structure that our algorithms operate on.

Calculating Probabilities. In Equation 5.8, the probability of a match M found on G_U is defined based on a set of possible world graphs, summing their probabilities as given by Equation 5.7. The graphs in this set are exactly those containing all nodes in V_M with correct labels as well as all edges in E_M, and arbitrary sets of additional nodes and edges. Thus, the probability of M can be rewritten as the

marginal

$$Pr(M) = Pr_n(M) \cdot Pr_{le}(M) \tag{5.9}$$

$$Pr_n(M) = Pr(V_M.n = T) \tag{5.10}$$

$$Pr_{le}(M) = \prod_{v \in V_M} Pr(v.l = l(v)) \cdot \prod_{e \in E_M} Pr(e.e = T) \tag{5.11}$$

where $Pr(V_M.n = T)$ is the corresponding marginal of $Pr(S.n)$ that sums out values of all node existence variables whose nodes are not part of M. In practice, as in Equation 5.6, we exploit independencies in the underlying graphical model to calculate this probability as a product of existence probabilities of smaller sets of nodes. Recall that $\mathcal{C}(S.n)$ partitions the set of node existence random variables $S.n$ based on the connected components of the Markov network. As each node in a match corresponds to one such random variable, we can use the same partitioning, restricted to the set of nodes V_M in the match, to calculate $Pr(V_M.n = T)$ as $\prod_{C.n \in \mathcal{C}(S.n)} Pr((V_M.n \cap C.n) = T)$. Note that $Pr_{le}(M)$ is *subgraph decomposable*, that is, for two disjoint subgraphs M_1 and M_2, $Pr_{le}(M_1) \times Pr_{le}(M_2) = Pr_{le}(M_1 \cup M_2)$, while this is not the case for $Pr_n(M)$.

5.5 Algorithms

The problem of probabilistic subgraph pattern matching with identity uncertainty is #P-complete. To increase efficiency, we propose a new *path-based* approach to find probabilistic matches of queries. Our approach decomposes the query into a

set of paths, finds matches of individual paths, and exploits probabilistic information to prune the space of possible matches.

By focusing on paths rather than nodes when finding candidate matches, we can better exploit probabilistic information for pruning. If we would consider the probabilities associated with the nodes only as the criteria for candidacy (as opposed to paths), the search space would end up being very large, because node probabilities tend to be much larger than the final query probability, leading the search space to contain many more false positives. On the other hand, a path-based approach has better pruning capabilities, especially when used in association with *path context information* and further reduction techniques as outlined in the following paragraphs.

In order to enable efficient and scalable online processing, we divide the work of answering probabilistic subgraph pattern matching queries into an offline and an online phase. The offline phase first precomputes entity-level probability information. Second, it builds a novel disk-based *context-aware path index* on the probabilistic entity graph, indexing not only all the paths in the PEG up to a given length, but also other context information that captures different properties of the path local neighborhoods (Section 5.5.1). The online phase answers the online user's query (Section 5.5.2). It first decomposes the query into paths and then constructs a search space over the paths in three steps, by 1) accessing the path index to find an initial set of path *candidates*, i.e., paths in the PEG that can potentially be a match for the paths in the decomposition, 2) employing context information to prune the sets of candidates for all query paths, and 3) reducing the search space

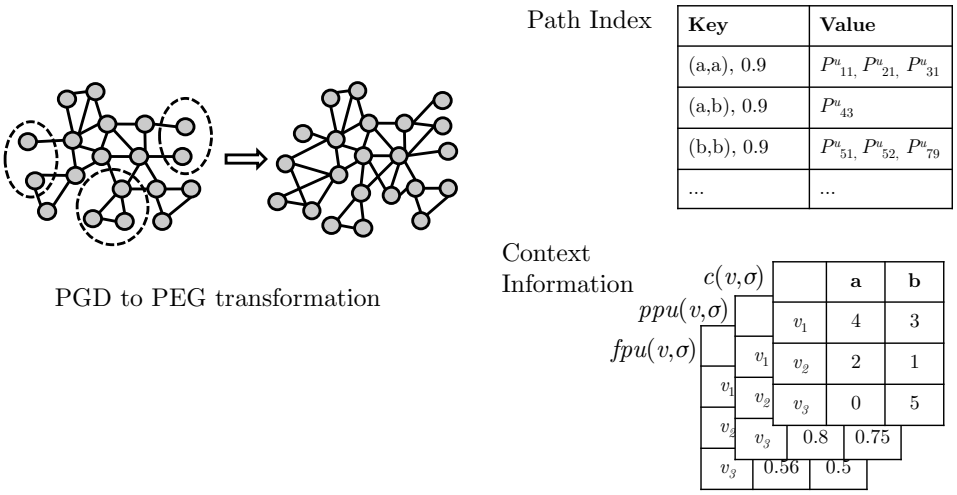


Figure 5.2: Schematic diagram of offline phase algorithms and indexes

for full graph matches using a technique called *reduction by join-candidates*, which performs message passing in a k-partite graph where each partition corresponds to a path in the query decomposition. This results in the final search space, from which the algorithm then generates the actual matches. Figures 5.2 and 5.3 summarize the offline and online phase algorithms, respectively. Notations used in this Section are listed in Table 4.2.

5.5.1 Offline Phase

The offline phase precomputes the following pieces of information over the probabilistic entity graph: component probabilities, path index, and context information on nodes. We discuss each type in the following subsections.

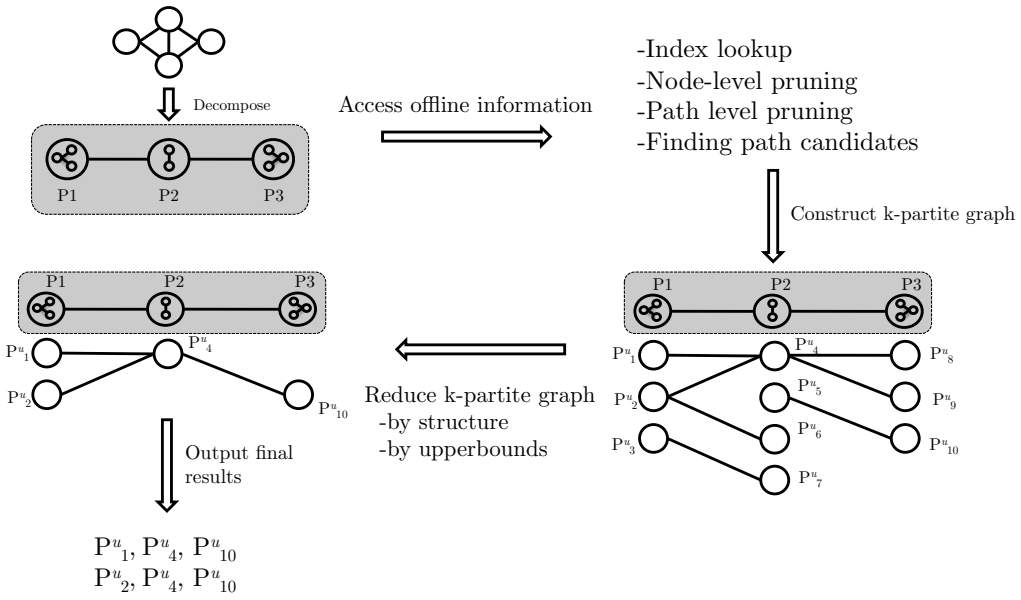


Figure 5.3: Schematic diagram of online phase algorithms

Notation	Definition
P	Query path
P^u	Entity graph path
$\psi(v)$	Entity graph node matching the query node v
$l(v)$	Label of node v
V_M	Set of nodes of subgraph M
E_M	Set of edges of subgraph M
$Pr_{le}(M)$	Probability of the node label and edge existence components of subgraph M
$Pr_n(M)$	Probability of the entity node existence component of subgraph M
α	Input query threshold
β	Path index construction threshold
γ	Path index resolution coefficient
L	Path index maximum path length
$\Gamma(v)$	Neighbors of node v
$refs(v)$	Set of underlying references of node v
$c(v, \sigma)$	Cardinality of node v with respect to σ
$ppu(v, \sigma)$	Partial probability upperbound of node v with respect to σ
$fpu(v, \sigma)$	Full probability upperbound of node v with respect to σ
$\mathcal{P}$	Set of paths in a path decomposition
$JP(P_1, P_2)$	Join predicates between P_1 and P_2
$J(P_1)$	Paths that join with P_1
$cn(P)$	Set of candidates of path P
$cn(P_1, P_1^u, P_2)$	Set of paths in $cn(P_2)$ that are candidates to be joined with $P_1^u \in cn(P_1)$

Table 5.2: Notations used in Section 5.5

5.5.1.1 Component Probabilities

Calculating match probabilities requires evaluating Equation 5.9. To reduce calls to the PGM engine during online inference, we precompute and store the underlying probabilities. As $Pr_{le}(.)$ is decomposable, we only precompute its parts, that is, node label and edge existence probabilities, by applying the corresponding merge functions on the underlying input distributions as specified in Equations 5.1 and 5.2, respectively. Since $Pr_n(.)$ is not decomposable, we precompute node existence marginals for all possible valid configurations of every connected component, i.e., those consisting of entities not sharing a reference. In general the connected components are expected to be small enough in practice for this to be feasible. If not, we could instead either employ an approximate inference technique to compute the marginals, or compute them on demand using the PGM engine.

5.5.1.2 Path Index

The path index contains all paths in the probabilistic entity graph that have length at most L, probability at least β[1], and do not contain two nodes sharing an underlying reference. Every entry in the path index consists of the following information:

- **Key:** the entry's key is a pair $\langle \mathbf{X}, \pi \rangle$, where $\mathbf{X} \in \Sigma^{l+1}$ is a sequence of node labels of length $l+1$, and $\pi \in \{\beta, \beta+\gamma, \beta+2\gamma, \ldots, 1\}$ is a probability value. The parameter γ defines the resolution of the index and provides a tradeoff

[1]Paths with smaller probability are computed on demand.

between the accuracy of the index and the time needed to access a range of probabilities.

- **Value:** the entry's value is the set of paths $\mathcal{P}^u$ of length l with probability under the node label assignment $\mathbf{X}$ between π and $\pi+\gamma$ satisfying the reference constraint. For every $P^u \in \mathcal{P}^u$, we store the path itself as well as its two probability components $Pr_{le}(P^u)$ and $Pr_n(P^u)$.

Example: If a path $P^u = (1, 2, 3)$ has the probabilities $Pr_{le}(P^u) = 0.9$ and $Pr_n(P^u) = 1.0$ under the node label assignment (x_1, x_2, x_3), then assuming an index resolution $\gamma = 0.1$, this path will be associated with the key $\langle(x_1, x_2, x_3), 0.9\rangle$.

To increase efficiency, we build a *two-level index*, where the first level, accessing $\mathbf{X}$ via equality predicates, is a hash index, and the second level, accessing π via range predicates, is a B+ tree index. Index construction starts with paths consisting of a single node ($l = 0$) and builds entries of length $l + 1$ based on those of length l, exploiting the fact that all paths with probability at least β must consist of sub-paths with probability at least β as well. We exploit the fact that entries for different label sequences of the same length can be constructed independently to build those *in parallel* using multiple threads. We use a synchronization barrier to ensure that all paths of the current length have been indexed before proceeding to the next length. To increase I/O performance, we accumulate a group of records in a *memory buffer* before writing the buffer to disk. Finally, for undirected graphs, entries for labels $\mathbf{X} = \{X_1, X_2, \ldots, X_{l-1}, X_l\}$ are identical to those for $\mathbf{X'} = \{X_l, X_{l-1}, \ldots, X_2, X_1\}$ because of *symmetry*, and we therefore only store one direction for each such case

and derive the other one as needed.

5.5.1.3 Context Information

When pruning the set of candidate matches for a path in Section 5.5.2.2, we rely on context information for nodes, which is the third type of information precomputed during the offline phase. For a node $v \in G_U$ and a label $\sigma \in \Sigma$, let $N(v, \sigma)$ be the set of neighbors of v that have σ in their set of possible labels, i.e.,

$$N(v, \sigma) = \{v' | v' \in \Gamma(v), \sigma \in L(v), refs(v) \cap refs(v') = \emptyset\},$$

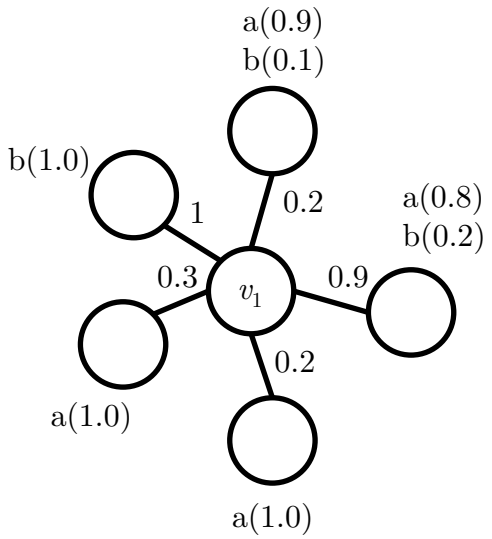
where $\Gamma(v)$ is the set of neighbors of node v, and $refs(v)$ is the set of underlying references of node v. For each node $v \in V_U$ and label $\sigma \in \Sigma$, we compute the following values:

- **Cardinality** $c(v, \sigma)$, which is simply the size of $N(v, \sigma)$.

$$c(v, \sigma) = |N(v, \sigma)|$$

- **Partial Probability Upperbound** $ppu(v, \sigma)$, which is an upperbound for the probabilities in the neighborhood of v considering only the edges between v and $N(v, \sigma)$.

$$ppu(v, \sigma) = max_{v' \in N(v,\sigma)} Pr((v, v').e = T)$$



v	σ	$c(v,\sigma)$	$ppu(v,\sigma)$	$fpu(v,\sigma)$
v_1	a	4	0.9	0.8
v_1	b	3	1.0	1.0

Figure 5.4: Context information example

- **Full Probability Upperbound** $fpu(v,\sigma)$, which is an upperbound for the probabilities in the neighborhood of v also taking into account the neighbors' labels.

$$fpu(v,\sigma) = max_{v' \in N(v,\sigma)} Pr(v'.l = \sigma) \cdot Pr((v,v').e = T)$$

These measures capture different aspects of node/path neighborhoods. During the online phase (cf. Section 5.5.2.2), we use a combination of the cardinality and full probability upperbound to prune path candidates at the individual node level, and a combination of full and partial probability upperbounds to prune path candidates at the entire path level.

Example: In Figure 5.4, $c(v_1, a) = 4, c(v_1, b) = 3$. $ppu(v_1, a) = 0.9$ because the highest edge probability that connects v_1 to a node with label a is 0.9. Similarly, $ppu(v_1, b) = 1.0$. Finally, $fpu(v_1, a) = 0.72$ because it has an edge with existence probability of 0.9 connecting it to a node with probability of 0.8 for the label a. Similarly, $fpu(v_1, b) = 1.0$.

5.5.2 Online Phase

Our online query processing technique consists of five main steps: decomposing the query into a set of paths, obtaining a set of candidates for every path in the decomposition, obtaining *join-candidate paths* for every candidate path, which are candidate paths whose query paths share a node with the given candidate and can thus extend it to form a partial match, jointly reducing the candidate search space by *reduction by join-candidates*, and finally finding matches to the full query. A schematic diagram of the online phase steps is shown in Figure 5.3. Below we discuss each step in detail.

5.5.2.1 Path Decomposition

The task of *path decomposition* is to split the query into a set of possibly overlapping paths, each of length L or less, that cover the entire query, and whose matches can be obtained from the path index. To preserve the structural information of the query, intersection points between the paths are expressed as join predicates, which have to be satisfied when combining path matches into a full query match.

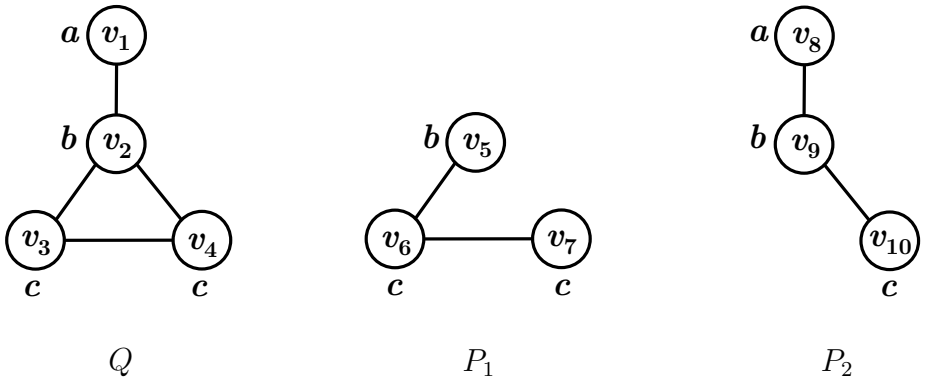


Figure 5.5: A query Q and its decomposition into two paths P_1 and P_2 that cover Q. Letters inside node represent node IDs, and letters outside nodes represent the node labels. The predicates associated with the path decomposition are $P_1.v_5 = P_2.v_9$ and $P_1.v_7 = P_2.v_{10}$.

For example, Figure 5.5 shows a query Q and its decomposition into two paths P_1 and P_2. In order to preserve the structural information of Q, any pair (P_1^u, P_2^u) that matches (P_1, P_2) must satisfy the predicates $P_1^u.v_5 = P_2^u.v_9$ and $P_1^u.v_7 = P_2^u.v_{10}$ (we use $P_1^u.v_5$ to denote the vertex in path P_1^u that matches the vertex v_5 in path P_1 in the query). Query path decomposition thus decomposes a query Q into a set of node/edge overlapping paths $\mathcal{P}$. For every pair of overlapping paths P_1 and P_2, the decomposition defines a set of *join predicates* $JP(P_1, P_2)$. Further, we denote the set of paths joining with a path P by $J(P)$.

Since a single query has multiple valid path decompositions, and each decomposition may lead to a different query processing cost, we would like to find a least-cost path decomposition. Ideally, the cost of a decomposition should express the number of operations involved in order to produce the final query results. As the intricacy of the algorithm makes it difficult to calculate such a number, we instead use an estimate of the initial query search space size SS_0. We would thus like to

find $argmin_{\mathcal{P} \subseteq \mathbb{P}(Q), \mathcal{P} \text{ covers } Q} SS_0(\mathcal{P})$, where $\mathbb{P}(Q)$ is the set of all possible paths of length at most L in Q. More specifically, for each path P in the decomposition, we estimate the number of matches, or its cardinality $C(P, \alpha)$, as discussed below. We then estimate the search space size as the product of all such path cardinalities. The cardinality is based on the number of database paths matching the query path P with probability at least α, but also takes into account the fact that those matches will have to be extended to neighboring query paths. We therefore express $C(P, \alpha)$ in terms of the following quantities.

1. The **number of candidates from the path index** $|PIndex(l_Q(V_P), \alpha)|$ is the number of paths matching P's label sequence $l_Q(V_P)$ with probability at least α as found in the path index under $PIndex(l_Q(V_P), \alpha)$.

2. The **path degree** $degree(P)$ is the sum of path node degrees, not counting edges on the path, that is, $degree(P) = \sum_{n \in V_P} degree(n) - 2 \times length(P)$.

3. The **path density** $density(P)$ measures how close the nodes on P are to forming a clique. Let K be the number of edges between the nodes of P, and M the number of nodes on the path, then $density(P) = \frac{2K}{M(M-1)}$.

Example: The path degree of path $(1, 2, 3, 4)$ shown in Figure 5.6 is 5, and its density is 4/6.

Taking into account the direction of influence of these components on the true number of matches, we approximate the cardinality of P as:

$$C(P, \alpha) \propto \frac{|PIndex(l_Q(V_P), \alpha)|}{degree(P) \cdot density(P)}$$

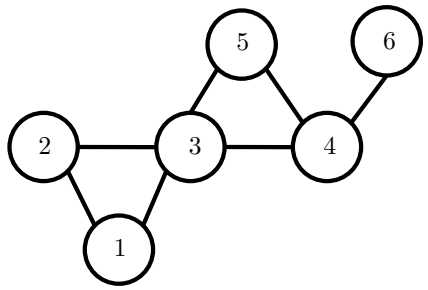


Figure 5.6: Path degree and density example

Therefore, our goal is to find

$$argmin_{\substack{\mathcal{P}\subseteq\mathbb{P}(Q),\\ \mathcal{P}\text{ covers }Q}} \prod_{P\in\mathcal{P}} \frac{|PIndex(l_Q(V_P),\alpha)|}{degree(P)\cdot density(P)}$$

Since it is not practical to query the index for an arbitrary α and $l_Q(V_P)$ at query runtime, we build a histogram for every possible label sequence $\mathbf{X}$ during the offline phase at selected probability points $(\alpha_0,\ldots,1)$. At runtime, we use exponential curve fitting to estimate the value of $|Index(l_Q(V_P),\alpha)|$ given $histogram(l_Q(V_P),\alpha_i)$ and $histogram(l_Q(V_P),\alpha_{i+1})$ where $\alpha_i<\alpha<\alpha_{i+1}$.

We reduce the problem of optimizing the cost function to that of SET COVER, where the set of query edges corresponds to the universal set (in the corresponding SET COVER instance), and each path P in the query with length at most L is a possible set in the cover. Note that we allow paths with shared edges, as this can reduce the cost of several paths at once (e.g., in the case of a very selective edge connected to multiple non-selective paths). The cost of the cover is the product of the individual costs of the participating paths. Since SET COVER is NP-complete, we use the standard greedy approximation to solve the problem, which calculates

an *efficiency* metric for every path, by dividing its length by its costs, and then greedily adding the path with the highest efficiency to the cover, and continuing iteratively until all the query edges are covered.

5.5.2.2 Finding Path Candidates

Given a path decomposition $\mathcal{P}$, the next step is to find candidate matches for every query path. Therefore, at a high level, for every path $P \in \mathcal{P}$, we access the path index to get its matches $PIndex(l_Q(V_P), \alpha)$, by only keeping those paths that satisfy certain context criteria. We denote the resulting set of matches by $cn(P)$ ($\subseteq PIndex(l_Q(V_P), \alpha)$). This second step relies on the following query statistics:

- **Node-level statistics:** For every node $n \in V_Q$, we calculate its neighborhood label count for every label $\sigma \in \Sigma$,

$$c(n, \sigma) = |\{m | m \in \Gamma(n), l_Q(m) = \sigma\}|$$

- **Path-level statistics:** For every path $P \in \mathcal{P}$, we collect information on its neighboring nodes in the query, the nodes on P these neighbors are connected to, and the query edges outside P that connect nodes on P. In order for a path match to be a candidate for contributing to a full query match, it has to be possible to extend this match to at least this neighborhood, and we can safely prune other path matches. More specifically, we use the following information:

  1. Path neighbors $\Gamma(P)$: the set of nodes that are not on P but are neighbors

of at least one node on P.

2. Reverse path neighbors: for every $m \in \Gamma(P)$, $rv(P, m)$ is the set of nodes on P that are neighbors of m.

3. Path cycles: for every $n \in V_P$, path cycles, $cyc(P, n)$, is the set of nodes on P that are also connected to n by a query edge outside the path, and thus appear together with n in a cycle. To avoid information duplication, each such edge only contributes to the path cycles of one of its endpoints.

Example: In Figure 5.6, path neighbors of the path $(1, 2, 3, 4)$ are the set of nodes $\{5, 6\}$. Reverse path neighbors of node 5 are $\{3, 4\}$. There is a path cycle formed by the edge between the nodes 1 and 3.

Node-level pruning Using the node-level statistics, we calculate a set of candidates $cn(n)$ for every node $n \in V_Q$ based on the following two criteria:

1. For every label $\sigma \in \Sigma$, v must have a number of neighbors that is greater than or equal to the number of neighbors of n with label σ, i.e., $c(v, \sigma) \geq c(n, \sigma), \forall \sigma \in \Sigma$.

2. For every label $\sigma \in \Sigma$, the probability of v having the correct label and at least the number of neighbors labeled σ required by the query has to exceed the query threshold α. Using precomputed full probability upperbounds as approximation and taking into account multiple occurrences of the same label, we therefore further restrict candidates v for n to those satisfying $Pr(v.l = l_Q(n)) \times fpu(v, \sigma)^{c(v,\sigma)} \geq \alpha, \forall \sigma \in \Sigma$.

Path-level pruning Next, we prune the set of candidate paths based on path-level statistics. For each path $P^u \in PIndex(l_Q(V_P), \alpha)$, we perform the following tests:

1. For every node $v \in V_{P^u}$, v must be a candidate for the corresponding node n in P, i.e., $v \in cn(n)$.

2. The probability of a path together with its neighboring nodes and cycles must be greater than or equal to α, which we test using $(Pr_{le}(P^u) \times Pr_n(P^u)) \times pu(P^u) \times cpr(P^u) \geq \alpha$, with $pu(P^u)$ and $cpr(P^u)$ defined as follows.

   The *path-neighborhood probability upperbound* $pu(P^u)$ of a candidate path P^u matching a query path P is an upperbound for the probability of all nodes matching $\Gamma(P)$ and their edges. Let $m \in \Gamma(P)$ be a path P neighbor, and n a node on P such that $n \in rv(P, m)$. We form a probability upperbound $pu(n, m, P^u)$ on the neighborhood of m as follows:

$$pu(n, m, P^u) = fpu(\psi(n), l_Q(m)) \prod_{n' \in rv(P,m), n' \neq n} ppu(\psi(n'), l_Q(m))$$

   where we use the full probability upperbound fpu for the edge between the match of m and the selected neighbor n, and partial probability upperbounds for all other neighbors of m's match, thus ensuring that information on m is only considered once. Since this upperbound may depend on the selected n, we obtain the tightest upperbound

$$pu(m, P^u) = min_{n \in rv(P,m)} pu(n, m, P^u)$$

Finally, to find the overall path P^u neighborhood probability upperbound, we aggregate over all $m \in \Gamma(P)$:

$$pu(P^u) = \prod_{m \in \Gamma(P)} pu(m, P^u)$$

The *path-cycles probability* $cpr(P^u)$ is the overall probability of edges not on the path P^u but connecting path nodes:

$$cpr(P^u) = \prod_{\substack{n \in V_P, \\ m \in cyc(P,n)}} Pr((\psi(n), \psi(m)).e = T)$$

Finally, for every path P in the decomposition, we obtain the list of candidates $cn(P)$ that contains exactly those paths from the initial set $PIndex(l_Q(V_P), \alpha)$ that pass the above tests.

5.5.2.3 Finding Join-Candidates

In this step, we find for every candidate path $P^u \in cn(P)$ of every query path P a set of paths that are candidates to be joined with P^u.

Recall that every query path $P_1 \in \mathcal{P}$ can be joined with a set of paths $J(P_1) \subseteq \mathcal{P}$, and there is a set of join predicates $JP(P_1, P_2)$ between P_1 and every path $P_2 \in J(P_1)$. For a query path $P_1 \in \mathcal{P}$, and a candidate path $P_1^u \in cn(P_1)$, we define

its join-candidate paths of type $P_2 \in J(P_1)$ as:

$$cn(P_1, P_1^u, P_2) = \{P_2^u | P_2^u \in cn(P_2) \wedge jp(P_1^u, P_2^u) = T, \forall jp \in JP(P_1, P_2)$$
$$\wedge Pr(P_1^u \circ P_2^u) \geq \alpha \ \wedge refs(V_{P_1^u}) \cap refs(V_{P_2^u}) = \emptyset\}$$

where $jp(P_1^u, P_2^u)$ is the instantiation of the predicate $jp \in JP(P_1, P_2)$ using paths P_1^u and P_2^u, and $P_1^u \circ P_2^u$ is the subgraph consisting of the two joined paths. Intuitively, $cn(P_1, P_1^u, P_2)$ refers to the set of paths in $cn(P_2)$ that are candidates to be joined with $P_1^u \in cn(P_1)$.

To facilitate finding join-candidate paths, for each $P \in \mathcal{P}$, while finding $cn(P)$, we build a lookup table $T(P, P_i)$ for each query path $P_i \in J(P)$. For every table $T(P, P_i)$, the set of positions $\langle p_{i1}, \ldots, p_{ik} \rangle$ indicates the nodes in P_i that participate in join predicates. The key for table $T(P, P_i)$ is a set of nodes $\langle n_1, \ldots, n_k \rangle$, and the values are paths in $cn(P)$ that have nodes $\langle n_1, \ldots, n_k \rangle$ at positions $\langle p_{i1}, \ldots, p_{ik} \rangle$. Given a path $P_i^u \in cn(P_i)$, paths in P which are joinable with P_i^u can now be obtained using a direct lookup operation from table $T(P, P_i)$, where the access key is obtained from P_i^u.

5.5.2.4 Joint Search Space Reduction

Joint search space reduction exploits the mutual relationship between the candidates and their join-candidates to reduce the size of all candidate lists before constructing full query matches, based on the following two observations. First, for a candidate match of a path P to contribute to a full query match, we need to be able

to combine it with at least one candidate for all query paths joining P. Second, if we can obtain an upperbound on the probability of all full query matches a candidate path can appear in, we can prune candidate paths based on the query threshold α. We refer to these two principles as *reduction by structure* and *reduction by upperbounds*, respectively, and discuss their details below. As they mutually influence each other, the overall algorithm for joint search space reduction iterates between them until no further changes occur.

We implement the reduction algorithm based on a *k-partite graph*, where each partition corresponds to a query path, each vertex to a candidate path match, and each link to a join between two candidate paths.[2] Pruning a candidate thus corresponds to deleting a vertex and its outgoing links from the k-partite graph.

Definition 6. ***Candidate k-partite Graph*** *A candidate k-partite graph is a k-partite graph that has a partition for each $P \in \mathcal{P}$, where the set of vertices of each partition P are $cn(P)$. There is a link between P_1^u in partition P_1 and P_2^u in partition P_2 iff $P_2^u \in cn(P_1, P_1^u, P_2)$ (of course, $P_2^u \in cn(P_1, P_1^u, P_2) \iff P_1^u \in cn(P_2, P_2^u, P_1)$).*

Every match of the query in the PEG corresponds to a subgraph of the candidate k-partite graph with one vertex per partition (i.e., one match for each query path) and all join links between them. We can thus safely prune all vertices that have no links to a partition they should link to, as well as those that cannot participate in any match with probability above the query threshold.

[2]To avoid confusion, we use the terms (vertex/link) when referring to the k-partite graph, and (node/edge) when referring to the PEG.

Reduction by structure. Reduction by structure removes vertices from the candidate k-partite graph by iterating the following two steps: 1) If a vertex has no links to at least one partition its query path joins with, remove the vertex and all of its links to vertices in all partitions, and 2) repeat (1) until no further changes take place.

Reduction by upperbounds. In order to exploit probabilistic information during search space reduction, we now introduce two types of vertex weights, based on $Pr_{le}(.)$ and $Pr_n(.)$, respectively, and then discuss a message passing scheme that exploits these weights to obtain bounds for reduction by upperbounds.

The first type of weights is assigned such that when a subgraph's weights are multiplied, we obtain the final $Pr_{le}(.)$ probability of the corresponding match. To avoid double contributions in cases of overlap between paths, we assign the overlapping elements' probability to exactly one partition, i.e., for every $v \in V_Q, e \in E_Q$, we choose exactly one partition to cover v's or e's probability. That is, if v or e exclusively belongs to one query path, it is assigned to the partition representing that path, and if v or e appear on multiple query paths, only one of their partitions is picked. Let partition P (we use P to refer to both the path and its corresponding partition) exclusively cover nodes and edges $cv(P)$ and $ce(P)$, respectively, then a vertex's first weight is

$$w_1(P^u) = \prod_{n \in cv(P)} Pr(\psi(n).l = l_Q(n)) \prod_{e \in ce(P)} Pr(\psi(e).e = T)$$

where $\psi(n)$ is the PEG node matching the query node n. As identity probabilities $Pr_n(P^u)$ are not decomposable, we directly use the identity probability of a path as the second weight of its corresponding vertex in the k-partite graph (however, we cannot multiply weights of this type together as it is the case with w_1 weights):

$$w_2(P^u) = Pr_n(P^u)$$

In addition to the two weights, each vertex P^u has an associated *perception vector* of length k, that is, with one entry per partition. Each entry is an upperbound on the w_1 weights of all vertices in that partition that can appear in a full match with P^u. Initially, we have $w_1(P^u)$ for the entry corresponding to P^u's own partition, and 1 for all other partitions. During message passing, each vertex first sends its current vector to each of its neighboring vertices (excluding the entry for the receiving neighbor's partition). Once all messages are received, each vertex P_1^u updates its own vector based on the values received from its neighbors as follows. For each vector entry corresponding to a partition P and each partition P_2 containing neighboring nodes of P_1^u, we choose the maximum value for P sent by the neighbors in P_2. We then take the minimum of these over all such P_2 as the new value in the vector, and iterate the overall process. The upperbound used to prune a vertex (and thus a candidate path) based on the query threshold α then is the product of all entries in the vertex' vector and its weight w_2.

As discussed above, the final algorithm iterates between both types of reduction until no further changes take place. We further improve efficiency by avoiding

unnecessary updates and exploiting parallelism, as discussed next.

Incremental maintenance. We only recompute upperbounds for vertices for which a neighbor has been deleted or has reduced its perception, and only consider vertices connected to a newly deleted link for deletion.

Parallel Implementation. We develop a shared-memory parallel implementation for the reduction algorithm, with one thread per partition. We introduce appropriate locking protocols to avoid incorrect modifications of the k-partite graph by multiple threads at the same time. We note here that in addition to the parallel implementation of the reduction algorithm, we also exploit parallelism in other parts of the system such as constructing node candidates, path candidates, and building join-candidate sets.

Example: Figure 5.7(a) shows an example of a query that is decomposed to three paths, where P_2 joins with P_1 and P_3. In Figure 5.7(b), we show an example of the k-partite graph construction, by introducing links between pairs of path matches that satisfy the join conditions. Once the k-partite graph is constructed, it can be reduced by removing vertices that do not have any links to a partition that it should join with. Therefore, $P_3^u, P_5^u, P_6^u, P_7^u, P_{10}^u$ can be removed with all their links, resulting in the k-partite graph in Figure 5.7(c). We can further apply reduction by upperbounds as shown in Figures 5.7(d), (e), (f). In Figure 5.7(d), each vertex is initialized by a partition perception vector that is all 1's except for the position of its own partition, which is initialized by the vertex's own weight. In this exam-

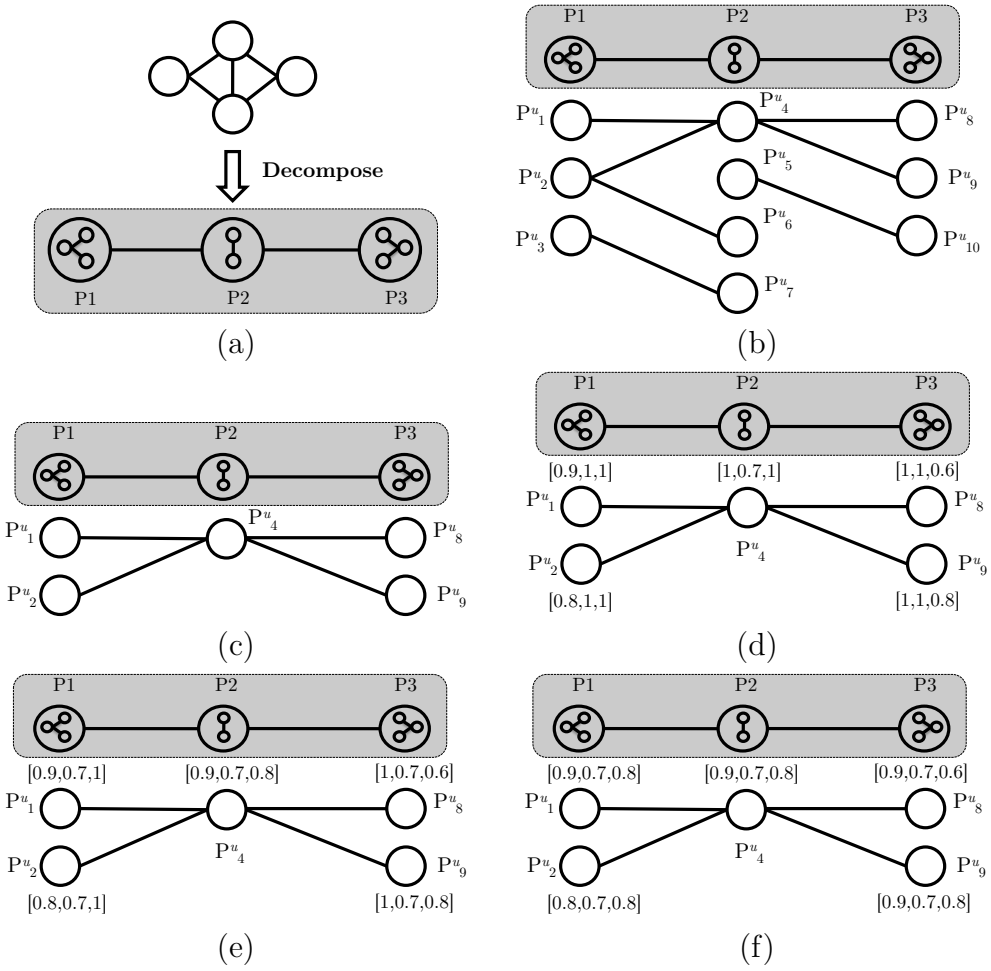


Figure 5.7: (a) An example query and its decomposition, (b) k-partite graph construction, (c) reduction by structure, (d), (e), (f), reduction by upperbounds

ple, we consider weights of type w_1 only for simplicity. In the second step, each vertex updates its upperbounds based on values from its neighbors, leading to the perception vectors in Figure 5.7(e). Figure 5.7(f) depicts the result of applying another iteration of the reduction algorithm, by performing one more pass of message exchange. Assuming that the input query probability threshold $\alpha = 0.4$, we can see that P_8^u can be removed from the graph along with its links. At this point, no further changes to the k-partite graph can take place, and we can proceed to the

final result generation step.

5.5.2.5 Finding Full Query Matches

The final step of the online query processing algorithm is finding the full query matches. The algorithm starts from the matches of one path and progressively adds matches of joining paths, based on an initially determined join order.

Join order determination. In principle, the optimal join order could be determined by minimizing the size of the intermediate results, that is, the sum of the numbers of candidates after each step. To avoid the extra burden of this step, we add paths to the join order one at a time, based on the following heuristic:

1. Choose the path with the largest number of nodes overlapping with the paths that already exist in the order.

2. In case of ties, choose the path with the largest number of join predicates with the existing paths.

3. In case of ties, choose the path with smallest cardinality (estimated as in path decomposition).

In general, a node on the new path can participate in multiple join predicates with existing nodes. However, when choosing the first path in the order, the first two criteria are equal for all the paths, and we just use the third one.

Finding matches. Given the join order $\{P_1, \ldots, P_{|\mathcal{P}|}\}$, we use the reduced candidate k-partite graph to construct matches incrementally. The initial set of matches

are the vertices in the partition corresponding to P_1. Each match M_i up to path P_i is extended to matches up to P_{i+1} as follows. We first identify all paths P_j with $j \leq i$ that join with P_{i+1}. For each vertex in P_{i+1}'s partition that has a link to the corresponding vertex in M_i for each such P_j, we extend M_i to a match up to P_{i+1} by adding that vertex's candidate match. We discard M_i if there is no such vertex, and only produce those extended matches that have probability at least α and do not contain two nodes sharing a reference.

5.6 Experimental Evaluation

In this section, we present the results of a comprehensive experimental evaluation using our prototype implementation. Our implementation is written in Java and uses the disk-based graph database engine Neo4j for storing the probabilistic graph, and the key/value store KyotoCabinet to store the index as a B+ tree. We begin by presenting the index construction algorithm's performance in terms of both time and space, and then demonstrate online query performance by comparing it to various baselines. We further study the effect of the different pruning methods we proposed on reducing the search space, and the relationship between the search space size and different parameters. Finally, we report results on a real-world dataset from DBLP. For the first set of experiments, we use synthetic graphs whose structure is generated according to the preferential attachment model [53]. To generate node label probabilities, we first generate a set of random probabilities $p_1, \ldots, p_{|\Sigma|}$, which we then weigh by a zipf distribution, i.e., $p'_i = \frac{p_i}{i}$, to introduce skew. We

normalize those to obtain final probabilities $p_i'' = \frac{p_i'}{\sum_j p_j'}$, which are assigned to node labels randomly. Edge probabilities are generated analogously. To generate reference sets corresponding to entities, we randomly choose k subsets of nodes from the graph, each of size s nodes, and randomly assign r pairs of nodes per group to the same reference set. That is, reference sets are of size 2, and the maximum size of a connected component is s. Probabilities of reference sets are generated randomly. We use merge functions that average the underlying distributions for both node attributes and edge existence. In our experiments, we use four settings with 50k, 100k, 500k, and 1m references, and a number of relations equal to 5× the number of references in every setting. We set $k =$ No. of references/1000, $s = r = 4$. We associate probability distributions with 20% of the references, relations, and reference sets unless otherwise stated. These settings result in probabilistic entity graphs of sizes (54k/292k), (108k/583k), (540k/ 2.95m) and (1.08m/5.88m) nodes/edges, respectively. Synthetic experiments are performed on an Amazon EC2 instance with a Linux operating system, 8 core processors, 117 GB of RAM and 2 TB of instance storage. The realworld experiment is performed on a Linux machine with two 2.66 GHz quad-core processors with hyper-threading, 48 GB of RAM, and a 1TB 7200 RPM disk drive.

5.6.1 Offline Phase Performance

We first assess performance of the offline phase for different maximum path lengths.

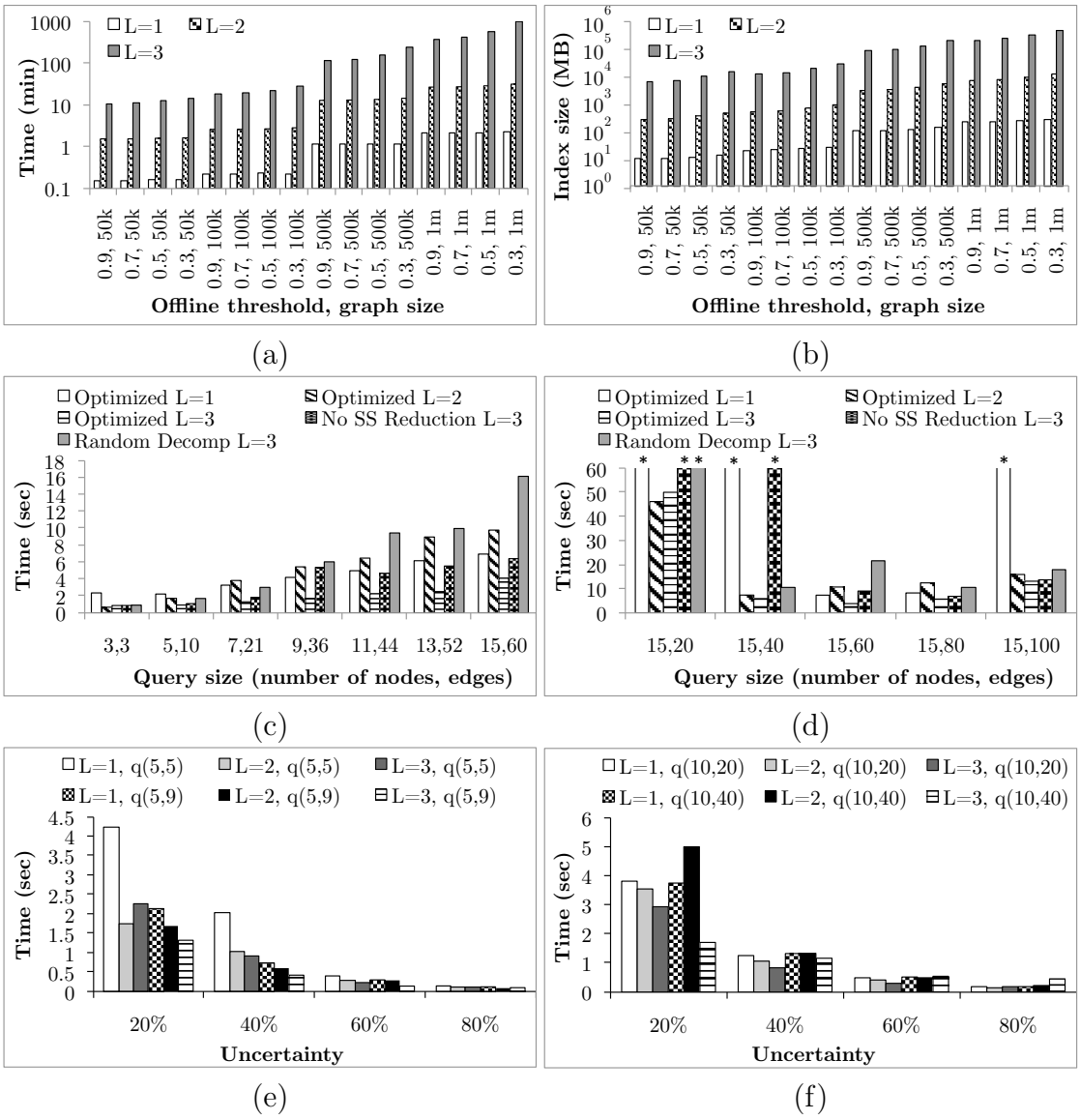


Figure 5.8: (a),(b) Offline phase performance, (c) varying query size, (d) varying query density, (e), (f), varying degree of uncertainty for queries with 5 and 10 nodes, respectively. A * above a bar indicates that the query did not finish in the allocated time (15 minutes), or the process ran out of memory.

5.6.1.1 Running Time

We first study the running time performance of the entire offline phase, which includes calculating the entity graph component probabilities, building the path index, and calculating context information. Figure 5.8(a) shows the running time

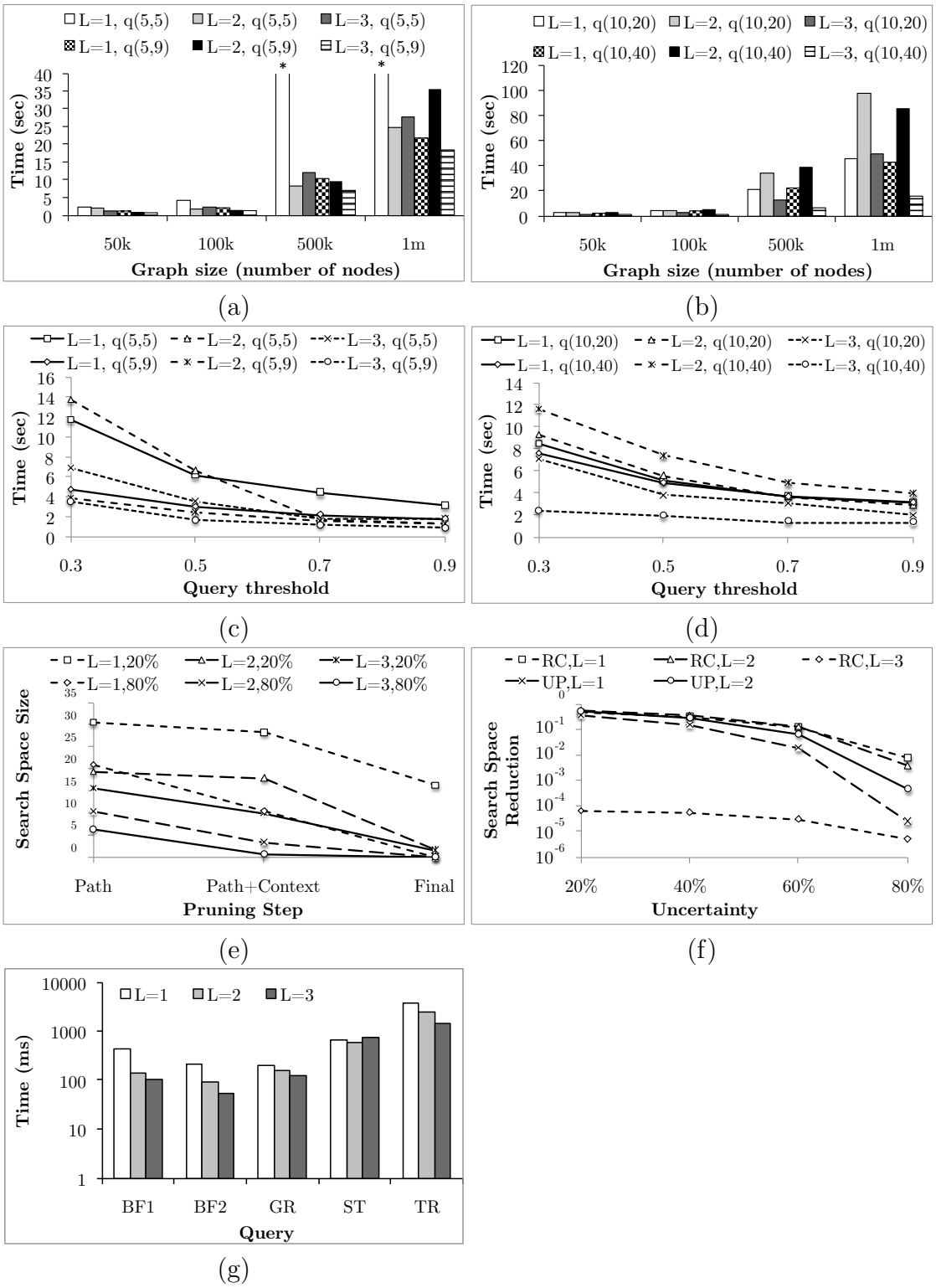


Figure 5.9: (a),(b) Varying input graph size for queries with 5 and 10 nodes, respectively, (c), (d) varying input query threshold for queries with 5 and 10 nodes, respectively, (e),(f) search space experiments, (g) performance on real-world data. A * above a bar indicates that the query did not finish in the allocated time (15 minutes), or the process ran out of memory.

for index path lengths of $L = 1, 2$, and 3. We vary both the graph size and the index lowerbound probability threshold, β, on the Y axis. As we can see, the offline phase running time at $L = 2$ is between 10 and 14 times that at $L = 1$, and at $L = 3$ it is between 7 to 30 times that of $L = 2$. Also, as the graph size increases, running time increases by a factor less than the graph size increase factor. For example, although the 1m graph is 20 times larger than the 50k graph, the running time increases by a factor of 14 on average at $L = 1$, 18 on average at $L = 2$, and 46 on average at $L = 3$. This is due to higher memory buffer utilization for larger graphs.

5.6.1.2 Path Index Size

We next compare the path index size at $L = 1, 2$, and 3, varying the graph size and index threshold as before. Results in Figure 5.8(b) show that index sizes at $L = 2$ are 32 times larger than those at $L = 1$ on average, and index sizes at $L = 3$ are 28 times larger than those at $L = 2$ on average. Index size increases at the same rate as the graph size at $L = 1$, and faster than the increase in the graph size at $L = 2$, e.g., the index size at 1m is 20 times larger than that of 50k on average at $L = 1$ and 25 times on average at $L = 2$. This is because indexes at $L = 1$ increase linearly with graph size, while at $L = 2$ the index size increases quadratically. The same trend applies at $L = 3$ as its size increases cubically.

5.6.2 Online Phase Performance

We now study different performance aspects of the online phase.

5.6.2.1 Online running time

We first compare the running time of our proposed algorithm to a range of baselines, using different input query sizes. We use the following algorithms and parameters:

1. **Optimized:** This refers to our proposed approach with all the proposed optimizations. For this method we use different path lengths: $L = 1$, $L = 2$ and $L = 3$.

2. **Random decomposition:** This is a variant of our proposed approach that does not employ the proposed query decomposition algorithm. In this baseline, we use random query decomposition instead of SET COVER, and when determining the path join order, we sort the paths according to their number of path index matches only, without taking into account the number of node intersections, number of predicates, path degree or path density. We set $L = 3$ for this baseline.

3. **No search space reduction:** This approach uses our optimized method, but without the joint search space reduction using the k-partite graph representation, and goes directly to generating final results after constructing the candidate and relative candidate lists. We set $L = 3$ for this baseline.

4. **SQL:** We implement our queries using SQL and run them on top of MySQL database. We run SQL on the 100k nodes dataset using a query with 5 nodes and 7 edges and a query threshold of 0.7. While our approach can answer this

query in less than a second, SQL never finishes it in a month. Therefore, we do not report any other SQL-based performance metrics.

Varying input query size. In this experiment, we study the running time performance of Optimized ($L = 1, 2, 3$), Random Decomp and No SS Reduction for varying query size. We use the 100k dataset and a query threshold of 0.7. Figure 5.8(c) shows running times for 7 different query sizes between (3,3) and (15,6) nodes/edges, averaged over five randomly generated queries per size. A query of n nodes has $4 \times n$ edges, unless the maximum number of edges for the query is less than $4 \times n$, in which case, we use the maximum possible number of edges. Our approach at $L = 3$ always outperforms $L = 1, 2$ and both of Random Decomp and No SS Reduction. For smaller queries (with 3 and 5 nodes), $L = 2$ outperforms $L = 1$, but it does not for the larger ones. The reason is that $L = 1$ has an advantage with querying the path index, as it returns a lower number of matches than both $L = 2, 3$, and at the same time, $L = 3$ has an advantage with context-based pruning, as higher path lengths have richer context information. At $L = 2$ the pruning performed with context information does not alleviate the processing needed for the larger number of matches returned from the path index, especially with larger query sizes. However, as we show in further experiments, $L = 2$ outperforms $L = 1$ when the input graph has higher degree of uncertainty, even for larger query sizes, and also sometimes outperforms $L = 1$ in extreme cases, such as queries with a very large number of results, or with a very large number of nodes and edges, or very large input graphs (e.g., (500k, 2.5m) and (1m, 5m) nodes, edges). Therefore, even

though $L = 2$ sometimes does not perform as well as $L = 1, 3$, it may be used as a compromise that does not take as much time and space as $L = 3$ in building its index, and still has an acceptable performance in extreme cases where $L = 1$ may not succeed.

Varying input query density. In this experiment, we study the running time performance of Optimized ($L = 1, 2, 3$), Random Decomp and No SS Reduction for varying the input query density. We use the 100k dataset and a query threshold of 0.7. Figure 5.8(d) shows running times for 5 different densities, by using queries with 15 nodes and a number of edges ranging between 20 and 100. Each result is the average over five randomly generated queries with the corresponding size. Again, our approach at $L = 3$ always outperforms $L = 1, 2$ and both of Random Decomp and No SS Reduction. $L = 1$ runs out of memory at the query $(15, 20)$ due to the large number of matches of that query (because it is very sparse). Therefore, we do not show its running time. Furthermore, there are configurations which have at least one run of the five runs whose execution time exceeded the maximum time allowed of 15 minutes. Those configurations are $L = 1$ at q(15,40), q(15,100), No SS Reduction at q(15,20), q(15,100), and Random Decomp at q(15,20).

Varying input graph degree of uncertainty. In this experiment, we study the effect of the degree of uncertainty in the PEG on the running time of our proposed approach, by varying the number of uncertain nodes and edges from 20% to 100%. We use query sizes q(5,5) and q(5,9) (in Figure 5.8(e)), and q(10,20) and q(10,40)

(in Figure 5.8(f)), using a query threshold of 0.7. As we can see, $L = 3$ always outperforms $L = 1, 2$, while $L = 2$ outperforms $L = 1$ for all degrees of uncertainty larger than 20%.

Varying input graph size. In this experiment, we study the performance of our proposed approach for all four input graph size settings, corresponding to graphs whose number of edges varies between 300 thousand and 6 million. We use query sizes q(5,5) and q(5,9) (in Figure 5.9(a)), and q(10,20) and q(10,40) (in Figure 5.9(b)), using a query threshold of 0.7. As we can see, $L = 1$ runs out of memory at both 500k and 1m, with query q(5,5) due to the high number of matches, while $L = 2, 3$ finishes normally in those cases. Otherwise, $L = 3$ outperforms $L = 1, 2$ in most cases.

Varying input query threshold. In this experiment, we vary the query threshold between 0.9 and 0.3. We use queries of size q(5,5), q(5,9) (in Figure 5.9(c)) and q(10,20), q(10,40) (in Figure 5.9(d)), using the 100k dataset. We can see that the performance improves for all path lengths with the increase of the lower input probability threshold, but at the same time, the performance of lower path lengths is the most sensitive to the change in the threshold, indicating that higher path lengths are the most stable with respect to such a parameter.

5.6.2.2 Search Space Performance

In this set of experiments, we study the search space performance, measured as the product of the candidate list sizes, and its reduction throughout different steps of our proposed method, under different circumstances.

Search Space Progression. In this experiment, we study the progression of the search space size throughout the main steps of our online querying algorithm. The results are depicted in Figure 5.9(e). The first step (labeled Path) refers to the search space size resulting from querying the path index. The second step (labeled Path+Context) is the size of the search space after pruning based on context information, that is, node-based neighborhood information, path neighbors and path cycle, as discussed in Section 5.5.2.2. The last step (labeled Final) refers to the final search space size after applying the mutual search space reduction using the k-partite graph representation (Section 5.5.2.4). We use a randomly generated query of size (5,7) with query threshold of 0.7 over two 100k datasets, one with 20% uncertainty, and the other with 80% uncertainty. Figure 5.9(e) shows the performance of our approach (in log scale) using the three path lengths of $L = 1, 2, 3$. As we can see, the mutual search space reduction step (Final) achieves effective reduction for all path lengths, although it is more effective with shorter path lengths. This is due to the fact that decompositions with shorter paths take into account information from smaller neighborhoods, and thus benefit more from distant information obtained via message passing. In contrast, the previous step (Path+Context) is most effective for longer paths, as those provide more context information for pruning. Also, generally,

higher degree of uncertainty results in smaller search spaces, because more paths are pruned at every step compared to lower degrees of uncertainty. Finally, we can see that overall, the final search space for longer paths is much smaller than that for shorter ones, which emphasizes the effectiveness of higher values of L in producing much smaller search spaces: *14 orders of magnitude smaller, comparing* $L = 3$ *to* $L = 1$ *at* 20%.

Joint Search Space Reduction Performance. In this experiment, we study the mutual search space reduction step (Section 5.5.2.4) in more detail, taking a closer look at the performance of both reduction methods: reduction by structure (ST), and reduction by upperbounds (UP). We use graphs of size 100k, a query that is a cycle with 5 nodes and 5 edges, and threshold 0.1. We have chosen a cycle query because it has a high diameter, thus illustrating the performance of information exchange using both reduction methods along the edges. For each method, we measure its reduction by dividing its resulting search space size by the initial search space size immediately before the reduction algorithm starts. Of course, since reduction by upperbounds is performed after reduction by structure, it will always perform higher reduction, but we are interested in its contribution to the overall reduction, and how it is affected by different parameters. Figure 5.9(f) shows the search space reduction for both ST and UP using three different path lengths 1, 2, 3 over graphs whose degrees of uncertainty vary from 20% to 80%. We do not show the case (UP,L=3) because the algorithm terminated (i.e., no further changes took place) before reduction by upperbounds already. As we can see, the effect

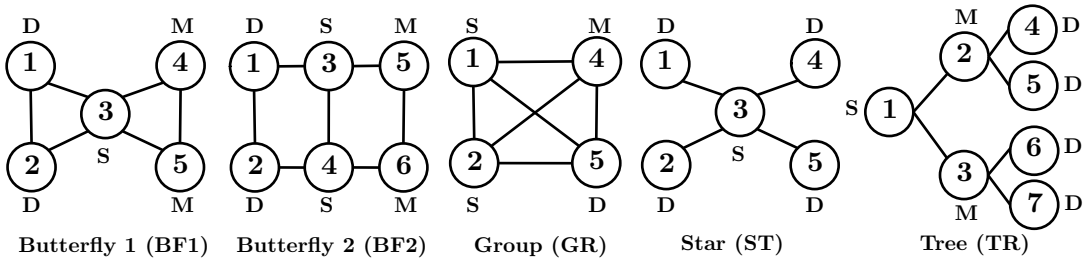


Figure 5.10: Collaboration pattern queries for real-world data.

of both reduction methods increases with the degree of uncertainty in the graph, again because more paths can be pruned. However, particularly, the effectiveness of UP increases with increased degree of uncertainty, as increased uncertainty often results in tighter upperbounds. Finally, we observe that reduction by upperbounds is more effective with shorter path lengths, as those obtain more additional information during message passing, while longer ones have already exploited part of this information during context based pruning and reduction by structure.

5.6.3 Performance on Real-world Data

In this subsection, we show our experimental results on a real-world dataset. We extracted collaboration data from DBLP. The nodes of the graph represent authors, the edges represent collaboration relationships. We annotated the collaboration graph with probabilistic data to capture different types of uncertaintities. For every author, we assigned a probability distribution over the areas that she/he is interested in, which can be Databases, Machine Learning or Software Engineering. We extract this information by counting the author's relative contribution in each area's conferences. For example, SIGMOD, VLDB, and ICDE count towards

Database interests, while ICSE, FSE and ICSM count towards Software Engineering interests, and so on. We assign existence probabilities between 0.5 and 1 to edges, depending on the number of times the pair of authors has collaborated. We create a reference set for every pair of authors whose names have normalized string similarity score above 0.9. The resulting graph has 16.8k nodes and 40.3k edges. We run probabilistic subgraph pattern matching using the collaboration patterns shown in Figure 5.10 with a query threshold of 0.01. Running times of the online phase using L=1,2,3 are shown in Figure 5.9(g). As we can see, L=3 outperforms L=2, which in turn outperforms L=1, for all queries except the star query, which has a maximum path length of two and thus does not benefit from indexing paths up to L=3. Another observation is that we found interesting insights about multi-disciplinary collaborations that span the three areas, where collaborations are somehow unusual. For example, the highest probability match of the tree query returned the group of authors: (Chao Liu, Christos Faloutsos, Jiawei Han) as the matches for nodes (1, 2, 3) respectively. On his webpage, Chao Liu states that he has worked on statistical methods for software reliability during his PhD, and after joining MSR, his research has shifted to Web search and statistical models to interpret web users' behaviors. Furthermore, Chao Liu collaborates with both of Christos Faloutsos and Jiawei Han who are known to be prolific authors in the fields of machine learning and databases, which also coincides with the query that specifies that nodes 2 and 3 are interested in machine learning, and their collaborators are interested in databases.

Chapter 6

Conclusions

In this dissertation, we addressed the problems of efficiently and declaratively cleaning, analyzing and querying graph-structured data.

On the cleaning aspect, we proposed a declarative framework to clean noisy graph data observed in various domains. We described the design of a data management system, called GrDB, for supporting declarative graph cleaning over noisy information networks. The system supports new constructs for defining graph-based inference operations, and heavily exploits the common properties shared by these operators to enable efficient storage and execution. The reason for choosing Datalog is its expressive power and ability to represent computation over graphs in an intuitive and easy-to-understand manner, in addition to its support for recursion, a commonly occurring operation in graph algorithms. We built a prototype system that implements this functionality and showed how to efficiently perform incremental maintenance of user-defined features and domain views. We presented a performance evaluation of the system; and the results show that the proposed approach can efficiently handle a wide spectrum of graph cleaning operations, like attribute prediction, link prediction and entity resolution.

For analyzing graph data, we proposed a new type of graph analysis query called *ego-centric pattern census*. The proposed approach for analysis focuses on

local properties of graph nodes, as opposed to traditional, well-known *global* graph properties such as community structure or PageRank. Ego-centric pattern census has broad applications in a variety of domains including targeted marketing, brokerage analysis, and social sciences. We designed a general and flexible language for specifying pattern census queries, and developed efficient algorithms for answering such queries. The results of a comprehensive experimental evaluation over a prototype system illustrate that the proposed algorithms can efficiently evaluate pattern census queries over large graphs.

For querying certain graph data, we designed and implemented a subgraph pattern matching algorithm that efficiently finds matches of a query graph in a large database graph. The key idea behind the proposed approach is to perform mutual pruning between candidate nodes and nodes that can be their neighbors in the final answer, i.e., candidate neighbors. By mutually reducing these two types of lists, the final search space can be reduced significantly. Performance evaluation shows that the proposed algorithm outperforms a state-of-the-art approach [9] by two orders of magnitude.

For querying uncertain graph data, we presented a probabilistic approach for modeling uncertain graphs and answering queries over them. The proposed graph model, called probabilistic entity graphs, captures node attribute uncertainty, edge existence uncertainty, and identity uncertainty. We presented efficient algorithms to solve subgraph pattern matching queries over such uncertain graphs, where queries are expressed and evaluated at the entity-level. Experimental evaluation shows that the proposed algorithm outperforms an equivalent SQL implementation by multiple

orders of magnitude.

In this dissertation, we have shown that declarative graph processing is a very powerful tool that can provide graph database users, programmers, and analysts both ease of use and powerful querying capabilities. There are many lessons we have learnt from building evaluation engines for declarative graph programs and queries:

- A considerable number of graph processing algorithms are iterative. In this dissertation, we have presented iterative algorithms for graph cleaning and querying. In graph cleaning, the algorithms are iterative because the predictions are applied in iterations according to their confidence values, and in graph querying, for both certain and uncertain graph cases, the algorithms are iterative because of the mutual reduction of search space step. Supporting fast iterative graph processing is a crucial step for implementing scalable graph algorithms.

- Iterative algorithms execute by performing repeated passes over the graph and usually introducing small changes after every iteration. Iterative graph algorithms are more efficient if they only react to the changes after every iteration, and avoid performing the full computation repeatedly over the entire graph. As we have seen, in graph cleaning and querying, we have implemented incremental evaluation of their iterative algorithms, and have shown that it outperforms the re-computation model.

- Many graph algorithms process the graph by exploring the local neighbor-

hoods around nodes. For example, in feature construction for graph cleaning, prediction features for nodes are extracted by examining the nodes' local neighborhoods. In ego-centric pattern census, patterns are searched in the k-hop neighborhood of nodes, and in graph querying, node neighborhood statistics are used to summarize information about the node and help to detect if it will participate in the query result. For building efficient graph cleaning, analysis, and querying systems, paying attention to the storage and retrieval of node neighborhoods in an efficient manner is crucial. In Chapters 4 and 5, we used node neighborhood summaries to avoid accessing the entire node neighborhood when it is sufficient to look up the summaries.

- When processing multiple node neighborhoods, it is important to consider the overlap between them, and utilize it to save computation. Our techniques for ego-centric pattern census show that methods that utilize overlap outperform methods that do not. In Chapter 4, we have used the shingle ordering [113] to access graph nodes in an order that helps exploiting their shared neighborhoods.

- Node-centric graph algorithms are highly parallelizable. We have discussed a parallel implementation of uncertain graph querying in Chapter 5.

In this dissertation, we have presented declarative methods for performing graph cleaning, analysis and querying. We have shown that the general design considerations mentioned above, and others that are specific to each task, enable us to process very large graph datasets. While in some cases, no other systems are

available to solve our problems (specifically, ego-centric pattern census (Chapter 4) and querying uncertain graphs with identity uncertainty (Chapter 5)), in all the cases, our approaches are superior to their baselines or prior approaches, and some of our optimizations lead to multiple orders of magnitude reductions over those.

Bibliography

[1] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[2] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[3] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[4] S. Mangan and U. Alon. Structure and function of the feed-forward loop network motif. *Proceedings of The National Academy of Sciences*, 100:11980–11985, 2003.

[5] Uri Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450–461, 2007.

[6] Marco Baiesi. Scaling and precursor motifs in earthquake networks. *Physica A*, 360(2):4, 2004.

[7] Shalev Itzkovitz and Uri Alon. Subgraphs and network motifs in geometric networks. *Phys. Rev. E*, 71, 2005.

[8] Sergi Valverde and Ricard V. Solé. Network motifs in computational graphs: A case study in software architecture. *Physical Review E*, 72(2):026107, August 2005.

[9] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[10] Jennifer Neville and David Jensen. Collective classification with relational dependency networks. In *Proc. of KDD Workshop on Multi-Relational Data Mining*, 2003.

[11] Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[12] Qing Lu and Lise Getoor. Link-based classification. In *ICML*, 2003.

[13] D. Jensen, J. Neville, and B. Gallagher. Why collective inference improves relational classification. In *SIGKDD*, 2004.

[14] Sofus A. Macskassy and Foster J. Provost. Classification in networked data: A toolkit and a univariate case study. *Journal of Machine Learning Research*, 8:935–983, 2007.

[15] J. Neville and D. Jensen. Iterative classification in relational data. In *AAAI Workshop on Learning Statistical Models from Relational Data*, 2000.

[16] Luke McDowell, Kalyan Moy Gupta, and David W. Aha. Cautious inference in collective classification. In *AAAI*, 2007.

[17] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[18] Ben Taskar, Abbeel Pieter, and Daphne Koller. Discriminative probabilistic models for relational data. In *UAI*, 2002.

[19] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[20] Matthias Bröcheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *UAI*, pages 73–82, 2010.

[21] Linyuan Lu and Tao Zhou. Link prediction in complex networks: A survey. *CoRR*, abs/1010.0725, 2010.

[22] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[23] Lada Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[24] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:3943, 1953.

[25] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006.

[26] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. In *KDD*, pages 538–543, 2002.

[27] Ben Taskar, Ming-Fai Wong, Pieter Abbeel, and Daphne Koller. Link prediction in relational data. In *Advances in Neural Information Processing Systems*, 2004.

[28] Alex Memory, Angelika Kimmig, Stephen H. Bach, Louiqa Raschid, and Lise Getoor. Graph summarization in annotated data using probabilistic soft logic. In *URSW*, pages 75–86, 2012.

[29] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proc. of IJCAI Workshop on Information Integration*, August 2003.

[30] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 2008.

[31] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[32] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.

[33] D. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting relationships for domain-independent data cleaning. In *SIAM SDM*, 2005.

[34] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM TKDD*, 1:1–36, 2007.

[35] P Singla and P Domingos. Entity resolution with markov logic. *IEEE International Conference on Data Mining (ICDM 2006)*, 21:572–582, 2006.

[36] Galileo Namata, Stanley Kok, and Lise Getoor. Collective graph identification. In *KDD*, pages 87–95, 2011.

[37] Arvind Arasu, Christopher Re, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[38] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.

[39] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic data base language (LDL1). In *PODS*, 1987.

[40] Raghu Ramakrishnan, Kenneth Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *ILPS*, 1994.

[41] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[42] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 1987.

[43] Boon Loo, Tyson Condie, Minos Garofalakis, David Gay, Joseph Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[44] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.

[45] Royi Ronen and Oded Shmueli. Evaluating very large datalog queries on social networks. In *EDBT*, 2009.

[46] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[47] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.

[48] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[49] L.C. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 40:35–41, 1977.

[50] Michelle Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academu of Science, USA*, 99:7821, 2002.

[51] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Science*, 103(23):8577–8582, June 2006.

[52] Martin Everett and Stephen P. Borgatti. Ego network betweenness. *Social Networks*, 27(1):31–38, 2005.

[53] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[54] Marc A. Smith, Ben Shneiderman, Natasa Milic-Frayling, Eduarda Mendes Rodrigues, Vladimir Barash, Cody Dunne, Tony Capone, Adam Perer, and Eric Gleave. Analyzing (social media) networks with nodexl. In *Proceedings of the fourth international conference on Communities and technologies*, pages 255–264, 2009.

[55] Ronald Burt. *Structural holes: The social structure of competition.* Harvard University Press, 1992.

[56] Jon M. Kleinberg, Siddharth Suri, Eva Tardos, and Tom Wexler. Strategic network formation with structural holes. *Sigecom Exchanges*, 7:284–293, 2008.

[57] EgoNet. `http://egonet.sf.net`.

[58] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42:844–856, July 1995.

[59] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[60] Michael Fellows, Guillaume Fertin, Danny Hermelin, and Stphane Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *ICALP*, 2007.

[61] Mariano Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, 1990.

[62] Marc Gyssens, Jan Paredaens, and Dirk van Gucht. A graph-oriented object database model. In *PODS*, 1990.

[63] Ralf Hartmut Güting. GraphDB: Modeling and querying graphs in databases. In *VLDB*, 1994.

[64] L. Sheng and G. Özsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.

[65] Ulf Leser. A query language for biological networks. *Bioinformatics*, 21:33–39, January 2005.

[66] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal of Computing*, 32(5):1338–1355, 2003.

[67] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, 2007.

[68] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Grail: scalable reachability index for large graphs. *Proceedings of VLDB Endowment*, 3:276–284, September 2010.

[69] Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast discovery of connection subgraphs. In *KDD*, 2004.

[70] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, 2006.

[71] Petros Drineas, Alan M. Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Machine Learning*, 56(1-3):9–33, 2004.

[72] Kathy Macropol and Ambuj K. Singh. Scalable discovery of best clusters on large graphs. *PVLDB*, 3(1):693–702, 2010.

[73] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[74] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[75] Huahai He and Ambuj K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[76] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: tree + delta <= graph. In *VLDB*, 2007.

[77] Shijie Zhang, Meng Hu, and Jiong Yang. TreePi: A novel graph indexing method. In *ICDE*, 2007.

[78] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. FG-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[79] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1:364–375, August 2008.

[80] Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT*, 2009.

[81] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3:340–351, September 2010.

[82] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: from intractable to polynomial time. *Proc. VLDB Endow.*, 3:264–275, September 2010.

[83] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[84] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: pattern match query in a large graph database. *Proc. VLDB Endow.*, 2:886–897, August 2009.

[85] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16(4):523–544, 2007.

[86] Omar Benjelloun, Anish Das, Sarma Alon, and Halevy Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[87] Prithviraj Sen, Amol Deshpande, and Lise Getoor. PrDB: Managing and exploiting rich correlations in probabilistic databases. *VLDB Journal, special issue on uncertain and probabilistic databases*, 2009.

[88] Bhargav Kanagal and Amol Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, 2009.

[89] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. k-nearest neighbors in uncertain graphs. *PVLDB*, 3(1):997–1008, 2010.

[90] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. Distance-constraint reachability computation in uncertain graphs. *PVLDB*, 4(9):551–562, 2011.

[91] Ruoming Jin, Lin Liu, and Charu C. Aggarwal. Discovering highly reliable subgraphs in uncertain graphs. In *KDD*, 2011.

[92] Petteri Hintsanen and Hannu Toivonen. Finding reliable subgraphs from large probabilistic graphs. *Data Mining and Knowledge Discovery*, 17(1):3–23, 2008.

[93] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Finding top-k maximal cliques in an uncertain graph. In *ICDE*, 2010.

[94] Odysseas Papapetrou, Ekaterini Ioannou, and Dimitrios Skoutas. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *EDBT/ICDT*, 2011.

[95] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Mining frequent subgraph patterns from uncertain graph data. *IEEE Transactions on Knowledge and Data Engineering*, 22(9):1203–1218, 2010.

[96] Lei Chen and Changliang Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *IEEE Transactions on Knowledge and Data Engineering*, 22(8):1093–1109, 2010.

[97] Ye Yuan, Guoren Wang, Haixun Wang, and Lei Chen. Efficient subgraph search over large uncertain graphs. *PVLDB*, 4(11):876–886, 2011.

[98] Octavian Udrea, V. S. Subrahmanian, and Zoran Majkic. Probabilistic RDF. In *IRI*, pages 172–177, 2006.

[99] Hai Huang and Chengfei Liu. Query evaluation on probabilistic RDF databases. In *WISE*, volume 5802 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2009.

[100] Xiang Lian and Lei Chen. Efficient query answering in probabilistic RDF graphs. In *SIGMOD*, 2011.

[101] Ekaterini Ioannou, Wolfgang Nejdl, Claudia Niederée, and Yannis Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *PVLDB*, 3(1):429–438, 2010.

[102] Ming Hua and Jian Pei. Aggregate queries on probabilistic record linkages. In *EDBT*, pages 360–371, 2012.

[103] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[104] Himanshu Gupta and Inderpal Singh Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.*, 2006.

[105] P. A. Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, 2007.

[106] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM TKDD*, 2007.

[107] D. Cartwright and F. Harary. Structural balance: a generalization of Heider's theory. *Psychological Review*, 63(5):277–93, 1956.

[108] Yuanyuan Tian, Richard C. Mceachin, Carlos Santos, David J. States, and Jignesh M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23:232–239, January 2007.

[109] Edward G. Allan, Jr., William H. Turkett, Jr., and Errin W. Fulp. Using network motifs to identify application protocols. In *GLOBECOM*, 2009.

[110] Valery Van Kerrebroeck and Enzo Marinari. Ranking by loops: a new approach to categorization. *Phys. Rev. Lett.*, 101:098701, 2008.

[111] Natasa Przülj. Biological network comparison using graphlet degree distribution. *Bioinformatics/computer Applications in the Biosciences*, 23:177–183, 2007.

[112] Bin Jiang and Christophe Claramunt. Topological analysis of urban street networks. *Environment and Planning B: Planning and Design*, 31:151–162, 2004.

[113] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, 2009.

[114] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[115] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[116] Neo4j open source NoSQL graph database. `http://neo4j.org/`.

[117] Ye Yuan, Guoren Wang, Lei Chen, and Haixun Wang. Efficient subgraph similarity search on large probabilistic graph databases. *PVLDB*, 5(9):800–811, 2012.

[118] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.