ABSTRACT

Title of dissertation:	SOCIAL NETWORK DATA MANAGEMENT
	Matthias Broecheler, Doctor of Philosophy, 2011
Dissertation directed by:	Professor V.S. Subrahmanian Department of Computer Science

With the increasing usage of online social networks and the semantic web's graph structured RDF framework, and the rising adoption of networks in various fields from biology to social science, there is a rapidly growing need for indexing, querying, and analyzing massive graph structured data. Facebook has amassed over 500 million users creating huge volumes of highly connected data. Governments have made RDF datasets containing billions of triples available to the public. In the life sciences, researches have started to connect disparate data sets of research results into one giant network of valuable information. Clearly, networks are becoming increasingly popular and growing rapidly in size, requiring scalable solutions for network data management.

This thesis focuses on the following aspects of network data management. We present a hierarchical index structure for external memory storage of network data that aims to maximize data locality. We propose efficient algorithms to answer subgraph matching queries against network databases and discuss effective pruning strategies to improve performance. We show how adaptive cost models can speed up subgraph matching query answering by assigning budgets to index retrieval operations and adjusting the query plan while executing.

We develop a cloud oriented social network database, COSI, which handles massive network datasets too large for a single computer by partitioning the data across multiple machines and achieving high performance query answering through asynchronous parallelization and cluster-aware heuristics.

Tracking multiple standing queries against a social network database is much faster with our novel multi-view maintenance algorithm, which exploits common substructures between queries.

To capture uncertainty inherent in social network querying, we define probabilistic subgraph matching queries over deterministic graph data and propose algorithms to answer them efficiently.

Finally, we introduce a general relational machine learning framework and rule-based language, Probabilistic Soft Logic, to learn from and probabilistically reason about social network data and describe applications to information integration and information fusion.

Social Network Data Management

by

Matthias Broecheler

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
2011

Advisory Committee:
Professor V.S. Subrahmanian, Chair
Professor Lise Getoor
Professor Amol Deshpande
Professor Ashok Agrawala
Professor Michele J. Gelfand

© Copyright by
Matthias Broecheler
2011

Preface

Some of the material presented in this dissertation has been previously published and peer-reviewed. In particular:

1. Parts of Chapter 4 were first published in [17].
2. Parts of Chapter 5 were first published in [23].
3. Parts of Chapter 6 were first published in [18].
4. Parts of Chapter 7 are currently under review.
5. Parts of Chapter 8 were first published in [22].
6. Parts of Chapter 9 were first published in [19, 21].

The author of this dissertation is the first and main author for all of the above mentioned publications.

Acknowledgements

By definition, a dissertation is decorated with only one author. However, the body of work presented in the following would not have been possible without the help and support from my advisors, colleagues, collaborators, family and friends to all of whom I want to express my sincere gratitude.

Specifically, I want to thank my advisor, Professor V.S. Subrahmanian, for all of his guidance, support, and trust over the past 4 and a half years. He taught me countless important lessons about research and challenged me with interesting research problems and thought provoking discussions. I am also grateful that he repeatedly endured painful first paper drafts. Most of the research presented in this dissertation was conducted in collaboration with Professor Andrea Pugliese from the University of Calabria. I thank him for his many invaluable contributions, enticing discussions, and his world-class Latex space saving techniques. Vanina, Amy, Gerardo, and Cristian were wonderful lab-mates, friends, and collaborators for most of the way. I thank them for steering me through the grad school jungle and keeping spirits high in light of daunting work loads. Dan was instrumental in turning some of the research visions presented in this thesis into running prototypes and I thank him for maintaining his diligence and rigor despite confusing project milestones and impossible deadlines.

I am very grateful to Professor Lise Getoor for providing me with the opportunity to explore machine learning research under her guidance. I thank her for sharing her knowledge and vision, facilitating many valuable connections, and keeping meeting marathons to the point. I also want to thank Stephen, Lilyana, and

Stanley for joining me on the PSL journey and their innumerable contributions that helped turn a class project into a robust and efficient machine learning framework.

I thank Professor Michele J. Gelfand, Professor Ashok Agrawala, and Professor Amol Deshpande for joining my dissertation committee and providing valuable feedback. I also thank Professor Samir Khuller as the "honorary" committee member who unfortunately had to resign from the committee.

I am deeply indebted to Franziska for giving me her utmost support through all of grad school, especially during the stressful periods. Her ability to cheer me up and watch my back was a tremendous help and kept me going. Last, but certainly not least, I want to thank my parents and family – for everything. Their guidance and encouragement made the long climb towards the PhD a whole lot easier.

Table of Contents

List of Tables

List of Figures

Part I

Social Network Data

Chapter 1
Introduction

Facebook. LinkedIn. Twitter. These are just three of hundreds of social networks functioning efficiently around the world today. Whether they serve to support social interaction (Facebook) or quick reporting of events of interest (Twitter) or the creation of professional networks (LinkedIn), social networks are here to stay. Moreover, they are growing at a dramatic pace. Table 1 shows the number of US users in these social networks.[1]

Social Network	Number of US users	Date
Facebook	129M	Nov. 25, 2010
LinkedIn	24.6M	Jan. 22, 2011
Twitter	75.6M	Nov. 25, 2010

Table 1.1: Number of US users in social networks

Whether these numbers are correct or not, it is clear that the total numbers are large – even if one considers just the US. In the case of Twitter alone, the number of users across the world is estimated at 190M (as on Jan. 25, 2011) and an estimated 65M tweets are being posted each day. According to Experian HitWise, Facebook became the world's most visited web site in 2010, surpassing Google, and accounting for 8.93% of site visits between January and November 2010.[2]

Over and above "stereotypical" social networks such as those listed above, the world is full of many other types of social networks. Phone companies are increasingly evolving into "social network" companies. Customers who call one another or exchange text messages or share image or video traffic increasingly form parts of social networks operating "on the air." Voice over IP providers like Skype are also well aware that not only their subscriber base, but non-users who are called on regular land-line phones or cell phones also form part of a rich social network. And many companies have "internal" social networks that they use within their companies to promote free-flowing discussion within corporate boundaries. The spectacular growth in social networks has not gone unnoticed by companies eager to exploit the rich wealth of the resulting data in order to return tidy profits to their shareholders.

In this thesis, we address the following questions:

(Q1) *What constitutes a social network?* Is a social network simply a set of individuals, connected together in some way? Should non-human artifacts be included as nodes in the social network (e.g. should the comments someone wrote on a Facebook wall, or a tweet sent by an individual on Twitter be considered

[1]Statistics estimated by `www.quantcast.com`

[2]`http://socialnetworkingwatch.typepad.com/.a/6a00e54f9b07dc88340148c72eb93b970c-pi`

nodes of the social network)? Should properties of nodes (e.g. male/female, age demographics) be considered part of the definition of a social network? Should edges be labeled with the property or properties that link those nodes together? Most people agree that "friend" should be a valid edge label, but perhaps "tagged" should be an edge label as well, saying a particular person tagged someone in an image. What is the possible space of edge labels? Should edges be weighted to indicate the strength of a relation (e.g. the number of messages one person posted to another person's Facebook wall might be a proxy for the interest the first person has in the other). Should edges have temporal tags expressing information about the period the relationship was valid (valid-time in database parlance) or when the edge was inserted (transaction time in database parlance)?

(Q2) *What kinds of queries do users want to pose against a social network database?* Given a body of information about a social network, what kinds of queries do users and applications wish to execute against such a social network database? Clearly, the space of possible queries is very large and depends upon the formal definition of a social network. At the very least, users should be able to ask simple queries such as "Find the person who wrote on John's Facebook wall last week, is a friend of Joe, and attended the Christmas Party (2010) organized by Mary". A user may want to use such a query to find a person he met at a party (but whose name he forgot). However, an organization like Twitter itself might wish to ask a whole different set of queries – *top-k* queries such as "Find the k people who wrote the most posts on walls in North America about uprisings in Egypt"? We will describe various classes of queries that different users and customers (internal and external) of a social network might want to ask and present algorithms to answer those efficiently.

(Q3) *What kind of database structures should we use to store social networks so as to efficiently answer such queries?* It is clear that information in most social networks of the type we see today can be readily stored in a relational database. What are the pros and cons of this approach? Are there other structures that can more efficiently answer the types of questions users will/ are posing to social networks? The bulk of this thesis will focus on answering certain types of queries to real-world social networks that are massive in size using specialized index structures.

In this chapter, we present the following four real-world use cases for social networks that will be used as running examples throughout the thesis to illustrate definitions, data structures, algorithms, and theoretical results.

Stock Market Regulatory Example A stock market regulatory network contains information about companies, stocks, and who traded what stocks. A major problem in such networks is to identify suspicious transactions, i.e. transactions fitting certain patterns. Government regulators are interested in identifying collusion between traders to either drive a stock price up or down.

Investors, on the other hand, are interested in exploiting such networks for information which might lead to changes in stock price.

Phone Network Example Our phone network example will consider a social network consisting of phone users who interact with each other through voice, text message, image and video transfers. Phone companies may use such social networks to identify fraud, analyze their user base and/or promote new phone plans and products to their customers.

Online Social Network Example There are the straightforward examples where social network users want to find others they know or others with whom they share a common interest. Such an application might involve searching through multiple online social networks.

Company Social Network Example Many companies have "internal" social networks capturing the interactions between employees, managers, customers and virtual as well as physical assets. Such social networks may be used to analyze bottlenecks in the information flow.

The rest of this chapter presents these three examples in some detail, explaining informally what a social network is, and providing examples of the kinds of queries that users typically ask in such applications.

This chapter sets the stage for Chapter 2 which provides a formal mathematical definition of social networks as graphs, together with a formal query language within which users and applications can query social networks.

The bulk of the thesis is devoted to answering question (Q3) above – how should we store social networks so that these types of queries can be efficiently answered? How can we couple the specialized data structures for storing social networks with scalable algorithms that will help answer queries on social networks containing hundreds of millions of nodes and billions of edges?

We start in Chapter 3 by describing techniques that have been proposed in the literature for storing graph data (e.g. based on relational databases). We describe the pros and cons of these approaches and suggest when they will work well, and when they will not.

Chapter 4 describes a disk-based index using which a large class of queries (called *subgraph matching* queries) to social networks can be efficiently answered.

In Chapter 5, we describe an efficient algorithm to answer subgraph matching queries to social networks that uses the notion of a "budget."

Chapter 6 shows how cloud computing technology can be used to efficiently process subgraph matching queries for greater scalability.

Chapter 7 shows how users can describe standing queries (or views) and how the answers to these queries can be incrementally maintained as new transactions cause the underlying social network to change.

Chapter 8 shows how we can efficiently process queries when users are uncertain about their query.

Finally, we introduce a machine learning framework to probabilistically reason about and learn from social network data in Chapter 9.

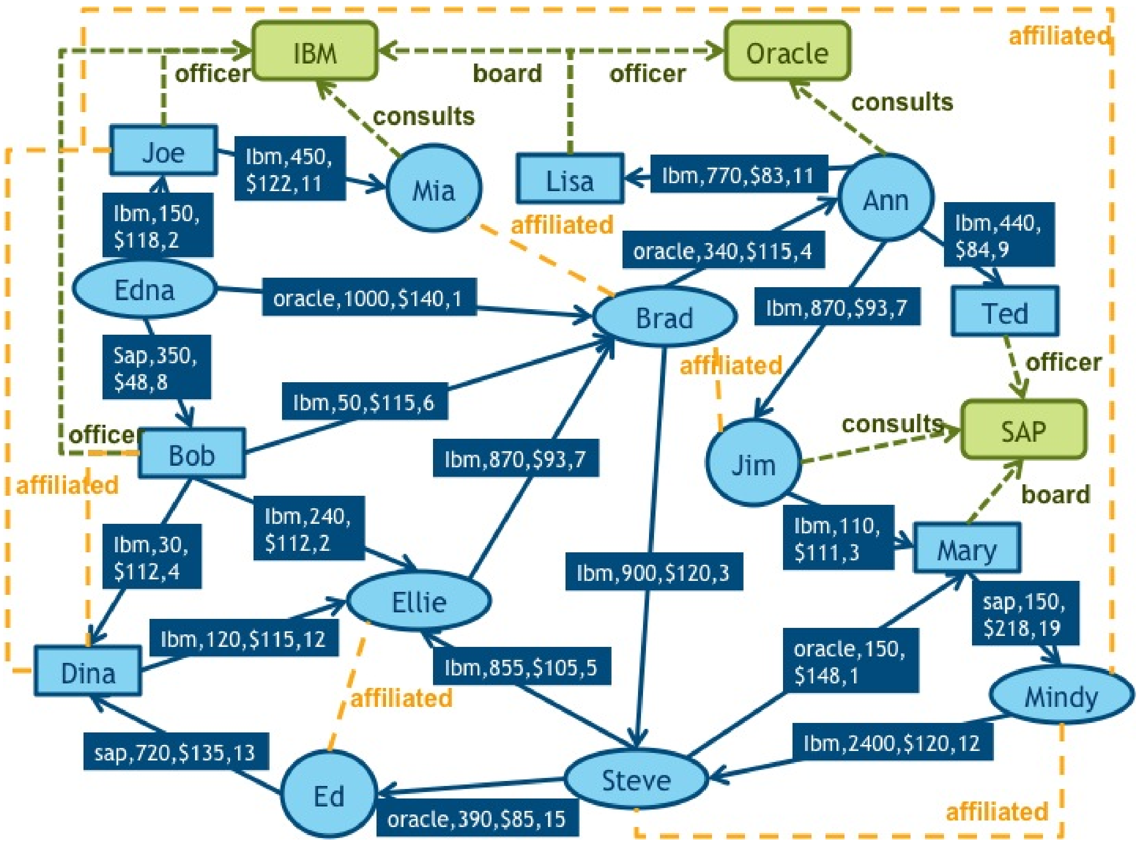


Figure 1.1: Toy "Stock Trade" Graph Database

1.1 Stock Market Example

Consider a social network consisting of individual investors, institutional investors, and authorized traders. A stock exchange is constantly involved in "trading" in which a certain stock is bought or sold. Figure 1.1 shows a small example of a set of people engaged in trading stock on a given stock exchange.

- The rectangular nodes represent individual investors, circular nodes represent institutional investors, and oval nodes represent traders. For instance, Joe, Lisa and Ted are all individual investors. Jim, Rod and Mia are institutional investors, while Ellie, Mindy and Brad are authorized traders.
- An edge is drawn from node A to node B if A sold a stock to B. For instance, the fact that there is an edge from Edna to Brad indicates that Edna sold stock to Brad.
- The edge can be labeled with many properties. To keep things simple, in our example, the edges represent the name of the stock, the quantity of stock sold, the per-price share at which the sale was made, and the time. In our figure, the edge from Edna to Brad is labeled (Oracle,1000,$140,1) indicating that

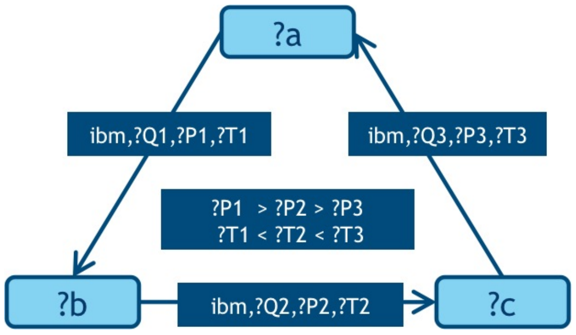


Figure 1.2: Toy "Stock Trade" Graph Query to find Colluding Investors

> Edna sold 1000 units of Oracle stock to Brad at time 1 at a price of \$140 per share.

A regulator may wish to find all sets of 3 investors (or a non-empty set of investors in general) such that all of these investors can sell stock to each other and, in turn, successively lower prices to increase the volume of stock traded and artificially deflate the stock price so that they can later buy the stock at a much lower price and realize gains when the stock price (naturally) moves back up [119]. This is a market manipulation that is illegal in many countries. This query can be expressed naturally as the graph pattern shown in Figure 1.2. In this query, we use expressions of form $?x$ to denote variables for which we seek bindings. Throughout this thesis, any node labeled with a label beginning with a "?" symbol denotes a variable (for which we would like to find a compatible binding). We also use rounded edge nodes to denote that the variables in question can be bound to any type of node in the Stock Trade database.

The query looks for exact matches of the query pattern in the database such that the "query constraint" (shown in the box in the center of the triangular region of the graph – we will define this term formally later) is satisfied.

Of course, this query can be easily extended and modified in many ways. We could easily add additional constraints – for instance, only trades above a certain amount (say 100K shares) might qualify for fraud. In this case, the query constraint $?P1 > ?P2 > ?P3 \,\&\, ?T1 < ?T2 < ?T3$ can be modified to $?P1 > ?P2 > ?P3 \,\&\, ?T1 < ?T2 < ?T3 \,\&\, ?Q1 > 100,000 \,\&\, ?Q2 > 100,000 \,\&\, ?Q3 > 100,000$. We might additionally want to qualify that the trades all occur within a short period of time (say 5 units) in which case we might additionally add the constraint $?T3 - ?T1 < 5$ to the query constraint shown. If we want to restrict the query to just brokers, then we could require the shape of the nodes to be oval shaped (in the figure). Likewise, in this query, we might want to insist that $?a, ?b, ?c$ all denote distinct nodes – in this case, we can add $?a \neq ?b \wedge ?a \neq ?b \wedge ?b \neq ?c$ as a "uniqueness" constraint to the query. *Throughout this thesis, whenever we express a query involving node variables, we will implicitly assume (unless explicitly stated otherwise) that the variables are all distinct and that such "uniqueness" constraints are implicitly present in the query.*

This shows that subgraph matching queries with additional query constraints can play a significant role in identifying suspicious groups of investors, thus helping

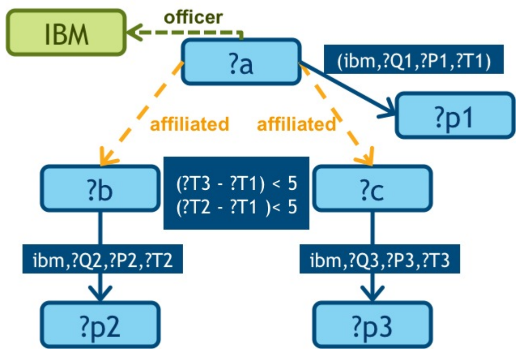


Figure 1.3: Toy "Stock Trade" Graph Query to find Insider Trades

regulatory bodies.

Continuing with the same example, in many countries, when a company officer sells (or buys) any stock, this is required to be disclosed to national regulatory bodies which often make the information public. Often times, such "insider" trades are closely watched by other investors. Suppose now that a third party brokerage house has notices that whenever a board member of IBM sells his stock in IBM, followed by sales of IBM stock by two (or more) of his close associates within a 5 time unit window, then the stock of IBM is likely to fall. This can be expressed via the query shown in Figure 1.3. This query tells us to find all exact matches for the query graph in the Stock Trade graph database shown in Figure 1.1 that satisfy the query constraint saying that all transactions occurred within a 5 time unit period after the IBM officer sold his stock.

It is clear that queries such as these represent queries that institutional investors might want to pose – if an IBM Board Member is selling off stock, followed closely by his affiliates, then there is reason to believe that this is a good time to sell IBM stock.

1.2 A Phone Example

Cell phone providers and Voice over IP providers are increasingly realizing the benefits of being connected. For instance, Skype and Verizon have recently signed a deal which offers Verizon users the ability to use Skype's free VoIP services (on Blackberry devices) whenever they are at a mobile WiFi hotspot. This means that two networks – Skype's online VoIP network, and Verizon's *physical* wired and wireless networks – are now logically integrated, at least to some degree. Both companies have a shared interest in exploiting this. Verizon has a social network consisting of information on who (amongst its subscribers) called who, how long they speak (on the average) per month, how much text message traffic they exchange, and so forth. Skype has something similar. As Skype supports calls to non-VoIP phones, this partnership integrates wired, wireless, and VoIP information about a huge number of consumers in the telecommunications sector.

Much work can be done in cases such as this one. [41] studied 94 subjects who were given cell phones and predicted friend relationships with 95% accuracy. Such applications might say, for example, that if A calls B for at least 30 minutes a month and B calls C for at least 40 minutes a month, and B and C are both friends (in Facebook) of common person D, then there is a high probability of A and D being friends. Given a person (John, for example), we might want to find all people D in the combined Skype-Verizon database with whom John has a high probability of being a friend. For instance, if John has adopted a new Skype/Verizon product, then Skype/Verizon might want to send D an email saying that John (who is related in the ways described to D) has adopted this plan, and maybe D wants to do so as well.

Figure 1.4 provides a small sample database of Facebook "friend" relations (shown via bold lines) and Skype "called" information (shown by dotted lines). The edge labels (numbers) in this figure are as follows. For Skype edges, the number along an edge (A, B) denotes the number of minutes spent in Skype calls initiated by A to B during a given time period. When the edge (A, B) denotes a Facebook friend edge, the integer labeling the edge shows the number of messages written by A on B's wall.

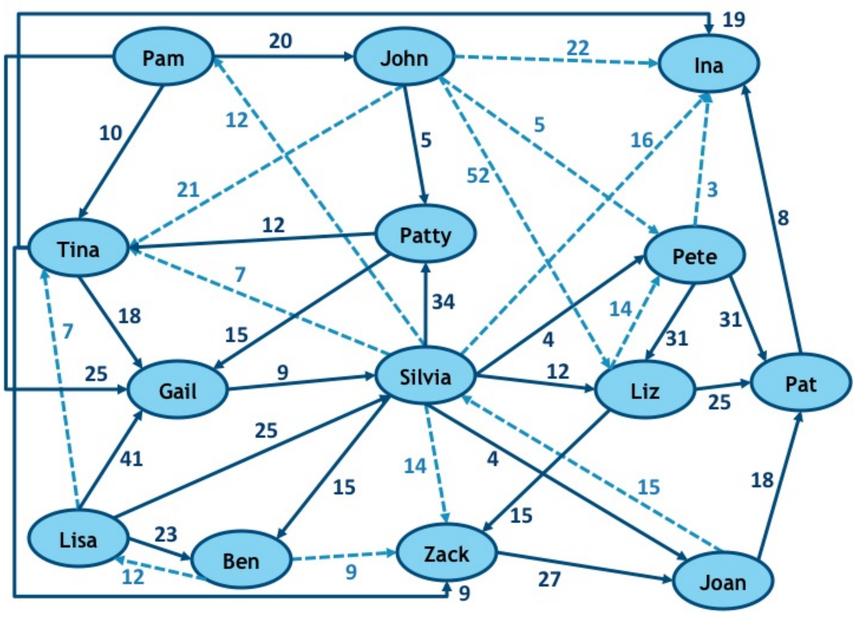


Figure 1.4: Toy Cell/VoIP Call Example. Facebook connections shown with bold lines, Skype connections shown as dotted lines

Figure 1.5 shows an example query. This query asks us to find all people $?B, ?C, ?D$ such that John calls both $?B, ?C$ on Skype for at least 10 minutes each (during the given time period) and both $?B, ?C$ write at least 5 times (during the given time period) on $?D$'s wall. In this case, Skype may decide that $?D$ is a potential person to whom they can market a VoIP plan that John has adopted.

In an alternative application for mobile phone vendors, a major international cell phone company told one of the authors that it is interested in identifying fraud

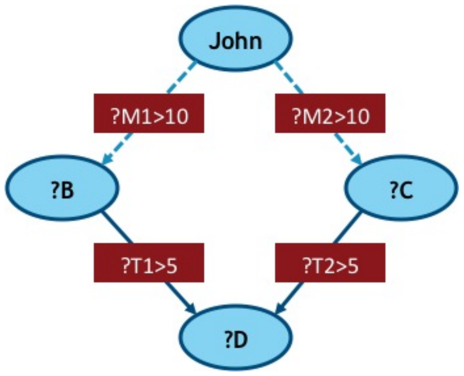


Figure 1.5: Toy Cell/VoIP Call Graph Query Example

in cell phone networks by searching for specific patterns in its social network consisting of hundreds of millions of nodes and billions of edges – there is much work being done on fraud detection in phone use [127]. Such methods fall into two categories – detecting *anomalous* use of such devices, and *identifying known patterns* of fraudulent use.

While fraud using cell phones is carried out in many ways, one well-known way (in the USA) is the so-called "*72" fraud. In this type of fraud, an individual calls someone ($?p$) and tells them a truly sad story, e.g. *I have been arrested by the police, but my five year old child needs to be picked up from school. I get only one phone call and I mis-dialed and got your phone number. Please call my wife and tell her to pick up our child immediately. Unfortunately, her number is a bit complicated. You have to dial *72 followed by a regular phone number* $?n$. If person $?p$ (the recipient of the call from the con artist) does follow the directions, what he does is "forward" his phone number to the number $?n$. After this, the person at phone number $?n$ can continuously call whatever phone numbers he or she wants to call, knowing that all calls from phone number $?n$ will be billed to person $?p$'s phone number.

A phone company that is looking for such fraud's might watch for patterns of the following form. Find all people (or phone numbers) $?p$ such that $?p$ dialed "*72" followed by a number $?n$ and after this, $?n$ made a pattern of suspicious phone calls. There are obviously many different ways that we could characterize "suspicious" phone calls. In our example, we merely assume that this means that $?n$ called at least three overseas phone numbers within a 60 minute interval. Figure 1.6 shows a graph reflecting the situation the phone company is interested in.

The above query is interesting from a number of different perspectives. First, we have only described one pattern of "suspicious" activities – there are likely to be many that the phone company needs to track. In addition, there are many different types of fraud patterns they need to monitor, not just *72 fraud patterns. In general, there is a set of patterns the phone company wishes to monitor. Moreover, they need to *continuously* watch the network for occurrences of these patterns as the social network is updated (in this case through new phone calls and other phone related transactions). This exposes the need for maintaining these queries as *views* and for managing these views and incrementally updating them as the underlying social

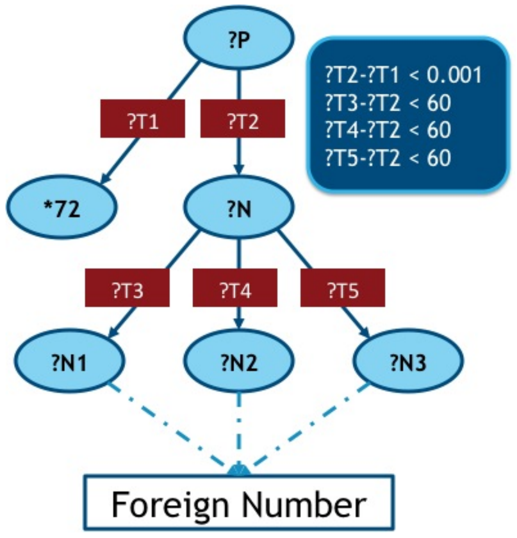


Figure 1.6: *72 Fraud Phone Call Graph Query Example

networks changes [63, 72]. We will discuss this in considerable detail in Chapter 7.

1.3 A Consumer-Oriented Social Network Example

A different application involves queries spanning across multiple social networks such as Flickr or Twitter.

Consider the small graph database depicted in Figure 1.7 which describes a social network of the type one might find on Facebook or Evite describing a set of people, events those people attended, things they like, friends and other relationships. The network here consists not only of people, but things those people like such as books and movies, events such as Peter's party, and descriptive terms such as "Drama" or "Thriller."

Consider the case of a user Francis who tries to recall the name of a drama movie recommended to him by both a friend and by someone he met at a party organized by Peter. This query can be formalized as the subgraph matching query shown graphically in Figure 1.8. Here, $?b$ is a variable denoting the movie Francis is trying to recall, $?f$ represents the friend who recommended it, and $?u$ is the person he met at Peter's party.

Though most of the previous database examples we have given (stock market, phone calls) deal with social network database applications for corporations or governments, queries such as the one listed above are meant for ordinary people who want to recall something or the other about their social network. This is just one example query – there are many others as well that we can think of that an individual Facebook or Twitter user might want to ask.

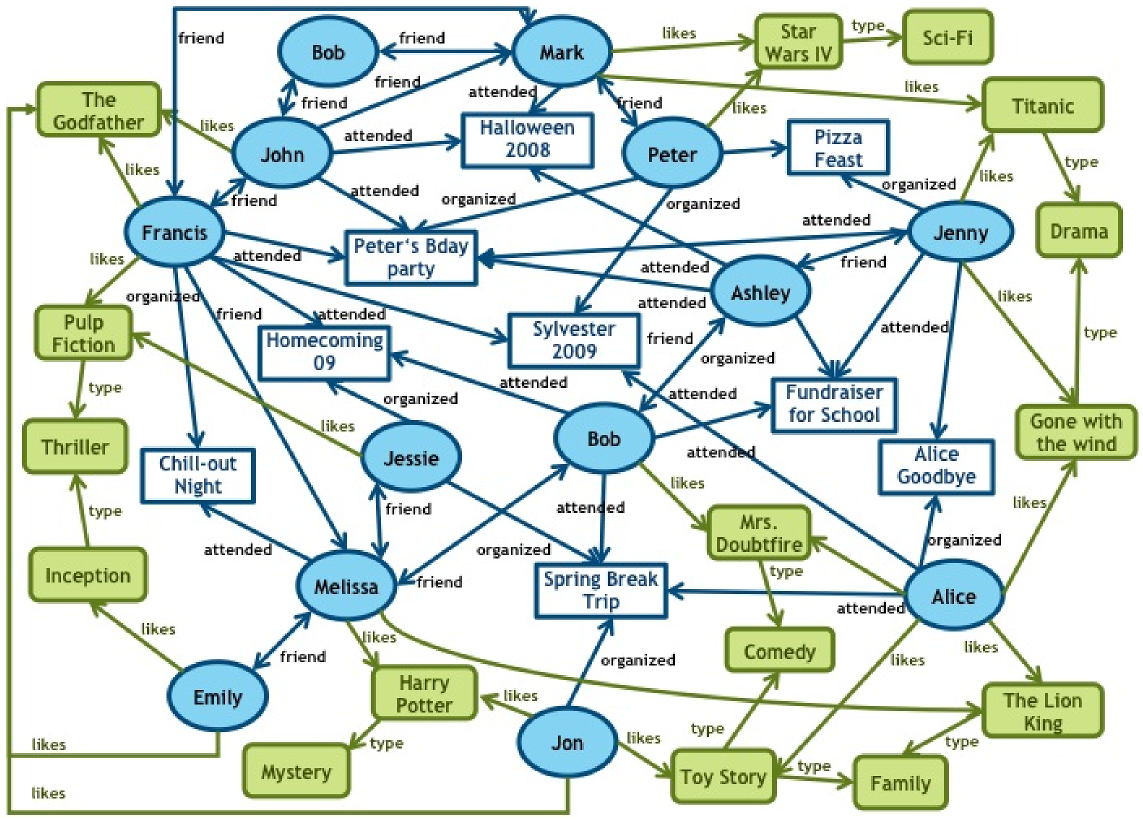


Figure 1.7: Example social network database

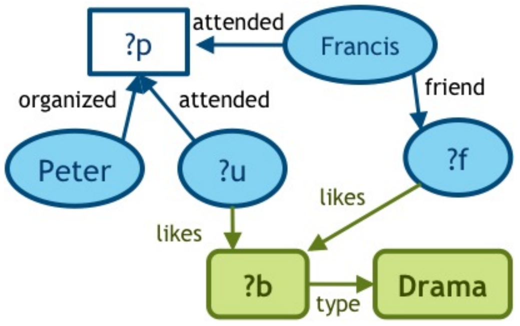


Figure 1.8: Example consumer query to social network database

1.4 Company Social Network

Many companies have implicit or explicit “internal” social networks capturing the interactions between employees, managers, customers and virtual as well as physical assets. Connections in such a social network include communications (via email, instant messenger, internal messaging board, etc.), collaborations, meetings, document modifications and other types of interactions that are recorded by company intranet and personal information management systems.

Figure 1.9 shows a small example social network representing the interactions between employees of a fictitious company. It includes employees, like "Jeff Ryser" and "Carla Bunes", and their roles in company internal teams. The social network captures the projects and deliverables these employees are working as well as customer information.

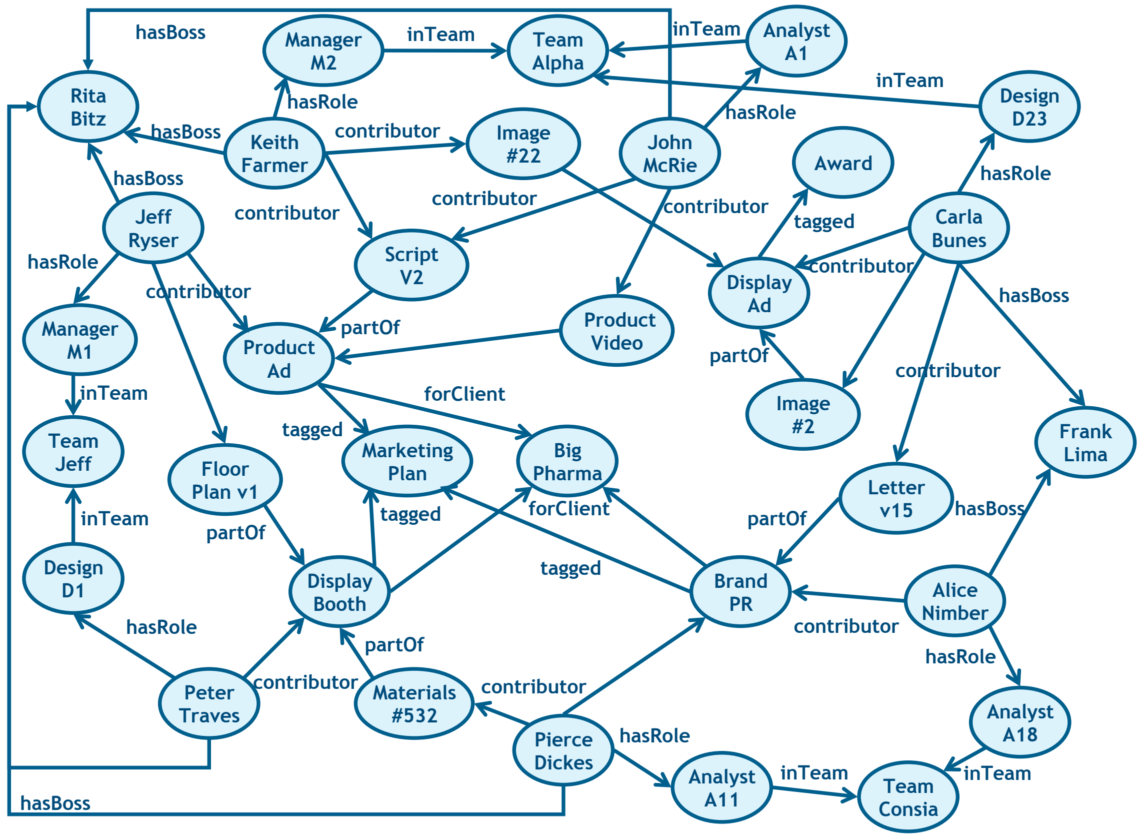


Figure 1.9: Example company social network database

Representing the internal dynamics of a company as a social network allows us to express and efficiently search for complex retrieval queries, like the one shown in Figure 1.10. The query asks for all deliverables $?v1$ to which Carla Bunes has contributed and which are part of some project $?v2$ that is tagged as a marketing plan. We also want that some unknown employee $?v3$ who is working for Rita Bitz has contributed to the project $?v2$.

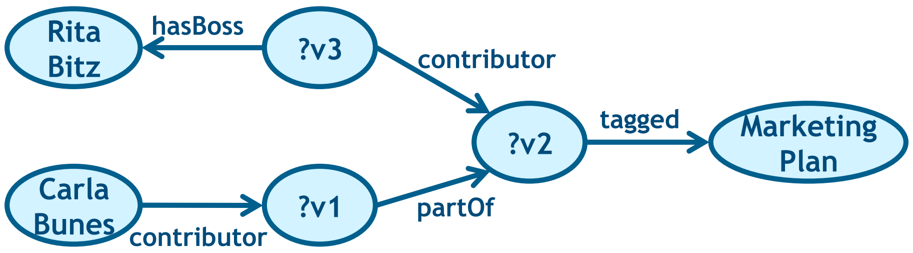


Figure 1.10: Example search query to company social network database

1.5 Other Applications

There are innumerable cases in various application domain that build upon and mirror the ideas and examples described above. We will use the four examples discussed above as motivating examples and as a running thread throughout this thesis. However, there are many other application domains where similar scenarios exist – we discuss a few of them below.

1.5.1 Public Health

A simple public health example may require all doctors and hospitals to provide some demographic information about patients who have certain symptoms (for years, certain states in the US required doctors to notify state government authorities when someone tested HIV-positive). Similar conditions apply to many more easily communicable diseases.

Let us suppose that rather than living in an Orwellian, fascist, society, we live on a more friendly planet where the goal is to prevent the spread of disease rather than ostracize and unreasonably restrict a stricken individual. In such a scenario, a country might be divided up into geographic regions (say 5km x 5km grid cells). Each such cell is a node in a graph, and the edges correspond to adjacency relations between cells. A node is usually green, but when the number of infected people reaches a certain level, the node turns amber. When the number of infections exceeds an even higher threshold, the node turns red.

Let us now suppose, momentarily, that things are slightly more complex. Two diseases are making their way through a population – the H1N1 flu virus, and pneumonia. Our nodes are labeled with two numbers – the percentage of the population of the grid cell represented by the node with H1N1 and pneumonia, respectively. This information can be obtained easily enough if doctors and health care facilities are required to report the grid cells of people with infections. A health care researcher wants to find all neighborhoods that are "at risk" of a pneumonia outbreak. He may define this formally as a query where he wants to find all nodes $?v$, such that, at least one neighbor $?n1$ has a pneumonia infection rate of over 5% and at least two neighbors $?n2, ?n3$ have H1N1 ratings of at least 10%. This may be expressed as a subgraph matching query. Many other definitions of what it means for a node to be "at risk" are obviously possible. The beauty of a general graph database framework is that each user can define whatever he thinks "at risk" should mean.

1.5.2 Marketing

Let us suppose we have a social network of bloggers (say on `blogger.com`, or across multiple blog sites). Using sophisticated opinion mining technology such as those in [11, 124], these blogs have been analyzed using natural language processing methods, and an expanded social network has been created. Each node in the expanded social network consists of either a blogger's user id or a topic on which an opinion can be expressed. There are three kinds of edges in this network:

Blogger-topic edge An edge exists from a blogger to a topic if the blogger has expressed an opinion on the topic. This edge is labeled with the average intensity of opinion assigned to the blogger's post(s) on the topic.

Follower edge An edge exists between two bloggers (or at least, between two users of the blogging system) A and B if A follows B's posts (or if A has set an alert to be notified when B posts something).

Topic relationship edge Last, but not least, edges between topic A and topic B might tell us that topics A and B overlap.

A marketer – say someone marketing Sony Cybershot cameras – wants to find all people $?P1$ who have expressed negative sentiments/opinions toward their camera, together with all followers $?P2$ of $?P1$ who have also expressed negative sentiments toward the camera in question, together with all followers $?P3$ of $?P2$. The marketer might think of such people as "hostile" territory to their marketing efforts. They may want to study the posts in further detail in order to figure out the reason for the hostility (e.g. all were treated inappropriately by service representatives at a particular camera dealership, in which case Sony might want to take appropriate corrective action). This can easily be expressed as a subgraph matching query.

1.5.3 National Security

Investigators for a national investigative agency might want to identify people who are suspected of being radicalized and prone to terrorism. For instance, consider a social network of individuals in a particular region. Each of these individuals has certain properties (e.g. age, sex, family history, economic status, travel history, etc) which may be encoded as properties of nodes in this network. In addition, some of the people in this network are known "bad guys" through some other investigative means not involving social network analysis.

The police want to identify other potential "bad guys" for further surveillance. This is a long, time honored tactic by investigative agencies. Anyone who has watched Hollywood movies involving Mafia bad guys knows that the FBI watches their spouses, relatives, girl friends and casual acquaintances, even though many of these individuals may be completely innocent of any wrongdoing. The fact that somebody crosses an investigate agency's radar in an ongoing investigation does not, by itself, offer evidence of guilt or innocence – it is just something that must be investigated further.

However, investigative agencies have limited resources. They cannot investigate everybody photographed in the same city block as a known "bad guy". They must pick and choose. Investigators for such agencies have extensive knowledge and experience in categorizing individuals they come across in an investigation. They use specific criteria to decide whether they should investigate them further. Often times, these criteria can be expressed as queries of the kind we have seen so far. For instance, if A and B are two known bad guys, and a person C has shipped a packet to A and transferred funds to B – either directly or through an intermediary D – then C might be viewed as a suspect as he has had two "professional" (as opposed to personal or familial) transactions with two known bad guys.

This can be expressed as the union of two subgraph matching queries. In the first, we are looking for persons C who have shipped a packet to A and who have transferred funds through a bank transfer directly to B. In the second query, we are looking for persons C who have shipped a packet to A and who have transferred funds through a bank transfer directly to a person D who, in turn, executed a subsequent bank transfer to bad guy B. Both of these queries are subgraph matching queries (with temporal constraints attached to them).

1.6 Conclusion

Reasoning about social networks requires a vast diversity of capabilities. For instance, in order to track diffusion through a social network, we need to look at the social network over a period of time. Many diffusion models have been built for social networks – these include diffusion models for product adoption [96], diffusion of opinion in social networks [154], diffusion of diseases through population networks [44], and many others.

There is also a significant body of work on the problem of community or group detection in social networks, where the goal is to find those subsets of vertices that constitute a community based on their connectivity structure (e.g, [58, 121, 51]).

This thesis focuses on the primitives of social network data management and addresses basic problems of data storage and retrieval that underlie many complex social network analysis technique such as the ones mentioned above. We focus on the core database technology needed to query vast social networks consisting of hundreds of millions of nodes and billions of edges. Specifically, we focus on the following basic types of queries:

Exact Subgraph Queries These queries correspond to situations where we specify a query graph structure and want to find all subgraphs of a database that exactly match the query graph. Both, vertices and edges, may be labeled to further restrict the answer set. In addition, we may have additional constraints on the variables in the query.

Inexact Subgraph Queries Of course, in many cases, nodes and/or labels may not match exactly. For instance, in a counter-terrorism application, a national investigative agency might need to perform inexact subgraph matching because the node labels (or edge labels) specified in a query may not match the content of a database exactly. A simple example is one where a user expresses a query using a node labeled "Mohammed", but the spelling listed for the individual sought is "Muhammed" or "Mohamed". In addition, there are more complex cases where a user says he thinks a certain node (e.g. Mohammed) banks at a certain bank (e.g. BCCI), but he is not sure. So matches with "Mohammeds" or "Muhammed"s or "Mohamed"s who bank at BCCI may be preferred, in accordance with some preference criteria, to those who don't. But of course, different preference criteria would determine if a "Mohammed" (exact match) who does not bank at BCCI should be preferred to a "Mohamed" (inexact match) who does bank at BCCI.

Standing Queries Standing queries (or views) are subgraph matching queries for which we want to monitor the result set. Standing queries are very useful in social network monitoring applications or for caching the result set of frequently issued queries.

Of course, there are many variants of these simple queries, many of which we shall investigate in this thesis.

Chapter 2
Formal Definition of a Social Network

In this chapter, we will formally define a social network in such a way that the examples in Chapter 1 can be expressed within this formal framework. Moreover, our definition is rich enough to account for many kinds of applications that are not explicitly discussed in Chapter 1. In addition, we will formally define a query language within which queries to social networks may be expressed.

In order to express the various applications in Chapter 1 and others that might come up in the future, we need a *very general* definition of a social network. This has the disadvantage that various specific social networks might be special cases of the very general definition – and there may often be cases where these specific social networks can be much more efficiently handled (in terms of query processing and/or view maintenance) by taking their specific features into account.

As a consequence, even though we will provide a very general definition of a social network, we will also define various *restricted classes* of social networks (i.e. social networks according to the general definition that have certain additional specific properties).

2.1 Social Networks Formalized

In its most basic form, a *directed graph* G is defined as a 4-tuple

$$G = (V, E, \mathit{start}, \mathit{end})$$

where V is a finite, non-empty set whose elements are called *vertices*, and E is a finite set whose elements are called *edges*. We assume that the vertex and edge sets are disjoint, i.e. $V \cap E = \emptyset$. We call both, vertices and edges, *graph objects*. The structure of the graph is defined by two functions on E, $\mathit{start} : E \to V$ and $\mathit{end} : E \to V$. For any edge $e \in E$, $\mathit{start}(e) = u$ is the *start vertex* of e and $\mathit{end}(e) = v$ is the *end vertex* of e. We introduce the short-hand notation $e = (u, v)$ to represent the edge e when the functions *start* and *end* are known in the current context. For instance, in Figure 1.1 there exists an edge e with $\mathit{start}(e) =$ Ann and $\mathit{end}(e) =$ Oracle which we write as $e =$ (Ann,Oracle).

Thus, a graph consists of a set V of vertices and a set E of edges that connect some of these vertices together. The above definition allows us to have multiple edges from one vertex to another. The *degree* of a vertex is the number of incident edges. The *in-degree* of a vertex $v \in V$ is defined as $\mathit{in\text{-}deg}(v) = |\{e \in E \mid \mathit{end}(e) = v\}|$, the *out-degree* is $\mathit{out\text{-}deg}(v) = |\{e \in E \mid \mathit{start}(e) = v\}|$ and the *degree* is the sum of the two $\mathit{deg}(v) = \mathit{in\text{-}deg}(v) + \mathit{out\text{-}deg}(v)$. The *neighborhoods* of a vertex $v \in V$ describe the set vertices connected to v. The *out-neighborhood* of v is defined as $\mathit{out\text{-}ngh}(v) = \{u \mid \exists e \in E : \ \mathit{start}(e) = v \ \wedge \ \mathit{end}(e) = u\}$ and, likewise, the *in-neighborhood* is defined as $\mathit{in\text{-}ngh}(v) = \{u \mid \exists e \in E : \ \mathit{start}(e) = u \ \wedge \ \mathit{end}(e) = v\}$.

We define the *neighborhood* of v as $ngh(v) = out\text{-}ngh(v) \cup in\text{-}ngh(v)$.

For *undirected graphs* we do not distinguish between start and end vertices of edges. Any two edges e_1, e_2 such that $start(e_1) = end(e_2)$ and $start(e_2) = end(e_1)$ are considered identical. In this thesis, we focus on directed graphs and therefore refer to directed graphs simply as *graphs*.

We assume a fixed and possibly infinite set of labels $\mathbb{L}$. A *property-labeled graph*, or simply *labeled graph*, is a 5-tuple $G = (V, E, start, end, \mathcal{P})$ extending the above definition of a graph by a set of labeling relations $\mathcal{P} = \{\wp_1, \wp_2, \ldots, \wp_n\}$. Each *property labeling relation* $\wp \in \mathcal{P}$ is defined as a relation $\wp \subseteq (V \cup E) \times \mathbb{L}_\wp$ where $\mathbb{L}_\wp \subseteq \mathbb{L}$ is a non-empty, fixed subset of *labels* or *attributes*. We call $\wp$ a *labeling function* iff $\wp$ is a functional relation, that is, for all $o \in V \cup E$ with $(o, l_1) \in \wp$ and $(o, l_2) \in \wp$, it follows that $l_1 = l_2$. In other words, each entity can have at most one label assigned to it under the labeling function $\wp$. We use functional notation for labeling functions, that is, we write $\wp(o) = l_1$ instead of $(o, l_1) \in \wp$. The *domain* of a labeling relation or function is defined as $\mathsf{domain}(\wp) = \{o \mid \exists l \in \mathbb{L} : (o, l) \in \wp\}$. Hence, the domain of a labeling relation $\wp$ contains all graph objects that have a label assigned by $\wp$. Similarly, the *range* defines the set of all labels assigned by $\wp$: $\mathsf{range}(\wp) = \{l \mid \exists o \in V \cup E : (o, l) \in \wp\}$.

A *vertex labeling* is a labeling relation $\wp$ such that $\mathsf{domain}(\wp) \subseteq V$. Similarly, an *edge labeling* is a labeling relation $\wp$ with $\mathsf{domain}(\wp) \subseteq E$. A labeling relation $\wp$ is *partial* if $\mathsf{domain}(\wp) \subset V \cup E$, i.e., some graph objects are not assigned a label by $\wp$, and *total* if $\mathsf{domain}(\wp) = V \cup E$, i.e., all graph objects are assigned a label by $\wp$. Similarly, we define *partial and total vertex labeling* and *partial and total edge labeling* relations restricted to the set of vertices V and set of edges E, respectively. Most of the examples in this thesis use only labeling functions and distinguish between edge and vertex labeling.

Labeling relations are a flexible and expressive means to assign arbitrary attributes to vertices and edges in a graph. The set of labels $\mathbb{L}$ might contain internal structure – for instance, we might have edges labeled by time intervals defined as $\{[x, y] \mid x, y \in \mathbb{R} \wedge x \leq y\}$. *In the following we assume that the social network database is agnostic to such internal structures, meaning elements of label sets are treated as symbols, unless we explicitly note otherwise.*

The concept of a vertex neighbhorhood can be restricted in the presence of labeling functions. The *l-neighborhood* of v contains only those neighbors of v that are connected by an edge labeled l.

All example social networks introduced in Chapter 1 are property-labeled graphs. For instance, in Figure 1.1 edges denoting stock transactions are labeled by four edge labeling functions: $\{company, quantity, price, time\}$. We see that there are two edges from Bob to Dina. One edge, e_1, is labeled with (ibm,30,112,4) – which is short-hand for: *company*(e_1)=ibm, *quantity*(e_1)=30, *price*(e_1)=112, *time*(e_1) = 4 – reflecting a sale of 30 units of IBM stock by Bob to Dina at a price of \$112 per share at time 4. The other edge, e_2, merely says that Bob is affiliated with Dina since it is labeled by the function *type*, i.e. *type*(e_2)=affiliated. Thus, the two edges have distinct labels. By the same token, had Bob sold other kinds of stock to Dina, we could insert edges from Bob to Dina labeled with details of those transactions, e.g. (Dell,100,12,5) to indicate he sold 100 units of Dell stock to Dina at time 5 at a price of \$12.

Vertices can also be labeled. In Figure 1.1, the shape of a vertex represents the type of market actor which we capture by the vertex labeling function *actor*. Joe, Lisa and Ted are all individual investors, and therefore, *actor*(Joe)= *actor*(Lisa)= *actor*(Ted)= individual-investor. Likewise, Jim, Rod and Mia are labeled as "institutional-investor" while Ellie, Mindy and Brad are labeled as "trader", and Oracle and IBM are labeled as "company".

Definition 2.1 (Social Network Database)**.** A *social network database* $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is a property-labeled graph.

Our definition of a social network database $\mathcal{S} = (V, E, start, end, \mathcal{P})$ allows for multiple edges between two vertices u, v. For instance, we may have two edges e_1, e_2 with $start(e_1) = start(e_2) = u$ and $end(e_1) = end(e_2) = v$. We say that $e_1, e2$ are *duplicates* or *identical* iff $start(e_1) = start(e_2)$, $end(e_1) = end(e_2)$, and $\forall \wp \in \mathcal{P} : \{l \,|\, (e_1, l) \in \wp\} = \{l \,|\, (e_2, l) \in \wp\}$. In other words, two edges are duplicates if and only if they have the same end points and the same set of labels for all labeling relations. Note that for unlabeled graphs, the second condition is necessarily true, because $\mathcal{P} = \emptyset$. *Throughout this thesis, we assume that all graphs and social network databases have no duplicate edges.* Graph database implementations, however, differ in their treatment of duplicate edges. Some ignore duplicate edges, others warn the user or raise errors, while some allow and store duplicate edges.

In general, property labels may be required to satisfy various integrity constraints. Declaring a labeling relation to be functional defines an integrity constraint, because it enforces that each vertex or edge has at most one label for a given labeling function. Similarly, declaring a labeling function to be total also defines a constraint. However, applications may require additional integrity constraints. For instance, in Figure 1.1, we may require that each edge labeled by the function *company* must also be labeled with *quantity*, *price*, and *time*. *Throughout most of this thesis, we assume that property labels satisfy any application specific integrity constraints.* In particular, we do not provide any specific syntax for integrity constraints at this time. We will return to this issue later in Chapter 7 – as this chapter deals with view maintenance, it is possible that an update will cause an integrity constraint violation (e.g. if we remove only the *company* label on stock transaction edge).

In the rest of this chapter, we quickly show how two of the examples of Chapter 1 can be represented formally as social networks in accordance with the above definitions. Subsequently, we will formally define social network queries – as already shown in Chapter 1, we can represent such queries as graphs as well, but with attached constraints.

2.2 The Stock Market Example as a Social Network

In order to formally represent the Stock Market Example of Figure 1.1, we first need to specify the set $\mathcal{P}$ of property labeling relations and the set $\mathbb{L}$ of labels.

- $\mathcal{P}$ consists of six labeling relations: *company*, *quantity*, *price*, *time*, *actor*, and *type*.
- $\mathbb{L} = \mathbb{L}_{company} \cup \mathbb{L}_{quantity} \cup \mathbb{L}_{price} \cup \mathbb{L}_{time} \cup \mathbb{L}_{actor} \cup \mathbb{L}_{type}$ with:
  - $\mathbb{L}_{company}$ is an enumerated set of names of various companies;
  - $\mathbb{L}_{quantity} = \mathbb{N}$;
  - $\mathbb{L}_{price} = \mathbb{R}^+$;
  - $\mathbb{L}_{time} = \mathbb{N}$;
  - $\mathbb{L}_{type} = \{$officer, board, consults, affiliated$\}$;
  - $\mathbb{L}_{actor} = \{$individual-investor, institutional-investor, trader, company$\}$.

  where $\mathbb{N}$ is the set of natural numbers and $\mathbb{R}^+$ is the set of non-negative real numbers.

Now we describe the sets of vertices and edges and the property labeling relations:

Vertices The set V of vertices is $\{$Joe, Edna, Bob, Dina, Ed, Ellie, Mia, IBM, Lisa, Brad, Steve, Jim, Ann, Oracle, Ted, SAP, Mary, Mindy$\}$.

Edges and edge labels The set E of edges along with edge labels is shown in Figure 2.1. Note that we use a short-hand notation where an edge is written as $(u,\, l,\, v)$, where u, v are vertices in V and l denotes a member of $(\mathbb{L}_{company} \times \mathbb{L}_{quantity} \times \mathbb{L}_{price} \times \mathbb{L}_{time}) \cup \mathbb{L}_{type}$.

Vertex labels The vertex labeling function *actor* assigns:

- “company” to the vertices IBM, Oracle, and SAP;
- “individual-investor” to the vertices Joe, Bob, Dina, Lisa, and Mary;
- “institutional-investor” to the vertices Ed, Mia, Ann and Jim;
- “trader” to the vertices Edna, Brad, Ellie, and Mindy.

2.3 The Cell Phone Example as a Social Network

We now formalize the Cell Phone Example of Figure 1.4.

- $\mathcal{P} = \{gender, age, marketing, network, \omega\}$.
- $\mathbb{L} = \mathbb{L}_{gender} \cup \mathbb{L}_{age} \cup \mathbb{L}_{marketing} \cup \mathbb{L}_{network} \cup \mathbb{L}_{\omega}$ where:
  - $\mathbb{L}_{gender} = \{$male, female$\}$;
  - $\mathbb{L}_{age} = \{$under-20, 20-40, 40-60, over-60$\}$;
  - $\mathbb{L}_{marketing} = \{$responded-Skype-marketing, responded-facebook-marketing$\}$;

{(Joe, officer, IBM), (Joe, (ibm,450,122,11), Mia), (Joe, affiliated, Dina),
(Edna, (ibm,150,118,2), Joe), (Edna, (oracle,1000,140,1), Brad),
    (Edna, (sap,350,48,8), Bob),
(Bob, (ibm,30,112,4), Dina), (Bob, (ibm,240,112,2), Ellie), (Bob, (ibm,50,115,6), Brad),
    (Bob, officer, IBM), (Bob, affiliated, Dina),
(Dina, (ibm,120,115,2), Ellie),
(Ed, (sap,720,135,13), Dina), (Ed, affiliated, Ellie),
(Ellie, (ibm,870,93,7), Brad),
(Mia, affiliated, Brad), (Mia, consults, IBM),
(Lisa, board, IBM), (Lisa, officer, Oracle),
(Brad, (oracle,340,115,4), Ann),
(Steve, (oracle,390,85,15), Ed), (Steve, (ibm,855,105,5), Ellie),
    (Steve, (oracle,150,158,1), Mary),
(Mindy, (ibm,2400,120,12), Steve), (Mindy, affiliated, Steve), (Mindy, affiliated, Joe),
(Mary, (sap,150,218,19), Mindy), (Mary, board, SAP),
(Jim, consults, SAP), (Jim, (ibm,110,111,3), Mary),
(Ted, officer, SAP),
(Ann, (ibm,440,84,9), Ted), (Ann, (ibm,770,83,11), Lisa), (Ann, consults, Oracle),
    (Ann, (ibm,870,93,7), Jim)}

Figure 2.1: Edges and edge labels in the Stock Database

  - $\mathbb{L}_{network} = \{\mathsf{Facebook}, \mathsf{Skype}\}$;
  - $\mathbb{L}_{\omega} = \mathbb{N}$.

- V = {Pam, John, Ina, Tina, Patty, Pete, Gail, Silvia, Liz, Pat, Lisa, Ben, Zack, Joan}.
- Set E and edge labeling functions *network* and ω are shown in Figure 2.2, where an edge is written as (u, l, v), where u, v are vertices in V and $l \in \mathbb{L}_{network}$.
- We do not explicitly list vertex labeling relations *gender*, *age*, and *marketing*, and just assume they satisfy the appropriate integrity constraints.

2.4 Social Network Queries

We now define social network queries. As discussed in Chapter 1, it is clear that there are many kinds of queries that users might want to ask. We will focus in this chapter on a basic type of query called *subgraph matching query.* Subgraph matching queries ask for all subgraphs in a social network database that match the query graph pattern. Such queries are *primitive* or *atomic* queries over network databases and form the basic building blocks of more complex queries. In later chapters of this thesis, we will address *aggregate queries*, *group-by* queries, *nearest neighbor* and *top-k* queries, all of which either directly extend subgraph matching queries or re-use the basic retrieval principles developed for subgraph matching.

$e \in E$	$\omega(e)$	$e \in E$	$\omega(e)$
(Pam, Facebook, John)	20	(Liz, Skype, Pete)	14
(Pam, Facebook, Tina)	10	(Liz, Facebook, Pat)	25
(Pam, Facebook, Gail)	25	(Liz, Facebook, Zack)	15
(John, Skype, Tina)	21	(Pat, Facebook, Ina)	8
(John, Facebook, Patty)	5	(Lisa, Skype, Tina)	7
(John, Skype, Liz)	52	(Lisa, Facebook, Gail)	41
(John, Skype, Pete)	5	(Lisa, Facebook, Silvia)	25
(John, Skype, Ina)	22	(Lisa, Facebook, Ben)	23
(Tina, Facebook, Ina)	19	(Ben, Skype, Lisa)	12
(Tina, Facebook, Zack)	9	(Ben, Skype, Zack)	9
(Tina, Facebook, Gail)	18	(Zack, Facebook, Joan)	27
(Silvia, Skype, Tina)	7	(Joan, Skype, Silvia)	15
(Silvia, Skype, Pam)	12	(Joan, Facebook, Pat)	18
(Silvia, Facebook, Patty)	34	(Patty, Facebook, Tina)	12
(Silvia, Skype, Ina)	16	(Patty, Facebook, Gail)	15
(Silvia, Facebook, Pete)	4	(Pete, Skype, Ina)	3
(Silvia, Facebook, Liz)	12	(Pete, Facebook, Pat)	31
(Silvia, Facebook, Joan)	4	(Pete, Facebook, Liz)	31
(Silvia, Skype, Zack)	14	(Gail, Facebook, Silvia)	9
(Silvia, Facebook, Ben)	15		

Figure 2.2: Edges and edge labels in the Cell Phone Database

In order to define subgraph matching queries, or SM-queries for short, we assume the existence of some infinite set VAR of variable symbols. As a convention in this thesis, we assume that VAR is the set of all alphanumeric strings that start with the symbol "?". Subgraph matching queries are defined similarly to graphs but allow for variable vertices, edges, and labels. As such, they define pattern graphs for which we seek to find all matches in a social network database.

Definition 2.2 (Subgraph Matching Query)**.** A *subgraph matching query* (SM-query for short) with respect to a social network database $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is a 5-tuple $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q)$ where:

- $V_Q \subseteq V \cup VAR$;
- $E_Q \subseteq E \cup VAR$;
- $V_Q \cap E_Q = \emptyset$;
- $start_Q : E_Q \to V_Q$ and $end_Q : E_Q \to V_Q$ are vertex functions which extend $start$ and end, i.e., $\forall e \in E$ it holds that that $start_Q(e) = start(e)$ and $end_Q(e) = end(e)$;
- $\mathcal{P}_Q = \{\wp_Q \mid \wp \in \mathcal{P}\}$ is a set of *query labeling relations* such that, for each labeling relation $\wp \in \mathcal{P}$, there is an associated query labeling relation $\wp_Q \in \mathcal{P}_Q$ such that $\wp_Q \subseteq (V_Q \cup E_Q) \times (\mathbb{L}_\wp \cup VAR)$.

We see that the definition of a subgraph matching query Q is almost identical to that of a social network database $\mathcal{S}$, with the main difference that queries allow variable symbols. Given an SM-query Q, we use VAR_Q to denote the set of all variables in Q and $\mathbb{L}_Q = \bigcup_{\wp_Q \in \mathcal{P}_Q} \mathsf{range}(\wp_Q)$ to denote all the labels and label variables in Q. We distinguish between the set of *vertex variables* $V_Q \cap VAR$, the set of *edge variables* $E_Q \cap VAR$, and the set of *label variables* $\mathbb{L}_Q \cap VAR$. We assume those variable subsets to be disjoint for any given query. The *constant vertices* are those in $V_Q \setminus VAR$.

A *substitution* formally defines a "match" between an SM-query and a social network database. Suppose $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is a social network database and $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q)$ is a SM-query. A *substitution* is a mapping $\theta : VAR_Q \to V \cup E \cup \mathbb{L}$. It substitutes variables in the query for vertices, edges, and labels in $\mathcal{S}$. The application of a substitution θ to an SM-query Q, denoted $Q\theta$, is the result of replacing all variables x in Q by $\theta(x)$, that is, applying θ to all vertex, edge, and label variables. We can restrict substitutions to vertex, edge and label substitutions as follows: $\theta_V : V_Q \cap VAR \to V$ denotes the *vertex restricted* substitution of θ, i.e. $\theta(v) = \theta_V(v)$ for all $v \in V_Q \cap VAR$, and similarly $\theta_E : E_Q \cap VAR \to E$, $\theta_{\mathbb{L}} : \mathbb{L}_Q \cap VAR \to \mathbb{L}$ denote the *edge and label restricted* substitutions of θ respectively.

Definition 2.3. Suppose $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is a social network database and $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q)$ is a SM-query. A substitution θ is an *answer substitution* for Q iff:

- $Q\theta$ is a subgraph of $\mathcal{S}$, that is, $\forall e \in E_Q$ it holds that $start(e\theta) = start_Q(e)\theta$ and $end(e\theta) = end_Q(e)\theta$;
- Labels substitutions match, that is, $\forall \wp_Q \in \mathcal{P}_Q$, $\forall (x, l) \in \wp_Q$ it holds that $(x\theta, l\theta) \in \wp$.

The *answer set* for an SM-query Q is the set $\{Q\theta | \theta$ is an answer substitution for $Q\}$.

2.4.1 Constrained Queries

Next, we extend the definition of a subgraph matching query by introducing the notion of a *query constraint* so that the constraints shown informally in the examples in Chapter 1 are special cases of the general definition of a query constraint. A query constraint requires that some condition must hold for any valid substitution.

To formally define the syntax and semantics of query constraints, we first need to introduce some auxiliary definitions. We assume that a query Q and social network database $\mathcal{S}$ have been fixed.

First, we assume the existence of a set of function symbols $\mathcal{F}$. Each function symbol $f \in \mathcal{F}$ has an associated arity, $arity(f) \in \mathbb{N}$. A *simple term* t is either a vertex, edge, label or variable in Q, i.e., $t \in V_Q \cup E_Q \cup \mathbb{L}_Q \cup VAR_Q$. A *term* is either a simple term or an expression of the form $f(t_1, \ldots, t_m)$ where $f \in \mathcal{F}$ is a function symbol with $arity(f) = m$ and $t_1, \ldots, t_m$ are terms. A term is *ground* if it does not contain variables. Each function symbol $f \in \mathcal{F}$ with arity m has a *denotation* $[\![f]\!]$ defined as $[\![f]\!] \subset (V \cup E \cup \mathbb{L})^m \rightarrow (V \cup E \cup \mathbb{L})$. If t is a ground term, then the denotation of t, $[\![f]\!]$, is equal to t if it is a simple term or, if $t = f(t_1, \ldots, t_m)$ is a function term, given by $[\![t]\!] = [\![f]\!]([\![t_1]\!], \ldots, [\![t_m]\!])$.

We also assume the existence of a set $\mathbf{P}$ of predicate symbols. Each predicate $p \in \mathbf{P}$ has an associated arity, $arity(p) \in \mathbb{N}$, and a *denotation* $[\![p]\!]$ defined as $[\![p]\!] \subset (V \cup E \cup \mathbb{L})^n$ when $arity(p) = n$.

Definition 2.4 (Query Constraint)**.** A *query constraint formula* is defined as follows:

- if $p \in \mathbf{P}$ is a predicate symbol with arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a query constraint formula;
- if C_1 and C_2 are query constraint formulas, then so is $C_1 \wedge C_2$.

A query constraint formula is *ground* iff all of its terms are ground. A ground query constraint formula C is *true* iff any of the following conditions holds:

- $C = p(t_1, \ldots, t_n)$ and $([\![t_1]\!], \ldots, [\![t_n]\!]) \in [\![p]\!]$;
- $C = C_1 \wedge C_2$ and both C_1 and C_2 are true.

Examples of query constraints are readily visible in Figures 1.2 and 1.3. For instance, in Figure 1.3, we see the query constraint $(?T3 - ?T1) < 5 \wedge (?T2 - ?T1) < 5$. Note that we use infix rather than prefix notation for function $-$ and predicate $<$. In prefix notation, the constraint would be written as $<(-(?T3, ?T1), 5) \wedge <(-(?T2, ?T1), 5)$. We will always use infix notation in our examples to ease comprehension.

To keep the definitions simple, query constraints are conjunctive formulas and we do not distinguish between different types or sorts of terms. However, one can easily extend the above definitions to accommodate more complex constraints such as disjunctions, or introduce type checking.

Definition 2.5 (Constrained Subgraph Matching Query)**.** A *constrained subgraph matching query* is a 6-tuple $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q, C_Q)$ which extends a standard subgraph matching query by an additional constraint formula C_Q. A substitution θ is an *answer substitution* for Q iff it is an answer substitution to Q and $C_Q\theta$ evaluates to true.
The *answer set* for Q is the set $\{Q\theta \mid \theta \text{ is an answer substitution for } Q\}$.

As an example, let us consider the query Q shown in Figure 1.3 with respect to the stock market database shown in Figure 1.1. The query contains 6 vertex variables, 1 constant vertex, 9 label variables, and 6 edge variables. In most cases, the user is not interested in the actual edge variable substitutions, which is why we omit edge variables from the figures and also from answer substitutions, shown in Table 2.1.

Table 2.1: Answers for the query in Figure 1.3 against the Stock Database

θ_1	$?a$	$?b$	$?Q2$	$?P2$	$?p2$	$?T2$	$?c$	$?Q3$
	Joe	Dina	120	115	Ellie	12	Mindy	2400
	$?P3$	$?T3$	$?P3$	$?p1$	$?Q1$	$?P2$	$?T1$	
	120	12	Steve	Mia	450	122	11	
θ_2	$?a$	$?b$	$?Q2$	$?P2$	$?p2$	$?T2$	$?c$	$?Q3$
	Joe	Mindy	2400	120	Steve	12	Dina	120
	$?P3$	$?T3$	$?P3$	$?p1$	$?Q1$	$?P2$	$?T1$	
	115	12	Ellie	Mia	450	122	11	

Though these two answer substitutions are different, they in fact lead to just one answer, namely the subgraph shown in Figure 2.3, since $Q\theta_1$ and $Q\theta_2$ are isomorphic. This is the reason why we distinguish between answer substitutions and query answers.

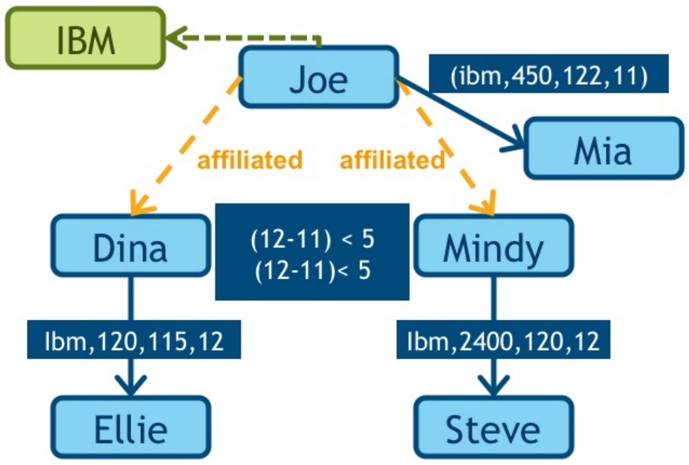


Figure 2.3: Answer to the query of Figure 1.3 against the Stock Database

Let us now consider the cell phone social network example of Figure 1.4 and consider the query shown in Figure 1.5. As in the previous case, there are two answer substitutions for this query, θ_1 and θ_2 shown in Table 2.2 – but they both lead to the same instance of the query graph.

Table 2.2: Answers for the query in Figure 1.5 against the Cell Phone Database

θ_1	$?B$	$?C$	$?D$	$?M1$	$?M2$	$?T1$	$?T2$
	Tina	Liz	Zack	21	52	9	15
θ_2	$?B$	$?C$	$?D$	$?M1$	$?M2$	$?T1$	$?T2$
	Liz	Tina	Zack	52	21	15	9

2.5 Simple Graphs

Our definition of a basic graph and social network defines edges as entities in their own right and introduces the two edge connection functions *start* and *end*. This non-conventional definition supports parallel edges and complex edge labelings – both, in particular the latter, occur frequently in real social network databases. However, in many contexts, simpler formalizations suffice, and our discussion can be greatly simplified by using specific restricted definitions. Hence, in this section, we introduce restricted cases called *simple graphs*.

Simple Type-Only Graphs

Typed social networks are a frequently occurring restricted class of social network databases. A labeled graph $G = (V, E, start, end, \mathcal{P})$ is *typed* iff $\mathcal{P}$ contains an edge labeling function $type \in \mathcal{P}$ which is a total function on E mapping onto a fixed and finite set of type labels denoted $\mathbb{T} \subset \mathbb{L}$, i.e. $type : E \to \mathbb{T}$.

All of the example social networks presented in Chapter 1 have typed edges. Edge types typically encode the semantics of an edge and are used in almost all social network applications. For instance, in Figure 1.7 the edge e_1 = (John, Mark) has type $friend$, denoting that John is a friend of Mark. Similarly, the edge e_2 = (Jenny, Titanic) is typed *likes* to denote that Jenny likes Titanic. Hence, $type(e_1)$=friend and $type(e_2)$=likes.

Since types are used very frequently, we extend our short-hand notation for edges to triple notation for typed networks, e =(u, t, v), where t denotes the edge type. Hence, we write e_1 = (John, friend, Mark) and e_2 = (Jenny, likes, Titanic). Moreover, for any type $t \in \mathbb{T}$ and vertex $v \in V$, $out\text{-}ngh_t(v) = \{u \mid \exists e \in E : start(e) = v \ \wedge \ end(e) = u \ \wedge \ type(e) = t\}$ denotes the neighborhood of vertex v *restricted to type* t. Likewise, $in\text{-}ngh_t(v) = \{u \mid \exists e \in E : \ start(e) = u \ \wedge \ end(e) = v \ \wedge \ type(e) = t\}$ and $ngh_t(v) = out\text{-}ngh_t(v) \cup in\text{-}ngh_t(v)$.

A *simple type-only graph* is a graph where *type* is the only labeling function. Some of the following chapters focus entirely on type-only graphs. Considering only type labeling functions simplifies the presentation of algorithms and query answering strategies which can easily be extended to arbitrary labeling relations. However, *type* is arguably the most important as it is used for selectivity estimation and central to most optimization strategies.

One important use case for type-only graphs is the resource description framework (RDF) [88]. RDF is endorsed and maintained by the World Wide Web Consortium as the standard format for data interchange on the Web. Through the Linked Data movement[1] and government open data initiatives, RDF has seen wider adoption in recent years. A significant number of datasets, from census data to biological knowledge, is now available in RDF which provides data interoperability and eases access to data. Any RDF dataset can be represented as a type-only graph with unlabeled vertices. In RDF, vertices are either uniform resource identifier (URIs), blank

[1]For more information see http://www.linked-data.org

node identifiers, or attribute literals, all of which are uniquely identified. Edges are called "triples" in RDF and they are commonly represented using the triple notation $e = (u,\, t,\, v)$, since every edge has a type and no other labels associated with it. Finally, $\mathbb{T}$ – the set of type labels – is a set of uniform resource identifiers.

Simple Weighted Graphs

In many scenarios, edge *weights* are used to denote, e.g., the strength of association between the vertices connected by the edge. A labeled graph $G = (V, E, start, end, \mathcal{P})$ is *weighted* iff $\mathcal{P}$ contains a total edge labeling function $\omega : E \to \mathbb{R}$. For instance, Figure 1.4 shows a weighted graph where the edge weights are given by the edge annotations. The edge $e =$ (Pete, Ina) has an edge weight of 3, i.e., $\omega(e) = 3$. Given any two vertices $u, v \in V$ we define $\omega(u, v) = \sum_{e \in E : start(e) = u \wedge end(e) = v} \omega(e)$. The range of the weight labeling function ω is a subset of the real numbers which allows manipulation by arithmetic operations.

Throughout this thesis we rely on concepts, notions and approaches from standard graph theory applied to weighted graphs without parallel edges and labels. Therefore, we define a *simple weighted graph* $G = (V, E, \omega)$ as composed by a set of vertices V, a set of edges $E \subseteq V \times V$, and a weight function $\omega : E \to \mathbb{R}$. A simple weighted graph may be undirected or unweighted. Given a simple weighted graph $G = (V, E, \omega)$, the equivalent labeled graph is $\tilde{G} = (V, \tilde{E}, start, end, \{\tilde{\omega}\})$ such that $(u, v) \in E \Leftrightarrow \exists e \in \tilde{E} : start(e) = u \wedge end(v)$ and $\tilde{\omega}(e) = \omega((start(e), end(e))$.

Chapter 3
Storing Social Networks

This chapter reviews some approaches to social network data management and querying that have been proposed in the literature and implemented in commercial database systems. Faced with the increasing popularity of social networks, researchers and industry practitioners have recognized the need to build database systems tailored to the data model and access patterns for graph structured data.

Many approaches to social network data management are backed by relational database systems. They either provide a logical abstraction layer on top of an RDBMS and translate SM-queries into SQL or utilize relational index structures paired with custom query answering algorithms. We review relational approaches to social network data management in Section 3.1.

Recently, native social network database systems have been introduced. Native social network stores implement their own storage backend and provide an entire database stack centered around the graph data model. We review native social network databases in Section 3.2.

The focus of this thesis is on managing large social network datasets where the entire database comprises one massive graph. However, the term *graph data management* also encompasses another effort, namely that of managing a database which contains *multiple* smaller graphs. Such a *database of graphs* may contain the graphical structures of many protein molecules. We review the literature on databases of graphs in Section 3.3, but note, that the problem of as well as the solutions to managing many small graphs are different from managing one massive social network.

We close this chapter with a discussion of XML databases and object oriented database systems (ODBMS) in Section 3.4. XML databases store and retrieve tree-structured documents. Object oriented databases provide a persistence layer for programming data by exposing an interface layer that is tailored to object oriented programming languages.

Our review of existing social network databases focuses on transactional, disk-resident databases. As social network data grows to billions of edges it becomes necessary to store data in external memory and to develop query mechanisms that quickly retrieve the requested data from disk. However, there is a large body of work on in-memory graph databases which focuses on graph data that can fit entirely in memory. In order to push the limits on the size of networks that can fit into memory, techniques for compressing graph structured data have been proposed (e.g. [95, 108, 32]). Most applications of in-memory graph databases are for social network mining which is beyond the scope of this thesis.

3.1 Relational Social Network Storage

Relational database systems have become the de-facto standard database management system due to their simple and flexible data model, an expressive, standardized query language (SQL), and robust, commercial-grade implementations. The most common approach to storing graph data in a relational model utilizes two tables: one for vertex information and one for the edges. The vertex table uses an integer identifier for each vertex as its unique key and additional columns for vertex labels. The edge table has one column for the start and one for the end vertex of each edge which are foreign keys to the vertex table. If multiple edges are allowed, each edge has an integer identifier as its unique key as well. Edge labels are captured using additional columns.

Example 3.1. Consider the consumer social network shown in Figure 1.7. This network can be stored in a relational database using a vertex and edge table. We show a fragment of those two tables in Figure 3.1.

Vertex Table

ID	Label
1	Bob
2	John
3	The Godfather
4	Halloween 2008
5	Francis
6	Pulp Fiction
...	

Edge Table

ID	Start ID	End ID	Type
1	1	2	friend
2	2	1	friend
3	2	3	likes
4	2	4	attended
5	5	6	likes
6	5	3	likes
...			

Figure 3.1: Relational tables for the social network in Figure 1.7

Many graph database systems, such as Jena [157], Sesame [24], Rstar [93], Oracle 11g's Network Data Model [141], DB2's RDF store [94], and RDFBroker [138], map graph data onto a relational model in a manner similar to the one described above but use different approaches to deal with the schema flexibility of social networks. Relational databases require that the user specifies a schema for the data initially. The schema can be changed later, but any such change (e.g. adding or removing a column) is typically very expensive when data has already been loaded. Social networks, on the other hand, have a flexible schema[1] which changes throughout the lifetime of the database. Recall that in Chapter 2 we defined labeling relations to be *partial*, that is, some vertices and edges have certain labels while other don't. We could create one column for each labeling relation, however, this would inevitably lead to many *null* values in the respective columns which wastes storage space and results in decreased performance.

[1]Some people use the term *schema-free* instead.

```
SELECT p.label, f.label, u.label, b.label FROM
 Vertex cf INNER JOIN Edge e1 ON cf.id=e1.start
INNER JOIN Vertex f ON e1.end=f.id
INNER JOIN Edge e2 ON e2.start=f.id
INNER JOIN Vertex b ON e2.end=b.id
INNER JOIN Edge e3 ON e3.start=b.id
INNER JOIN Vertex cd ON e3.end=cd.id
INNER JOIN Edge e4 ON e4.end=b.id
INNER JOIN Vertex u ON e4.start=u.id
INNER JOIN Edge e5 ON e5.start=u.id
INNER JOIN Vertex p ON e5.end=p.id
INNER JOIN Edge e6 ON e6.end=p.id
INNER JOIN Vertex cp ON e6.start=cp.id
INNER JOIN Edge e7 ON e7.end=p.id
WHERE e7.start=cf.id AND cf.label = ‘‘Francis’’ AND cd.label=‘‘Drama’’
    AND cp.label=‘‘Peter’’ AND e1.type=‘‘friend’’ AND e2.type=‘‘likes’’
     AND e3.type=‘‘type’’ AND e4.type=‘‘likes’’ AND e5.type=‘‘attended’
    ’ AND e6.type=‘‘organized’’ AND e7.type=‘‘attended’’
```

Figure 3.2: SQL query for the SM-query in Figure 1.8

Some graph to relational model mappings, including Jena and Sesame, introduce another table for labeling relations to avoid having many sparse columns. While this reduces the storage space, it requires another join between the vertex (resp. edge) table and the labeling tables when answering queries involving labels. In contrast, RDFBroker attempts to “discover” the schema in the social network by analyzing the signature of vertices defined as the set of labels and edge types incident on it. It then partitions the vertex and edge tables accordingly and builds separate tables based on the discovered signatures. While this approach avoids sparse columns and additional joins, it only works on static datasets since updates to the social network database can cause changes in the signatures.

A more significant shortcoming of the graph to relational model mapping is exposed during query answering. Mapping SM-queries to SQL queries is straight forward, but requires many joins with the vertex and many self-joins on the edge table, which is very expensive. Consider the query Q on the consumer social network shown in Figure 1.8. Assuming the graph to relational mapping outlined in the example above, Q can be expressed as the SQL query shown in Figure 3.2 (for detailed description of the conversion process, see [28]). From Figure 3.2 it is readily visible that the simple query Q results in a long and complex SQL query with **13 joins**. It is well known that joins are an expensive relational operation and, moreover, query optimizers run into difficulties when attempting to find optimal query plans with many joins [27].

To speed up the join processing, most systems aggressively index the vertex and edge tables. That is, secondary indexes are installed for all pairs of columns in the edge table and the individual columns in the vertex table. Maintaining all the indexes as well as the table is expensive during edge insertion. To reduce the storage requirements of the system and to save time on updates, *covering indexes* have been widely adopted.

Before we introduce covering indexes, consider the case of type-only graphs (e.g. RDF) introduced in Section 2.5, which is the data model adopted by most semantic web database systems (also called "triple stores"). In that case, there is a single labeling relation "type" for the edges in the social network as shown in our example. If we furthermore disallow multiple edges between pairs of vertices, the edge table only has three columns: start, end, type – which we abbreviate by S,E,T, respectively. A *covering index* is an index built over all three columns of the edge table, in other words, it "covers" the table. For a covering index, the order of the columns matters. For instance, we write STE to denote the index which indexes the edges first by the start id, then edge type, and finally the end id. There are 6 possible covering indexes for the 3 columns given by the distinct permutations of S,E, and T: SET, STE, EST, ETS, TES, and TSE. Since the covering indexes each contain all the information about a referenced edge, there is no need to access the edges table after an index retrieval, which saves time. In fact, there is no need to maintain the edge table at all.

Semantic Web database systems like Hexastore [155], Yars [66], RDF-3X [110, 111], Virtuoso [43], Jena TDB [118], BigData [120], 4store [65], and others adopt the idea of covering indexes.

Hexastore, Yars, and Jena TDB maintain all six index structures as B-trees. Using B-trees has the advantage that the index entries are maintained in sorted order. Consider the index structure STE. Suppose we want to look up all friends of the vertex "Francis". After retrieving the id of Francis (5), we can use the B-tree to retrieve all index entries that have the prefix (5,friends), returning the ids for all of Francis' friends. Furthermore, the sort order is *significant* – in the sense of traditional relational databases – as it can be exploited during query answering. To intersect two lists of friends, both of which are maintained in sorted order, one can use the standard merge-join algorithm which has linear time complexity and preserves the order.

RDF-3X optimizes the covering B-tree indexes in several ways. Firstly, the authors use a technique similar to B-tree page prefixes in that they store the increment between subsequent ids in a B-tree leaf page instead of the actual id. Since increments are typically smaller, this technique reduces the storage requirement and allows to fit more data on each disk page, thereby improving retrieval performance. Additionally, they compress the B-tree pages for additional space savings.

Virtuoso uses compressed bitmap indices instead of plain numeric ids in the B-tree leaf nodes. In our example, Francis has four friends in the network, which means there would be 4 entries with the prefix (5,friends) in the STE B-tree index. Instead of actually storing the numeric ids for all 4 friends in sorted order, Virtuoso stores a compressed bitmap index over all vertices in the network where only the bits corresponding to Francis' 4 friends are set to 1. The uncompressed bitmap index would require $O(|V|)$ storage space, but since the network is sparse, compressing the index is very efficient. Similar to RDF-3X, compressed bitmap indexes are particularly useful when the social network exhibits strongly connected subnetworks over vertices with increasing vertex ids. Furthermore, compressed bitmaps can be efficiently processed during query answering.

4store does not use B-trees to store the covering indexes but radix tries. 4store builds one radix trie per edge type so that the index structure only stores pairs of vertex ids. A radix trie has lookup time complexity $O(k)$ where k is the index key length. Since both vertex ids are 8 bit integers the key length is short. While this makes lookups like "find all friends of Francis" fast because only one radix trie is involved, finding all of Francis' connections requires lookup in all radix tries.

Finally, BigData adopts the standard covering indexes approach, but uses a distributed B-tree index infrastructure to distribute the data across multiple machines and achieve better read and write performance.

Instead of building multiple B-trees, some researchers have proposed the use of column-oriented database systems (e.g. [144, 15]) to store type-only graph structured data. Column-oriented databases store relational tables by column rather than by row. While row-oriented access to the data is beneficial when retrieving individual entries, column-oriented databases have been shown to yield better performance on analytic queries, such as computing the sum of all attribute values for a table. To store graph structured data in a column-oriented database, Abadi et al. [1] have the same approach as 4store in that they decompose the edge table into separate tables for each edge type. However, they store those edge tables by column in a column-database in sorted order. This allows quick edge retrieval when the edge type and one end point is known and also facilitates join processing due to the ordering, but it suffers the from same shortcoming as 4store in that multiple index retrievals are necessary when a retrieval operation does not specify the edge type.

3.2 Native Social Network Databases

There has been some foundational work on native graph data models and corresponding query languages (see, e.g., [8, 9, 130]). To date, this line of work has been theoretical. No specific implementations or suitable index structures have been developed. SPath [166] is a native graph index structure based on paths to quickly answer subgraph matching queries. SPath indexes paths up to length l in the social network, where l is user specified. SM-queries can be decomposed into a collection of (possibly) overlapping path queries which are executed against the SPath index. SPath demonstrated very fast query response times when the index is kept in memory. However, constructing the index is expensive. The experimental data published by the authors suggests that index construction time grows cubically in the size of the network even when the network is sparse. On a network with 10 million edges and 2 million vertices, constructing an SPath index took more than 20 hours on modern hardware. Hence, building a path based index is clearly infeasible for real-world social networks. In that regard, SPath demonstrates a general scalability limitation of frequent substructure based approaches to graph indexing when applied to massive social networks.

Neo4j[2] is an open-source graph database with a native storage backend [129].

[2]`http://neo4j.org/`

Neo4j provides an entire database stack geared toward graph data management. Neo4j supports arbitrary vertex and edge labeling relations. It stores labels and edges in separate repositories. Edges are stored in an appendable store using pointers to link to previous edges in the adjacency list. While this storage model provides very fast write access to the database, it leads to slow read access when edges are not inserted consecutively, because many disk pages need to be retrieved. Furthermore, Neo4j does not support returning only a subset of edges for a user provided edge type. In addition, Neo4j currently does not provide efficient subgraph matching algorithms. In a recent comparison [23], we showed that the algorithm presented in Chapter 5 outperformed Neo4j by orders of magnitude.

3.3 Database of Graphs

There has been a tremendous amount of work on database systems that store many small graphs (e.g. [86, 59, 159, 70, 160, 68, 30, 169, 48, 82]). The most prominent use cases are found in the life sciences where researchers utilize such databases to store and query molecular structures using subgraph matching queries. Users store entire graph objects in the database (like molecules) and subgraph matching queries are used to retrieve all graphs from the database that contain the given subgraph. We refer to such database systems as *databases of graphs* to distinguish them from *graph databases*, which store a single, large graph.

Most databases of graphs build index structures over graph features, such as paths or subgraphs, which are used to prune the search at query time. Given a subgraph matching query Q, the query processor extracts all relevant graph features from Q and utilizes the index structures to locate those graphs in the database that contain all of these features. Then, a subgraph isomorphism algorithm is used to match Q against each of the returned graphs to check which ones are indeed query answers. This general framework is commonly referred to as *filter-then-verify*. The index structures are used to *filter* the graphs in the database down to those that contain all of the query's features and then it is *verified* whether the query is actually a proper subgraph for each of the retrieved graphs.

The various database of graph systems differ in their feature selection and subgraph isomorphism algorithms. We do not review individual filtering and verification strategies here (see, e.g., [131] for a recent survey) but discuss why the techniques developed for databases of graphs are not applicable to massive social network databases, which is the focus of this thesis.

Filtering Obviously, filtering itself is not very useful when executing subgraph matching queries against a single, large social network. Moreover, the features used by databases of graphs, like frequent subgraphs, frequent paths, vertex signatures and other graph substructures, cannot be efficiently computed for large social networks. Graphs stored in databases of graphs are typically on the order of 100s to 1000s of vertices, and therefore computing all feature occurrences is efficient. However, the same computation is infeasible on large social networks with millions to billions of vertices. Moreover, the feature occurrences are likely to be too numerous to persist in an index.

Verification The verification step uses subgraph isomorphism checking algorithms to determine whether a particular graph is an answer to the query. Most subgraph isomorphism testing algorithms are based on the original SD algorithm proposed by Ullman [151]. The algorithm expands an initially empty match one vertex at a time while ensuring that all edge constraints are met of neighboring vertices to guarantee that a proper subgraph is found if such exists. The graphs stored in a database of graphs are on the order of 100s to 1000s of vertices, hence the entire graph can be held in memory when executing the subgraph matching isomorphism (queries are even smaller and are kept in memory as well). When the entire graph is in memory, more efficient subgraph matching isomorphisms can be used that use aggregate counts of vertex neighborhoods to prune the search further (e.g. [36, 100]). In addition, some databases of graphs (such as [30]) reuse the retrieved features to prune the search further.

In the context of one massive social network, recent advances in subgraph matching isomorphisms are not applicable because the social network cannot be held in memory. When the graph is not in memory, heuristics, such as one-step look ahead, are too expensive to compute because they would require additional disk access in order to retrieve the larger neighborhood of a vertex. We will discuss subgraph matching strategies for massive social networks in more detail in the following chapters.

3.4 XML Databases and Object Oriented Databases

XML documents follow a tree-structured data model and have a flexible schema similar to social network databases. For that reason, XML documents are also referred to as *semi-structured* data. In the previous decade, there has been a lot of work on database systems for XML documents. The problem is similar to that of databases of graphs in that an XML database contains multiple XML documents, each of which is represented by a tree. XML databases, such as Lore [4], pioneered the idea of path index structures (see also [101, 34]). For instance, DataGuides [61] are summaries of the path structures found in an XML database and used at query time to efficiently locate all documents that match a path query expressed in XQuery [14]. However, as discussed in Section 3.2, path based index structures are expensive to build for massive social network database due to large number of paths that exist in networks with millions of nodes and billions of edges.

Object oriented databases (ODBMS) were developed in the 1990s to simplify persistence of data in object oriented programming languages that were becoming increasingly popular (see [163, 10] for an overview). ODBMS support storing and querying entire object graphs and therefore bear resemblance to graph databases.

However, the link graph for objects only provides a limited graph model and the access primitives to ODBMS do not support graph operations. Secondly, ODBMS do not support subgraph matching queries and are not optimized for the access patterns of SM-queries. Nevertheless, proponents of ODBMS have argued that they are readily applicable to social network data management. Recently, Objectivity Inc. [3] has released a social network database called InfiniteGraph [4] based on their object oriented database.

3.5 Conclusion

All the approaches to social network data management reviewed in this chapter are *edge-centric*, that is, they focus on storing individual edges in one or multiple index structures and retrieving sets of edges that match some condition. Such approaches lead to poor data locality, because the edges can be distributed across multiple indexes on disk which leads to costly retrieval operations. In the next chapter, we introduce a disk-based index structure for social network data which analyzes the graph structure of the data to achieve higher data locality and therefore faster query response times.

[3] `http://www.objectivity.com/`
[4] `http://www.infinitegraph.com/`

Chapter 4
Disk-based Storage and Retrieval of Social Networks

An important performance metric for any database index structure is the number of disk accesses required to retrieve the data needed to answer a query. Reducing the number of disk accesses has a significant impact on the overall query time, since access to external memory is orders of magnitude slower than main memory access. To reduce the number of disk retrievals, one needs to ensure *data locality*, that is, the data needed to answer the query is stored on the same disk page or a page that is physically close on disk. To answer SM-queries, we need to retrieve subgraphs from the social network database. Hence, ensuring data locality for SM-queries means storing connected subgraphs on the same disk page. This, intuitively, is the idea behind the DOGMA index introduced in this chapter.

DOGMA[1] is a *graph-based* index for social network databases that aims to maximize the locality of data by employing concepts from graph theory. Greater data locality allows us to answer SM-queries faster using an efficient answering algorithm tailored to the index structure. DOGMA is tuned for scalability in several ways. First, the index itself can be stored on disk. This is very important. From experiences in relational database indexing, it is clear that when the data is large enough to require disk space, the index will be quite large and needs to be disk resident as well. DOGMA, defined in Section 4.1, is the first graph-based locality index for social networks that we are aware of that is specifically designed to reside on disk. In contrast to the graph database systems reviewed in the previous chapter, DOGMA stores entire subgraphs in close proximity on disk.

In the following, we define the DOGMA data structure and develop an algorithm to take an existing social network dataset and create the DOGMA index for it. In Section 4.2, we develop algorithms to answer subgraph matching queries. Our first algorithm, called DOGMA_basic, uses the index in a simple manner. Subsequently, we provide the improved algorithm DOGMA_adv and two extensions of the index called DOGMA_ipd and DOGMA_epd, that use sophisticated pruning methods to make the search more efficient without compromising correctness. Finally, in Section 4.3, we show the results of an experimental assessment of our techniques against four competing RDF database systems: JenaTDB, Jena2, Sesame2, and OWLIM. We show that DOGMA performs very well compared to these systems.

4.1 The DOGMA Index

In this section, we develop the DOGMA index to efficiently answer SM-queries over social network databases $\mathcal{S} = (V, E, start, end, \mathcal{P})$ in situations where the index itself must be very big (which occurs when $\mathcal{S}$ is very big). Throughout this chapter, we assume $\mathcal{S}$ to be a fixed social network database. Before we define DOGMA

[1] DOGMAstands for *disk oriented graph management*

indexes, we first define what it means to merge two graphs.

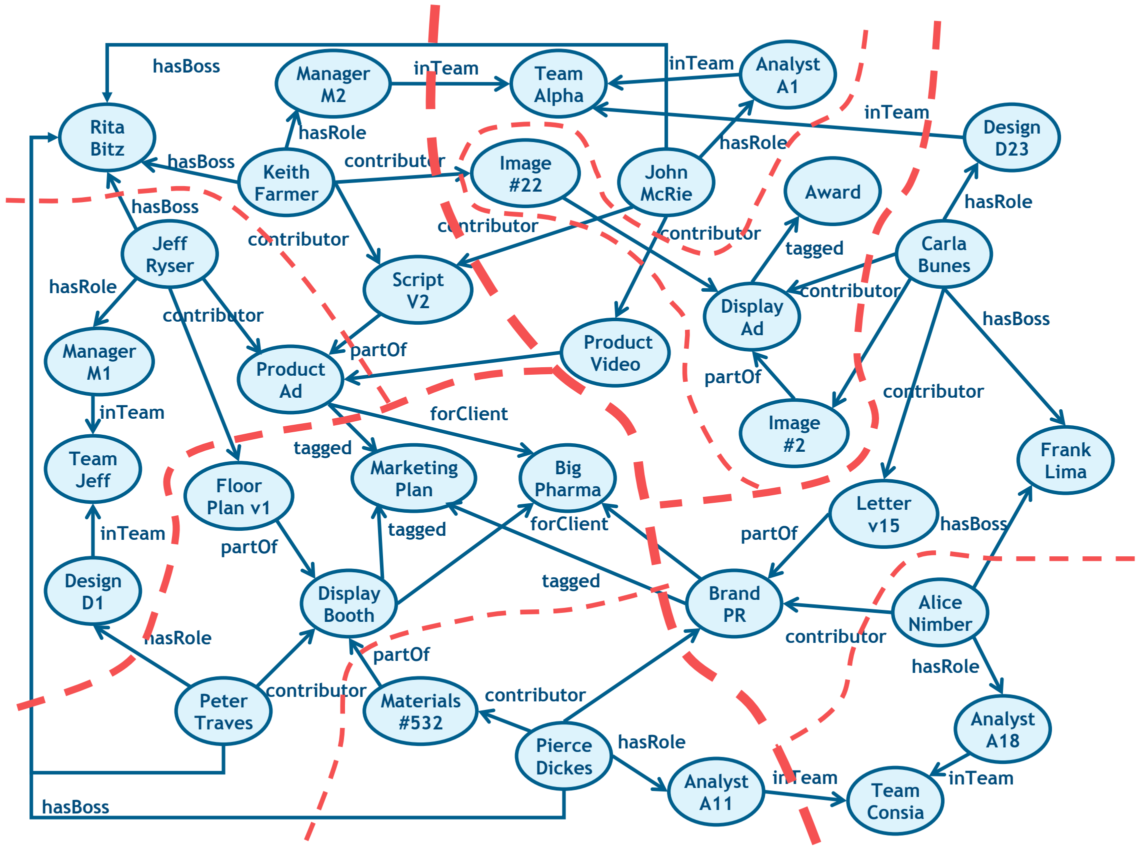


Figure 4.1: Example Partition of the company social network

Suppose $G = (V, E, start, end)$ is a graph, and $G_1 = (V_1, E_1, start_1, end_1)$ and $G_2 = (V_2, E_2, start_2, end_2)$ are two graphs such that $V_1, V_2 \subseteq V$ and k is an integer such that $k \le \max(|V_1|, |V_2|)$. Graph $G_m(V_m, E_m, start_m, end_m)$ is said to be a *k-merge of graphs G_1, G_2 w.r.t. G* iff: $(i)|V_m| = k$; (ii) there is a *surjective* (i.e. onto) mapping $\mu : V_1 \cup V_2 \to V_m$ called the *merge mapping* such that for all $v \in V_m$, $rep(v) = \{v' \in V_1 \cup V_2 \mid \mu(v') = v\}$, and $e_m = (v_1, v_2) \in E_m$ iff there exist $v'_1 \in rep(v_1), v'_2 \in rep(v_2)$ such that $e = (v'_1, v'_2) \in E$. The basic idea tying k-merges to the DOGMA index is that we want DOGMA to be a binary tree each of whose nodes occupies a disk page. Each node is labeled by a graph that "captures" its two children in some way. As each page has a fixed size, the number k limits the size of the graph so that it fits on one page. The idea is that if a node N has two children, N_1 and N_2, then the graph labeling node N should be a k-merge of the graphs labeling its children.

A DOGMA index for a social network database $\mathcal{S}$ is a generalization of the well known binary-tree specialized to represent social network data in the following manner.

Definition 4.1. *A DOGMA index of order k ($k \ge 2$)* is a binary tree $\mathbf{D}_{\mathcal{S}}$ with the following properties:

1. Each node in $\mathbf{D}_{\mathcal{S}}$ equals the size of a disk page and is labeled by a graph.
2. $\mathbf{D}_{\mathcal{S}}$ is balanced.
3. The labels of the set of leaf nodes of $\mathbf{D}_{\mathcal{S}}$ constitute a partition of $\mathcal{S}$.
4. If node N is the parent of nodes N_1, N_2, then the graph G_N labeling node N is a k-merge of the graphs G_{N_1}, G_{N_2} labeling its children.

Note that a single social network database can have many DOGMA indexes.

Example 4.1. Suppose $k = 4$. A DOGMA index for the company social network of Figure 1.9 might split the graph into the 8 components indicated by dashed lines in Figure 4.1 that become the leaf nodes of the index in Figure 4.2. Consider the two left-most leaf nodes. They can be 4-merged together to form a parent node. Other leaf nodes can also be merged together.

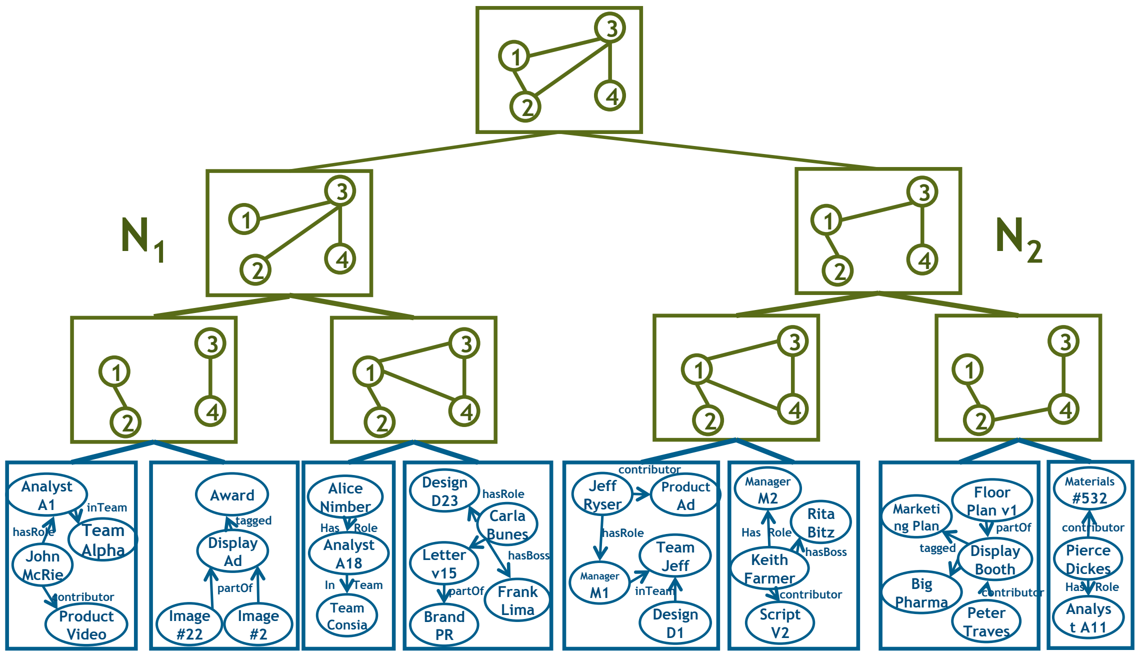


Figure 4.2: A DOGMA index for the social network database of Figure 1.9

Even though many different DOGMA indexes can be constructed for the same social network database, we want to find a DOGMA index with as few "cross" edges between subgraphs stored on different pages as possible. In other words, if node N is the parent of nodes N_1, N_2, then we would like relatively fewer edges in $\mathcal{S}$ between some node in G_{N_1} and some node in G_{N_2}. The smaller this number of edges, the more "self-contained" nodes N_1, N_2 are, and the less likely that a query will require looking at both nodes N_1 and N_2.

In the description of our proposed algorithms, we employ an external graph partitioning algorithm (many of which have been proposed in the literature) that, given a weighted graph, partitions its vertex set in such a way that (*i*) the total weight of all edges crossing the partition is minimized and (*ii*) the accumulated vertex weights are (approximately) equal for both partitions. In our implementation, we employ the *GGGP* graph partitioning algorithm proposed in [76].

```
Algorithm BuildDOGMAIndex
Input: Social network database $\mathcal{S}$, size threshold $k$
     (level $L$, colors $C$)
Output: DOGMA index $\mathbf{D}_{\mathcal{S}}$

1   $G_0 \leftarrow \mathcal{S}$
2   for all $v \in V$
3     $weight(v) \leftarrow 1$
4   for all $e \in E$
5     $weight(e) \leftarrow 1$
6   $i \leftarrow 0$
7   while $|V_i| > k$
8     $i \leftarrow i + 1$
9     $G_i, \mu_i \leftarrow$ CoarsenGraph($G_{i-1}, weight$)
10  $root(\mathbf{D}_{\mathcal{S}}) \leftarrow$ a new "empty" node $R$
11  BuildTree($R, i, G_i$)
12  ColorRegions($L, \mathbf{D}_{\mathcal{S}}, C$) /* Only required for the DOGMA_epd index */
13  return $\mathbf{D}_{\mathcal{S}}$
```

Figure 4.3: BuildDOGMAIndex algorithms

Figure 4.3 provides an algorithm to build a DOGMA index for social network $\mathcal{S}$. The BuildDOGMAIndex algorithm starts with the input social network graph, which is set to G_0. It assigns an arbitrary weight of 1 to each vertex and each edge in G_0. It iteratively coarsens G_0 into a graph G_1 that has about half the vertices in G_0, then coarsens G_1 into a graph G_2 that has about half the vertices as G_1, and so forth until it reaches a G_j that has k vertices or less.

The coarsening is done by invoking a CoarsenGraph algorithm that randomly chooses a vertex v in the input graph, then it finds the immediate neighbors $ngh(v)$ of v, and then finds those nodes in $ngh(v)$ that are best according to a total ordering $\succeq$. There are many ways to define $\succeq$; we experimented with different orderings and chose to order by increasing edge weight, then decreasing vertex weight. The CoarsenGraph algorithm appropriately updates node and edge weights and then selects a maximally weighted node, denoted m, to focus the coarsening on. The coarsening associated with the node v merges neighbors of the node m and m itself into one node, updates weights, and removes v. Edges from m to its neighbors are removed. This process is repeated till we obtain a graph which has half as many vertices (or less) than the graph being coarsened. The result of CoarsenGraph is a k-merge where we have merged adjacent vertices. The BuildDOGMAIndex algorithm then uses the sequence $G_0, G_1, \ldots, G_j$ denoting these coarsened graphs to build the

DOGMA index using the BuildDOGMAIndextree subroutine. Note that Line 12 in the BuildDOGMAIndex algorithm (where L denotes the level at which to color the subgraphs and C is a list of unique colors) is only needed for the DOGMA_epd index introduced later, as well as lines 12–14 in BuildDOGMAIndextree are for the DOGMA_ipd index.

```
Algorithm CoarsenGraph
Input: $G = (V, E, start, end)$, weight function $weight$
Output: Coarsened graph $G'$, merge mapping $\mu$

1  $G' \leftarrow G$
2  $\mu \leftarrow (V \rightarrow V')$ /* identity map */
3  while $2 \times |V'| > |V|$
4     $v \leftarrow$ uniformly random chosen vertex from $V'$
5     $m \leftarrow x \in$ ngh$(v)$ s.t. $x \succeq y\ \forall y \in$ ngh$(v)$
6     $weight(m) \leftarrow weight(m) + weight(v)$
7     for all $e \in E'$ s.t. $start'(e) = v$ and $end'(e) = u$
8        if $\exists f \in E'$ s.t. $start'(f) = m$ and $end'(f) = u$
9           $weight(f) \leftarrow weight(f) + weight(e)$
10       else
11          $E' \leftarrow E' \cup \{f\}$ with $start'(f) = m$ and $start'(f) = u$
12          $weight(f) \leftarrow weight(e)$
13       $E' \leftarrow E' \setminus \{e\}$
14    $V' \leftarrow V' \setminus \{v\}$
15    $\mu(\mu^{-1}(v)) \leftarrow m$
16 return $G', \mu$
```

Figure 4.4: CoarsenGraph algorithms

Proposition 4.1. *The worst-case time complexity of Algorithm BuildDOGMAIndex is $O(|E| \log \frac{|V|}{k} + \Lambda(k) \frac{|V|}{2k} \log \frac{|V|}{k})$ where $\Lambda(k)$ is the worst-case time complexity of Algorithm GraphPartition over a graph with k vertices and $O(k)$ edges.*

Proof. The two time consuming components of the BuildDOGMAIndex algorithm are coarsening the graph and actually building the tree. The time complexity of coarsening a graph G_i with vertices V_i and edges E_i is $O(|V_i| + |E_i|)$ assuming that one iteration over the vertices suffices to reduce the size of the merged graph to one half. This assumption holds when the graph is reasonably connected. Our implementation checks this condition and terminates the coarsening phase early if it is not met (this is not shown in the pseudo code). Copying the graph and iterating over all vertices has time complexity $O(V_i)$. Since we process each only a constant number of times when processing its end points, of which there are only two, the algorithm is also linear in the number of edges.

We invoke CoarsenGraph repeatedly from BuildDOGMAIndextree starting with $G_0 = \mathcal{S}$. However, since each invocation of CoarsenGraph returns a merged graph at

```
Algorithm BuildTree
Input: Binary tree node $N$, level $i$,
    subgraph $S$ at level $i$
Output: Graph merge hierarchy $\{G_j\}_{j\geq 0}$
    and merge mappings $\{\mu_j\}_{j\geq 0}$

1   $label(N) \leftarrow S$
2   if $|S| > k$
3     $S_1, S_2 \leftarrow$ GraphPartition($S$)
4     $L \leftarrow leftChild(N)$
5     $R \leftarrow rightChild(N)$
6     $S_L \leftarrow$ induced subgraph in $G_{i-1}$
7         by vertex set $\{v \mid \mu_i(v) \in V_{S_1}\}$
8     $S_R \leftarrow$ induced subgraph in $G_{i-1}$
9         by vertex set $\{v \mid \mu_i(v) \in V_{S_2}\}$
10    BuildTree($L, i-1, S_L$)
11    BuildTree($r, i-1, S_R$)
12  $P_N \leftarrow \{v \mid \mu_i(\mu_{i-1}(\ldots \mu_1(v))) \in V_S\}$
13  for all $v \in P_N$ /* Only for DOGMA_ipd */
14    $ipd(v, N) \leftarrow \min_{u \in V_0 \setminus P_N} d_{G_0}(u, v)$
```

Figure 4.5: BuildTree algorithms

most half as many entries, we need only on the order of $O(\log \frac{|V|}{k})$ calls to CoarsenGraph until the condition $|V_i| <= k$ holds and the while-loop terminates. We know that the number of vertices in G_{i+1} is half that of G_i, however, we cannot relate that bound to the number of edges. We only know that the number edges is reduced by at least $|V_i|/2$ but the edge set may not be a constant factor smaller. Hence, an upper bound on the runtime complexity of the repeated invocation of CoarsenGraph is given by $O(|E| \log |V|)$.

The time complexity of BuildDOGMAIndextree is dominated by the time it takes to partition the graph S. Each graph S has on the order of $O(k)$ vertices by construction. At the lowest level of the tree, we need to partition the most: there are roughly $\frac{|V|}{2k}$ nodes in the second lowest level of DOGMA index tree, each of which corresponds to some graph S that is partitioned. Assuming sparsity in $\mathcal{S}$, the cost of this total partitioning at the lowest level of the tree is $O(\Lambda(k)\frac{|V|}{2k})$. As argued above, the number of vertices halves with every step up the tree. As before, we cannot bound the number of edges, but the effort should be at most that of the lower level. Hence, the total cost of building the tree is upper bounded by $(\Lambda(k)\frac{|V|}{2k} \log \frac{|V|}{k})$. □

4.2 Algorithms for Processing Graph Queries

In this section, we first present the DOGMA_basic algorithm for answering queries against a DOGMA index stored on external memory. We then present various extensions that improve query answering performance on complex graph queries.

4.2.1 The DOGMA_basic Query Processing Algorithm

Figure 4.6 shows our basic algorithm for answering subgraph matching queries using the DOGMA index. In the description of the algorithm, we assume the existence of two index retrieval functions: retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, v, t$) that retrieves from DOGMA index $\mathbf{D}_{\mathcal{S}}$, the unique identifiers for all vertices v' that are connected to vertex v by an edge labeled with type t, i.e., the neighbors of v restricted to type t, and retrieveVertex($\mathbf{D}_{\mathcal{S}}, v$) that retrieves from $\mathbf{D}_{\mathcal{S}}$ a complete description of vertex v, i.e., its unique identifier and its associated metadata. Note that retrieveVertex implicitly exploits locality, since after looking up neighboring vertices, the probability is high that the page containing the current vertex's description is already in memory.

DOGMA_basic is a recursive, depth-first algorithm which searches the space of all substitutions for the answer set to a given query Q w.r.t an social network database $\mathcal{S}$. For each variable vertex v in Q, the algorithm maintains a set of constant vertices $R_v \subseteq V$ (called result candidates) to prune the search space; for each answer substitution θ for Q, we have $\theta(v) \in R_v$. In other words, the result candidates must be a superset of the set of all matches for v. Hence, we can prune the search space by only considering those substitutions θ for which $\theta(v) \in R_v$ for all variable vertices v in Q.

DOGMA_basic is called initially with an empty substitution and uninitialized result candidates (lines 4-6). We use uninitialized result candidates $R_v = \textbf{null}$ to efficiently denote $R_v = V$, i.e., the fact that there are no constraints on the result candidates yet. The algorithm then initializes the result candidates for all variable vertices v in Q which are connected to a constant vertex c in Q through an edge labeled by type t (lines 7-12). Here we employ the fact that any answer substitution θ must be such that $\theta(v)$ is a neighbor of c, and thus the set of all neighbors of c in $\mathcal{S}$ reachable by an edge of type t are result candidates for v. We use the DOGMA index $\mathbf{D}_{\mathcal{S}}$ to efficiently retrieve the neighborhood of c. If v is connected to multiple constant vertices, we take the intersection of the respective constraints on the result candidates.

At each recursive invocation, the algorithm extends the given substitution and narrows down the result candidates for all remaining variable vertices correspondingly. To extend the given substitution θ, we greedily choose the variable vertex w with the smallest set of result candidates (line 13). This yields a locally optimal branching factor of the search tree since it provides the smallest number of extensions to the current substitution. In fact, if the set of result candidates is empty, then we know that θ cannot be extended to an answer substitution, and we thus directly prune the search (lines 14-15). Otherwise, we consider all the possible result candidates $m \in R_w$ for w by deriving extended substitutions θ' from θ which assign m to w (lines 17-19) and then calling DOGMA_basic recursively on θ' (line 27). Prior to this, we update the result candidates for all remaining variable vertices (lines 20-26). By assigning the constant vertex m to w we can constrain the result

```
Algorithm DOGMA_basic
Input: SM-query $Q$ on social network $\mathcal{S}$, DOGMA index $\mathbf{D}_{\mathcal{S}}$, partial substitution $\theta$,
       candidate sets $\{R_z\}$
Output: Answer substitutions $A$

1  if $\forall z \in V_Q \cap VAR : \exists c : (z \mapsto c) \in \theta$
2    $A \leftarrow A \cup \{\theta\}$
3    return
4  if $\theta = \emptyset$
5    for all $z \in V_Q \cap VAR$
6      $R_z \leftarrow$ null
7    for all $c \in V_Q \cap V$
8      for all edges $e \in E_Q$ with $start_Q(e) = c$ and $end_Q(e) = v \in V_Q \cap VAR$
9        if $R_v =$ null
10         $R_v \leftarrow$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, c, type_Q(e)$)
11       else
12         $R_v \leftarrow R_v \cap$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, c, type_Q(e)$)
13 $R_w \leftarrow \text{argmin}_{R_z \neq \textbf{null}, \text{s.t. } z \in V_Q \cap VAR \setminus dom(\theta)} |R_z|$
14 if $R_w = \emptyset$
15   return "NO"
16 else
17   for all $m \in R_w$
18     retrieveVertex($\mathbf{D}_{\mathcal{S}}, m$)
19     $\theta' \leftarrow \theta \cup \{w \mapsto m\}$
20     for all $z \in V_Q \cap VAR$
21       $R'_z \leftarrow R_z$
22     for all edges $e \in E_Q$ with $start_Q(e) = w$ and $end_Q(e) = v \in V_Q \cap VAR \setminus dom(\theta)$
23       if $R_v =$ null
24         $R'_v \leftarrow$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, m, type_Q(e)$)
25       else
26         $R'_v \leftarrow R_v \cap$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, m, type_Q(e)$)
27     DOGMA_basic($Q, \mathbf{D}_{\mathcal{S}}, \theta', \{R'_z\}$)
```

Figure 4.6: DOGMA_basic algorithm

candidates for all neighboring variable vertices as discussed above.

Note that our description of the algorithm assumes that edges are undirected, to simplify the presentation. Obviously, our implementation takes directionality into account and thus distinguishes between outgoing and incoming edges when determining vertex neighborhoods.

Example 4.2. Consider the example query and company social network database from Chapter 1 in Figure 1.9. Figure 4.7 shows the initial result candidates for each of the variable vertices $?v_1, ?v_2, ?v_3$ in boxes. After initialization, DOGMA_basic chooses the smallest set of result candidates to extend the currently empty substitution $\theta = \emptyset$. We have that $|R_{v_1}| = |R_{v_2}| = 3$; suppose R_{v_2} is chosen. We can now extend θ by assigning each of the result candidates (Product Ad, Display Booth, Brand PR) to $?v_2$. Hence, we first set $\theta'(?v_2) =$ "Product Ad". This introduces a new constant vertex into the query and we thus constrain the result candidates of the two neighbor variable vertices v_1, v_3 by the neighborhood of "Product Ad" restricted to the types "contributor" and "partOf", respectively. The result is shown in Figure 4.8; here we call DOGMA_basic recursively to encounter the empty result candidates for v_1. Hence we reached a dead end in our search for an answer substitution and the algorithm backtracks to try the remaining extensions for θ. Eventually, DOGMA_basic considers the extension $v_2 \to$ "Brand PR" which leads to the query answer. □

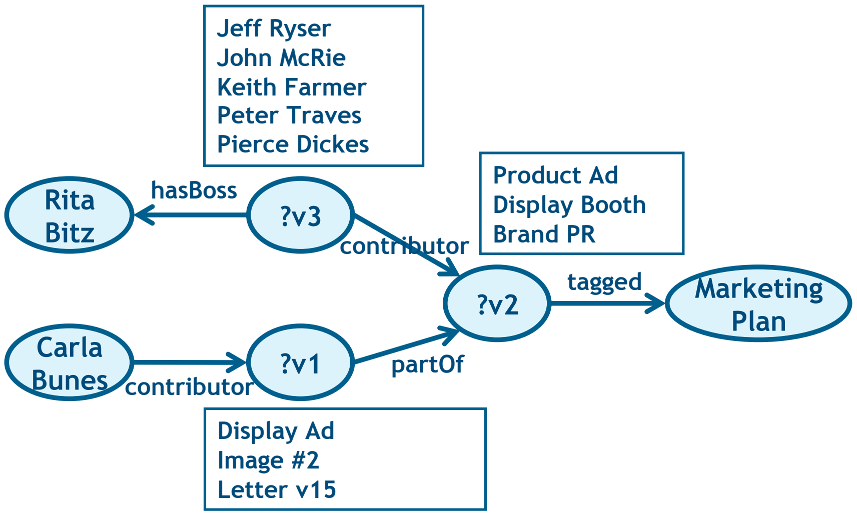


Figure 4.7: Execution of DOGMA_basic for query 1.10

Proposition 4.2. *Suppose $\boldsymbol{D}_{\mathcal{S}}$ is a DOGMA index for an social network database $\mathcal{S}$ and Q is a SM-query with at least one constant vertex.*
Then, DOGMA_basic$(Q, \boldsymbol{D}_{\mathcal{S}}, \{\}, null)$ returns the set of all correct answer substitutions for query Q w.r.t. $\mathcal{S}$. Moreover, the worst-case complexity of the DOGMA_basic algorithm is $O(|V|^{|V_Q \cap VAR|})$.

Proof. To prove the correctness of DOGMA_basic we show that the set of answer substitutions returned is (1) a superset of the true set of answer substitution A and

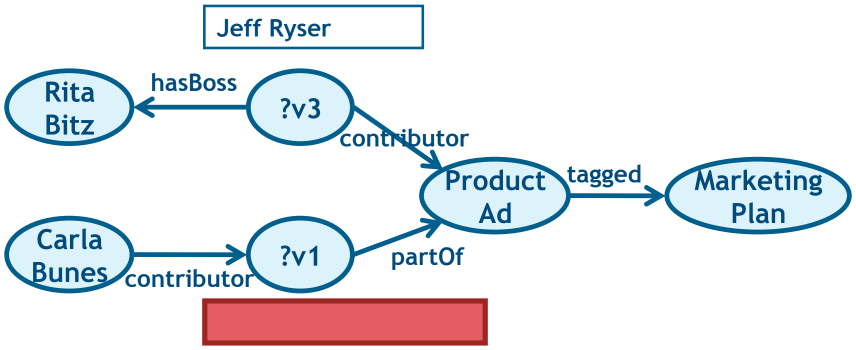


Figure 4.8: Execution of $\mathsf{DOGMA\_basic}$ for query 1.10

(2) a subset of A, which establishes that it must be equal to A. We assume that the query Q is fixed, as is the social network database $\mathcal{S}$ and its index $\mathbf{D}_{\mathcal{S}}$. **Superset of A:** To show that $\mathsf{DOGMA\_basic}$ returns a superset of A, we establish the following invariant when $\mathsf{DOGMA\_basic}$ is initially called with $\mathsf{DOGMA\_basic}(Q, \mathbf{D}_{\mathcal{S}}, \{\}, null)$: Throughout the execution of $\mathsf{DOGMA\_basic}$ it holds that for any answer substitution $\tilde{\theta} \in A$, s.t. for all $z \in \mathsf{domain}(\theta)$ $\tilde{\theta}(z) = \theta(z)$ we have for all $z \in (V_Q \cap VAR) \setminus$ $\mathsf{domain}theta$ $\tilde{\theta}(z) \in R_z$. In other words, for any true answer substitution $\tilde{\theta} \in A$ which extends the current partial substitution θ we have that the assignments by $\tilde{\theta}$ to the currently unmatched variable vertices $zz \in (V_Q \cap VAR) \setminus$ must be contained in the set of potential matches of z, R_z. Note, that we use $R_z = \mathbf{null}$ to represent $R_z = V$ space efficiently and therefore the invariant trivially holds after the initialization in line 5 and 6. After that, we use neighborhoods of constant vertices and newly matches variable vertices to constrain the sets of possible matches. Since any answer substitution specifies, by definition, an isomorphic subgraph in $\mathcal{S}$ is must also satisfy the neighborhood requirement and, therefore, the invariant must still hold after the set of possible matches is constrained.

Now, since the invariant holds and we substitute all of the possible matches in line 17 of the algorithm, $\mathsf{DOGMA\_basic}$ must necessarily consider all answer substitutions on lines 1 and 2 of the algorithm.

Subset of A: To establish that $\mathsf{DOGMA\_basic}$ returns only true answer substitutions, we establish the following invariant: Throughout the execution of $\mathsf{DOGMA\_basic}$ the $Q\theta$ is a subgraph of $\mathcal{S}$, where θ is the current partial substitution and $Q\theta$ denotes the partially substituted query Q where all remaining variables are removed. Since θ is initially empty, this invariant is trivially true. During initialization on the constant vertices c and as we extend the substitution by new matches for variable vertices w, the sets of possible matches R_v of adjacent variable vertices v to c or w are constrained by the neighborhoods of c or v, respectively. Hence, any subsequent extension of the substitution θ must satisfy the edge conditions to already matched or constant vertices and therefore the invariant continues to hold. Consequently, when the condition on line 1 holds, it must be that θ is an answer substitution.

The fact that DOGMA_basic has a worst case time complexity exponential in the number of variable vertices $V_Q \cap VAR$ is a standard result for subgraph isomorphism algorithms [151]. Consider a social network database $\mathcal{S}$ with vertices V that is fully connected and has just one type label. Then the set of answer substitutions for any (reasonable) subgraph matching query Q with m variable vertices is equal to V^m, i.e., of size exponential in m. This is the worst case time complexity, because in processing each variable vertex v in Q you can substitute at most all of the vertices in V. □

The algorithm is therefore exponential in the number of variables in the query in the worst case. However, the algorithm is efficient in practice as we will show in Section 4.3. Furthermore, we propose two extensions of the DOGMA index that improve its performance.

4.2.2 The DOGMA_adv Algorithm

The basic query answering algorithm presented in the previous section only uses "short range" dependencies, i.e., the immediate vertex neighborhood of variable vertices, to constrain their result candidates. While this suffices for most simple queries, considering "long range" dependencies can yield additional constraints on the result candidates and thus improve query performance. For instance, the result candidates for v_1 in our example query not only must be immediate neighbors of "Carla Bunes": in addition, they must be at most at a distance of 2 from "Marketing Plan". More formally, let $d_{\mathcal{S}}(u, v)$ denote the length of the shortest path between two vertices $u, v \in V$ in the undirected counterpart of a social network $\mathcal{S}$, and let $d_Q(u, v)$ denote the distance between two vertices in the undirected counterpart of a query Q; a long range dependency on a variable vertex $v \in V_Q$ is introduced by any constant vertex $c \in V_Q$ with $d_Q(v, c) > 1$.

We can exploit long range dependencies to further constrain result candidates. Let v be a variable vertex in Q and c a constant vertex with a long range dependency on v. Then any answer substitution θ must satisfy $d_Q(v, c) \geq d_{\mathcal{S}}(\theta(v), c)$ which, in turn, means that $\{m \mid d_{\mathcal{S}}(m, c) \leq d_Q(v, c)\}$ are result candidates for v.

This is the core idea of the DOGMA_adv algorithm shown in Fig. 4.9, which improves over and extends DOGMA_basic. In addition to the result candidates sets R_v, the algorithm maintains sets of distance constraints C_v on them. As long as a result candidates set R_v remains uninitialized, we collect all distance constraints that arise from long range dependencies on the variable vertex v in the constraints set C_v (lines 15-16 and 34-35). After the result candidates are initialized, we ensure that all elements in R_v satisfy the distance constraints in C_v (lines 17-18 and 37-38). Maintaining additional constraints therefore reduces the size of R_v and hence the number of extensions to θ we have to consider (line 23 onward).

DOGMA_adv assumes the existence of a *distance index* to efficiently look up $d_{\mathcal{S}}(u, v)$ for any pair of vertices $u, v \in V_{\mathcal{S}}$ (through function retrieveDistance), since computing graph distances at query time is clearly inefficient. But how can we build such an index? Computing all-pairs-shortest-path has a worst-case time complexity

```
Algorithm DOGMA_adv
Input: SM-query $Q$ on social network $\mathcal{S}$, DOGMA index $\mathbf{D}_{\mathcal{S}}$, partial substitution $\theta$,
       candidate sets $\{R_z\}$, constraint sets $\{C_z\}$
Output: Answer substitutions $A$

1  if $\forall z \in V_Q \cap \mathit{VAR} :\ \exists c : (z \mapsto c) \in \theta$
2    $A \leftarrow A \cup \{\theta\}$
3    return
4  if $\theta = \emptyset$
5    for all $z \in V_Q \cap \mathit{VAR}$
6      $R_z \leftarrow$ null
7    for all $c \in V_Q \cap V$
8      for all edges $e \in E_Q$ with $start_Q(e) = c$ and $end_Q(e) = v \in V_Q \cap \mathit{VAR}$
9        if $R_v =$ null
10         $R_v \leftarrow$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, c, type_Q(e)$)
11       else
12         $R_v \leftarrow R_v \cap$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, c, type_Q(e)$)
13   for all $c \in V_Q \cap V$
14     for all variable vertices $v \in V_Q \cap \mathit{VAR}$ s.t. $d_Q(c, v) > 1$
15       if $R_v =$ null
16         $C_v \leftarrow C_v \cup \{(c, d_Q(c, v))\}$
17       else
18         $R_v \leftarrow \{u \in R_v \mid$ retrieveDistance($\mathbf{D}_{\mathcal{S}}, c, u$) $\leq d_Q(c, v)\}$
19 $R_w \leftarrow \text{argmin}_{R_z \neq \textbf{null}, \text{s.t. } z \in V_Q \cap \mathit{VAR} \setminus dom(\theta)} |R_z|$
20 if $R_w = \emptyset$
21   return "NO"
22 else
23   for all $m \in R_w$
24     retrieveVertex($\mathbf{D}_{\mathcal{S}}, m$)
25     $\theta' \leftarrow \theta \cup \{w \mapsto m\}$
26     for all $z \in V_Q \cap \mathit{VAR}$
27       $R'_z \leftarrow R_z$
28       $C'_z \leftarrow C_z$
29     for all edges $e \in E_Q$ with $start_Q(e) = w$ and $end_Q(e) = v \in V_Q \cap \mathit{VAR} \setminus dom(\theta)$
30       if $R_v =$ null
31         $R'_v \leftarrow \{u \in$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, m, type_Q(e)$) $\mid$
                   $\forall (c, d) \in C_v :$ retrieveDistance($\mathbf{D}_{\mathcal{S}}, c, u$) $\leq d\}$
32       else
33         $R'_v \leftarrow R_v \cap$ retrieveNeighbors($\mathbf{D}_{\mathcal{S}}, m, type_Q(e)$)
34     for all variable vertices $v \in V_Q \cap \mathit{VAR} \setminus dom(\theta)$ s.t. $d_Q(w, v) > 1$
35       if $R_v =$ null
36         $C_v \leftarrow C_v \cup \{(m, d_Q(w, z))\}$
37       else
38         $R_v \leftarrow \{w \in R_v \mid$ retrieveDistance($\mathbf{D}_{\mathcal{S}}, m, v$) $\leq d_Q(w, v)\}$
39     DOGMA_adv($Q, \mathbf{D}_{\mathcal{S}}, \theta', \{R'_z\}, \{C'_z\}$)
```

Figure 4.9: DOGMA_adv algorithm

$O(|V|^3)$ and space complexity $O(|V|^2)$, both of which are clearly infeasible for large social network databases. However, we do not need to know the *exact* distance between two vertices for DOGMA_adv to be correct. Since all the distance constraints in DOGMA_adv are *upper bounds* (lines 18, 31, and 38), all we need is to ensure that $\forall u, v \in V$, $\mathsf{retrieveDistance}(\mathbf{D}_{\mathcal{S}}, u, v) \leq d_{\mathcal{S}}(u, v)$.

Thus, we can extend the DOGMA index to include distance information and build two "lower bound" distance indexes, DOGMA_ipd and DOGMA_epd, that use approximation techniques to achieve acceptable time and space complexity.

4.2.3 DOGMA_ipd

To build the DOGMA index, we used a graph partitioner which minimizes cross edges, to ensure that strongly connected vertices are stored in close proximity on disk; this implies that distant vertices are likely to be assigned to distinct sets in the partition. We exploit this to extend DOGMA to a distance index.

As seen before, the leaf nodes of the DOGMA index $\mathbf{D}_{\mathcal{S}}$ are labeled by subgraphs which constitute a partition of $\mathcal{S}$. For any node $N \in \mathbf{D}_{\mathcal{S}}$, let P_N denote the union of the graphs labeling all leaf nodes reachable from N. Hence, P_N is the union of all subgraphs in $\mathcal{S}$ that were eventually merged into the graph labeling N during index construction and therefore corresponds to a larger subset of $\mathcal{S}$. For example, the dashed lines in Figure 4.1 mark the subgraphs P_N for all index tree nodes N of the DOGMA index shown in Figure 4.2 where bolder lines indicate boundaries corresponding to nodes of lower depth in the tree.

The DOGMA *internal partition distance* (DOGMA_ipd) index stores, for each index node N and vertex $v \in P_N$, the distance to the outside of the subgraph corresponding to P_N. We call this the *internal partition distance* of v, N, denoted $ipd(v, N)$, which is thus defined as $ipd(v, N) = \min_{u \in V \setminus P_N} d_{\mathcal{S}}(v, u)$. We compute these distances during index construction as shown in 4.5 (BuildDOGMAIndextree algorithm at lines 12-14). At query time, for any two vertices $v, u \in V$ we first use the DOGMA tree index to identify those distinct nodes $N \neq M$ in $\mathbf{D}_{\mathcal{S}}$ such that $v \in P_N$ and $u \in P_M$, which are at the same level of the tree and closest to the root. If such nodes do not exist (because v, u are associated with the same leaf node in $\mathbf{D}_{\mathcal{S}}$), then we set $d_{ipd}(u, v) = 0$. Otherwise we set $d_{ipd}(u, v) = \max(ipd(v, N), ipd(u, M))$. It is easy to see that d_{ipd} is an admissible lower bound distance, since $P_N \cap P_M = \emptyset$. By choosing those distinct nodes which are closest to the root, we ensure that the considered subgraphs are as large as possible and hence $d_{ipd}(u, v)$ is the closest approximation to the actual distance.

Proposition 4.3. *Building the DOGMA_ipd index for a social network database $\mathcal{S}$ has time complexity $O(\log \frac{|V|}{k}(|E| + |V| \log |V|))$ and storage complexity $O(|V| \log \frac{|V|}{k})$.*

Proof. Suppose we fix some level L of the $\mathbf{D}_{\mathcal{S}}$ tree which contains the index nodes $\{N_1, \ldots, N_{n_L}\}$. Associated with those index nodes is the partition of the social network $\{P_{N_1}, \ldots, P_{N_{n_L}}\}$. For each block in this partition, we compute the distance to the boundary of that block for all vertices in the block. We use the single-source version of Dijkstra's shortest path algorithm with Fibonacci heaps to find these distances by introducing a virtual vertex that is connect to all vertices at the boundary of that partition block by an edge of length one as the single-source. Since we compute this single-source distance restricted to each partition block and since the partition encompasses all of $\mathcal{S}$, the total time complexity is $O(|E| + |V| \log |V|)$, i.e., the same as computing one single-source distance for all of $\mathcal{S}$. The internal partition distance is computed for each level of the $\mathbf{D}_{\mathcal{S}}$ tree, of which there are $O(\log \frac{|V|}{k})$. Hence, the overall time complexity is $O(\log \frac{|V|}{k}(|E| + |V| \log |V|))$.

For each vertex, we store the distance to each internal partition defined by each level in the $\mathbf{D}_{\mathcal{S}}$ tree. There are $O(\log \frac{|V|}{k})$ levels, the overall storage complexity associated with storing all internal partition distances is $O(|V| \log \frac{|V|}{k})$. □

Example 4.3. Consider the company example of Figure 1.9. As shown in Figure 4.10, there is a long range dependency between "Carla Bunes" and variable vertex v_2 at distance 2. The boldest dashed line in Figure 1.9 marks the top level partition and separates the sets P_{N_1}, P_{N_2}, where N_1, N_2 are the two nodes directly below the root in the DOGMA index in Figure 4.2. We can determine that $ipd(\text{Carla Bunes}, N_2) = 3$ and since "Product Ad" and "Display Booth" lie in the other subgraph, it follows that:

$$d_{ipd}(\text{Carla Bunes}, \text{Product Ad}) = d_{ipd}(\text{Carla Bunes}, \text{Display Booth}) = 3$$

and therefore we can prune both result candidates. Note that this reduces the query time to a third.

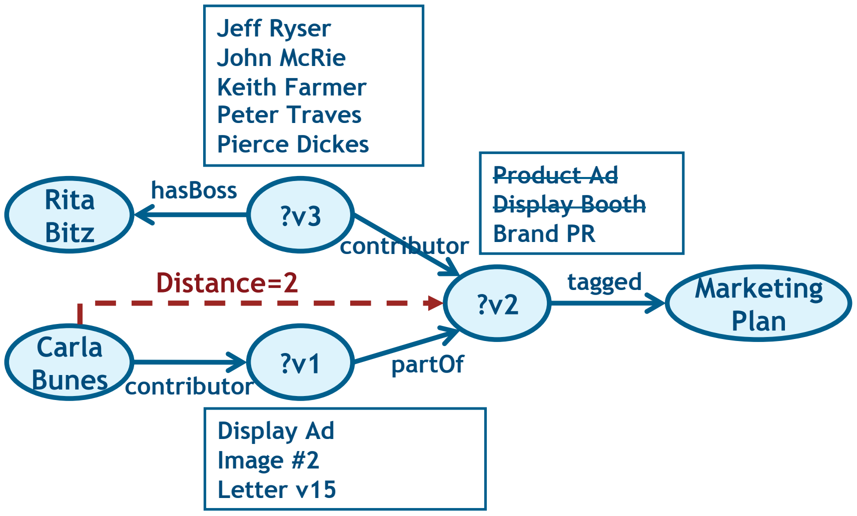


Figure 4.10: Using DOGMA_ipd for query answering

4.2.4 DOGMA_epd

The DOGMA *external partition distance* (DOGMA_epd) index also uses the partitions in the index tree to compute a lower bound distance. However, it considers the distance to *other* subgraphs rather than the distance within the *same* one. For some fixed level L, let $\mathcal{N}_L$ denote the set of all nodes in $\mathbf{D}_\mathcal{S}$ at distance L from the root. As discussed above, $P = \{P_N\}_{N \in \mathcal{N}_L}$ is a partition of $\mathcal{S}$. The idea behind DOGMA_epd is to assign a color from a fixed list of colors C to each subgraph $P_N \in P$ and to store, for each vertex $v \in V$ and color $c \in C$, the shortest distance from v to a subgraph colored by c. We call this the *external partition distance*, denoted $epd(v, c)$, which is thus defined as $epd(v, c) = \min_{u \in P_N, \phi(P_N)=c} d_\mathcal{S}(v, u)$ where $\phi : P \to C$ is the color assignment function. We store the color of P_N with its index node N so that for a given pair of vertices u, v we can quickly retrieve the colors c_u, c_v of the subgraphs to which u and v belong. We then compute $d_{epd}(v, u) = \max(epd(v, c_u), epd(u, c_v))$. It is easy to see that d_{epd} is an admissible lower bound distance.

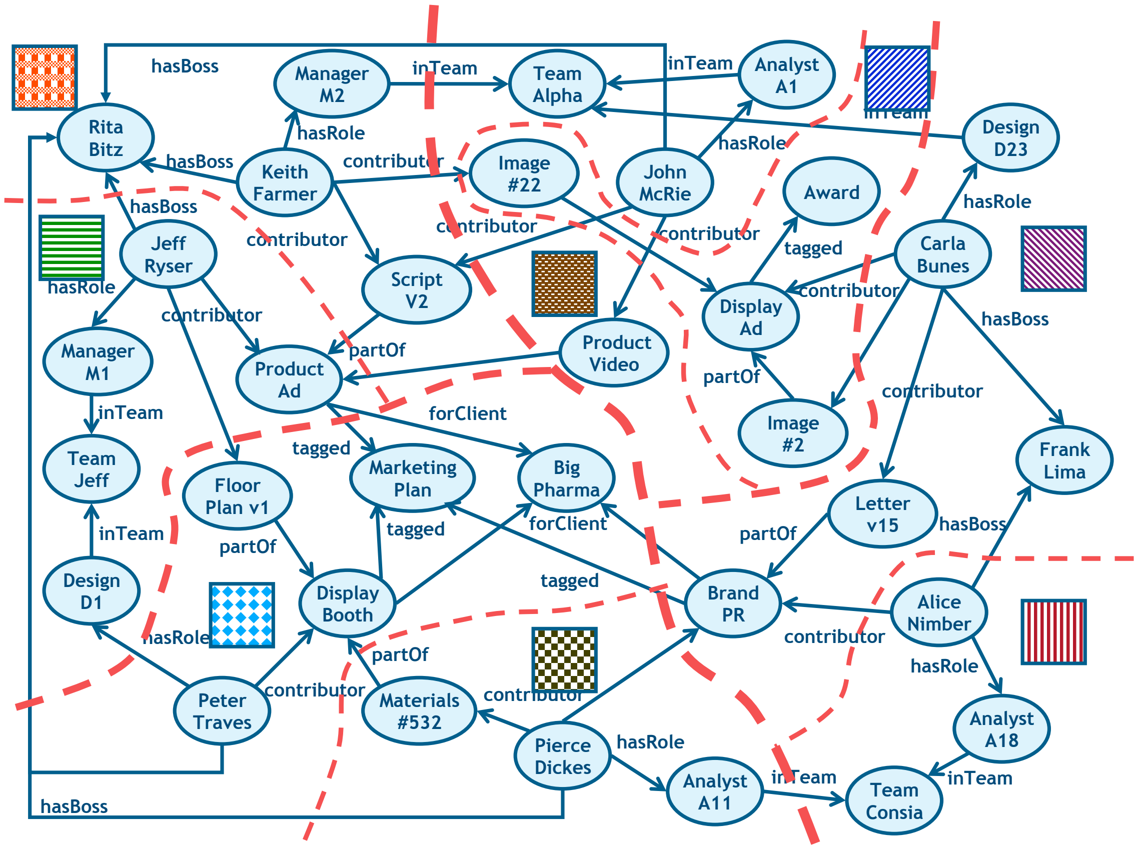


Figure 4.11: Example coloring of graph partitions

Ideally, we want to assign each partition a distinct color but this exceeds our storage capabilities for large database sizes. Our problem is thus to assign a limited number of colors to the subgraphs in such a way as to maximize the distance between subgraphs of the same color. Formally, we want to minimize the objective function $\sum_{P_N \in P} \sum_{P_M \in P, \phi(P_N)=\phi(P_M)} \frac{1}{d(P_N, P_M)}$ where $d(P_N, P_M) = \min_{u \in P_N, v \in P_M} d_\mathcal{S}(u, v)$. We leverage the work of Ko and Rubenstein on peer-to-peer networks [85],to design a probabilistic, locally greedy optimization algorithm for the maximum distance coloring problem named ColorRegions, shown in Figure 4.12. The algorithm starts with a random color assignment and then iteratively updates the colors of individual partitions to be locally optimal. A *propagation radius* determines the neighborhood that is analyzed in determining the locally optimal color. The algorithm terminates if the cost improvement falls below a certain threshold or if a maximum number of iterations is exceeded.

```
Algorithm ColorRegions
Input: level $L$, DOGMA index $\mathbf{D}_{\mathcal{S}}$, social network $\mathcal{S}$, merge function $\{\mu_i\}$,
    list of colors $C$
Global variables: propagation radius $\gamma$, maximum number of iterations $K$

1   $P \leftarrow \{P_N \mid N \in \mathbf{D}_{\mathcal{S}} \wedge depth_{\mathbf{D}_{\mathcal{S}}}(N) = L \wedge$
    $u \in P_N \Leftrightarrow \mu_L(\mu_{L-1}(\ldots \mu_1(u))) \in label(N)\}$
2   for all $P_N \in P$
3       $\phi(P_N) \leftarrow$ color chosen uniformly at random from $C$
4   $n \leftarrow 0$
5   while $n < K$ or little improvement
6       for all $P_N \in P$
7           $\phi(P_N) \leftarrow \text{argmin}_{c \in C} \sum_{\left[\begin{smallmatrix} P_M \in P, \text{ s.t. } \phi(P_M)=c \\ \wedge\, d(P_N, P_M) < \gamma \end{smallmatrix}\right]} \frac{1}{d(P_N, P_M)}$
8   for all $v \in V$
9       for all $c \in C$
10          $epd(v, c) \leftarrow \min_{u \in P_N, \phi(P_N)=c} d_{\mathcal{S}}(v, u)$
```

Figure 4.12: ColorRegions algorithm

Proposition 4.4. *Computing the external partition distance has a worst-case time complexity $O(|C|\,(|E| + |V| \log |V|))$ and storage complexity $O(|V|\,|C|)$.*

Proof. We assume that the user specified level L at which the color assignment is found is high in the $\mathbf{D}_{\mathcal{S}}$ tree and therefore the number of nodes in the tree at level L is orders of magnitude smaller than the number of vertices in $\mathcal{S}$. Under that assumption, the time taken to find a heuristically optimal color assignment does not impact the runtime complexity because it is dominated by the time needed to compute the actual external partition distance.

To find the actual external distances for each vertex $v \in V$ and each color in $c \in C$, we first identify all the set of all nodes $\mathcal{N}_c$ in the $\mathbf{D}_{\mathcal{S}}$ tree that been assigned the color c. Using the index structure, we identify all nodes $V_c \subset V$ in $\mathcal{S}$ that are subsumed by these nodes, i.e., $V_c = \bigcup_{N \in \mathcal{N}_c} P_N$. Next, we introduce a virtual node that is connected to all nodes in V_c by an edge of length 0 and compute the single-source shortest path distances for all vertices in $V \setminus V_c$ using Dijkstra's shortest path algorithm with Fibonacci heaps which has time complexity $O(|E| + |V| \log |V|)$. Since we repeat this process for all colors $c \in C$, the total time complexity is $O(|C|\,(|E| + |V| \log |V|))$.

For each vertex we store the shortest path distance for each color, which requires storage in the order of $O(|V|\,|C|)$. $\square$

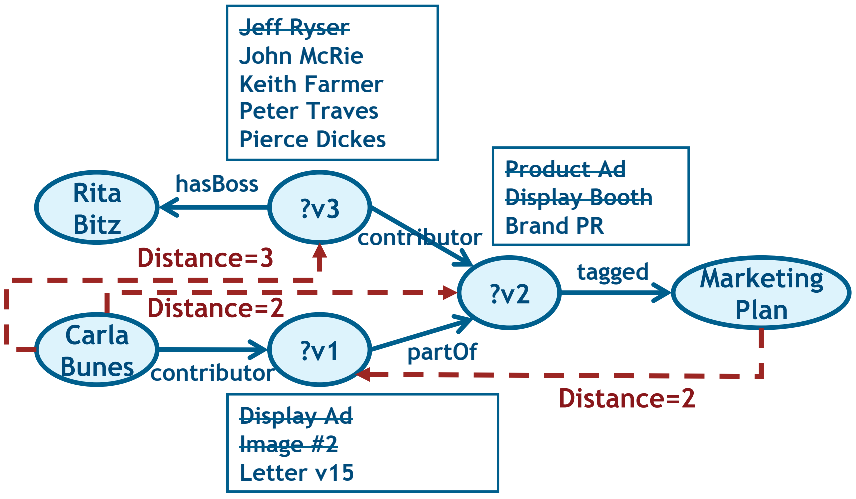


Figure 4.13: Using DOGMA_epd for query answering

Example 4.4. Consider the example of Figure 1.9 and assume each set in the lowest level of the DOGMA index in Figure 4.2 is colored with a different color as shown in Figure 4.11. Figure 4.11 shows a coloring of the partitions associated with level 3 of the DOGMA index displayed in figure 4.2 with $C = \{C_1, \ldots, C_8\}$. Figure 4.13 indicates some long range dependencies and shows how the external partition distance can lead to additional prunings in the three result candidates sets which can be verified against Figure 4.11.

4.3 Experimental Evaluation

In this section we present the results of the experimental assessment we performed of the DOGMA_adv algorithm combined with DOGMA_ipd and DOGMA_epd indexes.

We compared the performance of our algorithm and indexes with 4 leading graph database systems developed in the Semantic Web community that are most widely used and have demonstrated superior performance in previous evaluations [89]. All of these systems are primarily RDF triple stores, but as we have shown in Chapter 2, RDF is just a special subclass of social networks restricted to type labels. For compatibility, we converted SM-queries to SPARQL, the standardized query language for RDF [136]. *Sesame2* [137] is an open source RDF framework for storage, inference and querying of RDF data, that includes its own graph indexing and I/O model and also supports a relational database as its storage backend. We compare against Sesame2 using its native storage model since initial experiments have shown that Sesame2's performance drops substantially when backed by a relational database system. *Jena2* [157] is a popular Java RDF framework that supports persistent RDF storage backed by a relational database system (we used PostgreSQL [122]). SPARQL queries are processed by the ARQ query engine which also supports query optimization [142]. *JenaTDB* [73] is a component of the Jena framework providing persistent storage and query optimization for large scale graph datasets based on a native indexing and I/O model. Finally, *OWLIM* [81] is a high performance semantic repository based on the Sesame database. In the experiments, we compared against the internal memory version of OWLIM which is called *SwiftOWLIM* and is freely available. SwiftOWLIM loads the entire dataset into main memory prior to query answering and therefore must be considered to have an advantage over the other systems.

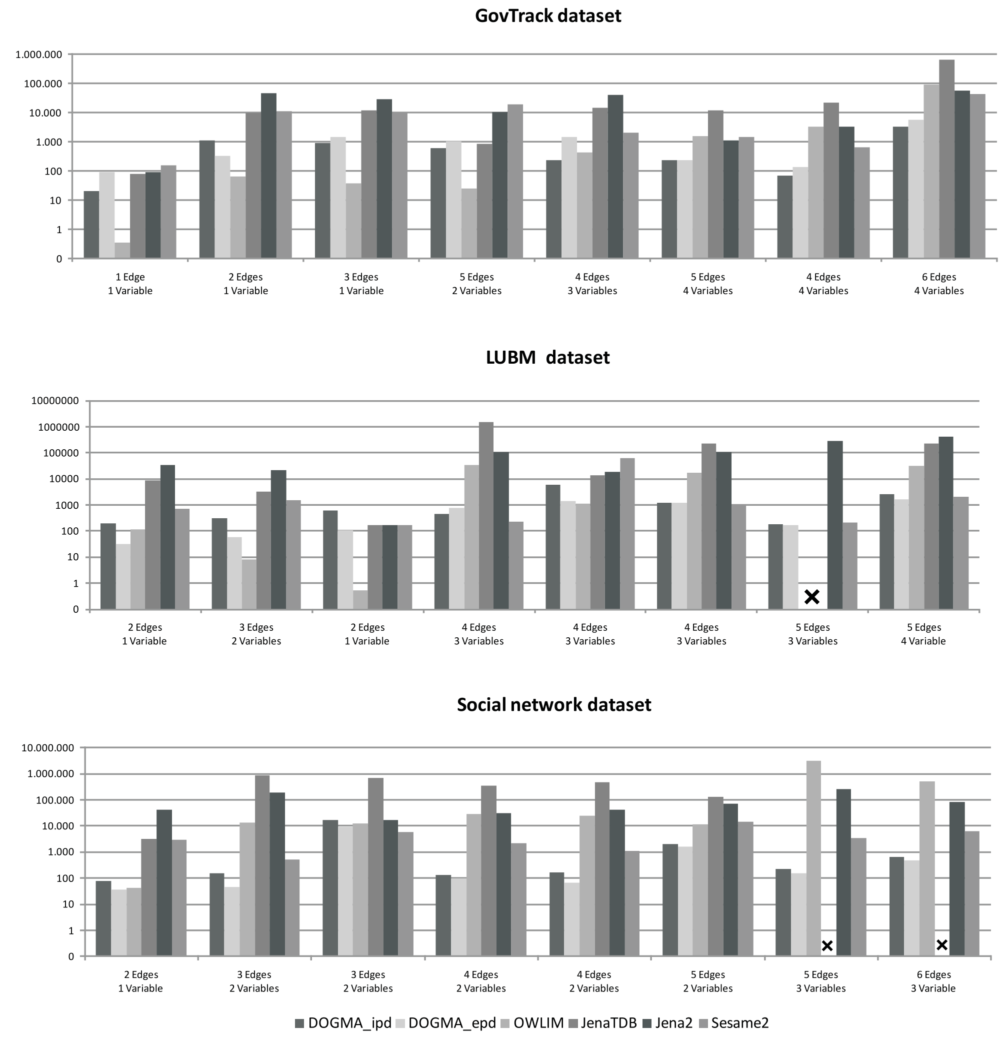


Figure 4.14: Query times (ms) for graph queries of low complexity

Moreover, we used 3 different graph datasets. *GovTrack* [62] consists of more than 14.5 million edges describing data about the U.S. Congress. The *Lehigh University Benchmark* (LUBM) [148] is frequently used within the Semantic Web community as the basis for evaluation of RDF and ontology storage systems. The benchmark's graph data generator employs a schema which describes the university domain. We generated a dataset of more than 13.5 million edges. Finally, a fragment of the Flickr online social network [50] dataset was collected by researchers of the MPI Saarbrücken to analyze online social networks [102] and was generously made available to us. The dataset contains information on the relationships between individuals and their memberships in groups. The fragment we used for the experiments was anonymized and contains approximately 16 million edges. The GovTrack and social network datasets are well connected (with the latter being denser than the former), whereas the dataset generated by the LUBM benchmark is a sparse and almost degenerate graph dataset containing a set of small and loosely connected subgraphs.

In order to allow for a meaningful comparison of query times across the different systems, we designed a set of graph queries with varying complexity, where constant vertices were chosen randomly and queries with an empty result set were filtered out. Queries were grouped into classes based on the number of edges and variable vertices. We repeated the query time measurements multiple times for each query, eliminated outliers, and averaged the results. Finally, we averaged the query times of all queries in each class. All experiments were executed on a machine with a 2.4Ghz Intel Core 2 processor and 3GB of RAM.

The most important performance metric in our experimental evaluation is query time, i.e. the time needed to answer an issued graph query. In order to allow for a meaningful comparison of query times across the different systems we designed a set of graph queries with varying complexity in accordance with the respective dataset. To reduce the bias in our design we selected constant vertices contained in the graph queries at random. Finally, queries which returned an empty result set were filtered out. To generalize the results and ease presentation, multiple queries are grouped into classes of different complexity based on the number of edges and variable vertices. Each edge of a graph query constitutes a constraint on its result set and each variable vertex decreases the selectivity of a query, hence both numbers can serve as an indicator for a query's complexity. Yet, a graph query's complexity also depends on the structure of the graph and its constant vertices, hence graph queries with fewer edges and less variables might actually have a higher complexity.

In a first round of experiments, we designed several relatively simple graph queries for each dataset, containing no more than 6 edges, and grouped them into 8 classes. The results of these experiments are shown in Figure 4.14 which reports the query times for each query class on each of the three datasets. Missing values in the figure indicate that the system did not terminate on the query within a reasonable amount of time (around 20 mins). Note that the query times are plotted in logarithmic scale to accommodate the large discrepancies between systems.

The results show that OWLIM has low query times on low complexity queries across all datasets. This result is not surprising, as OWLIM loads all data into main memory prior to query execution. The performance advantage of DOGMA_ipd and DOGMA_epd over the other systems increases with query complexity on the GovTrack and social network dataset, where our proposed techniques are orders of magnitude faster on the most complex queries. On the LUBM dataset, how-

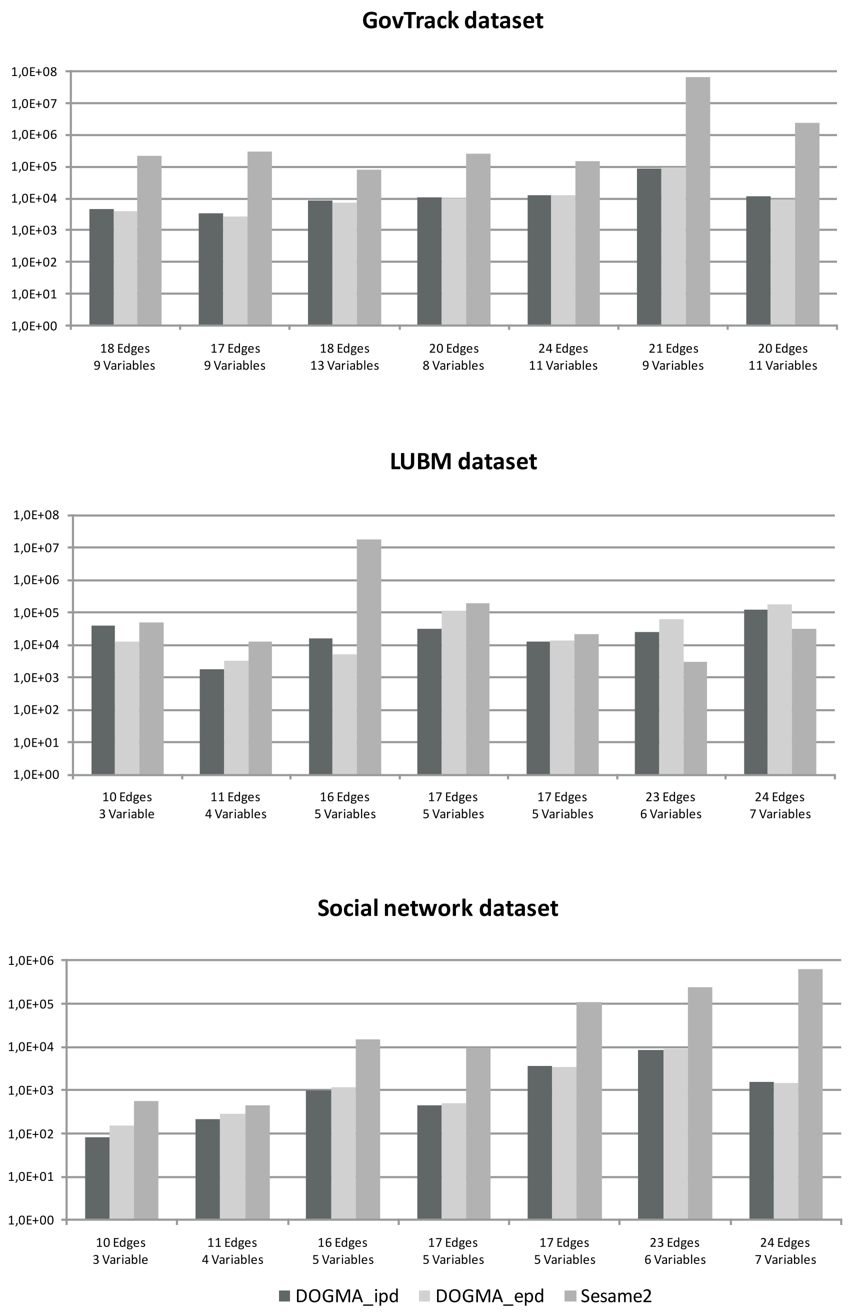


Figure 4.15: Query times (ms) for graph queries of high complexity

ever, Sesame2 performs almost equally for the more complex queries. Finally, DOGMA_epd is slightly faster on the LUBM and social network dataset, whereas DOGMA_ipd has better performance on the Govtrack dataset.

In a second round of experiments, we significantly increased the complexity of the queries, which now contained up to 24 edges. Unfortunately, the OWLIM, JenaTDB, and Jena2 systems did not manage to complete the evaluation of these queries in reasonable time, so we exclusively compared with Sesame2. The results are shown in Figure 4.15. On the GovTrack and social network dataset, DOGMA_ipd and DOGMA_epd continue to have a substantial performance advantage over Sesame2 on all complex graph queries of up to 40000%. For the LUBM benchmark, the picture is less clear due to the particular structure of the generated dataset explained before.

Finally, Figure 4.16 compares the storage requirements of the systems under comparison for all three datasets. The results show that DOGMA_ipd,DOGMA_epd and Sesame2 are the most memory efficient.

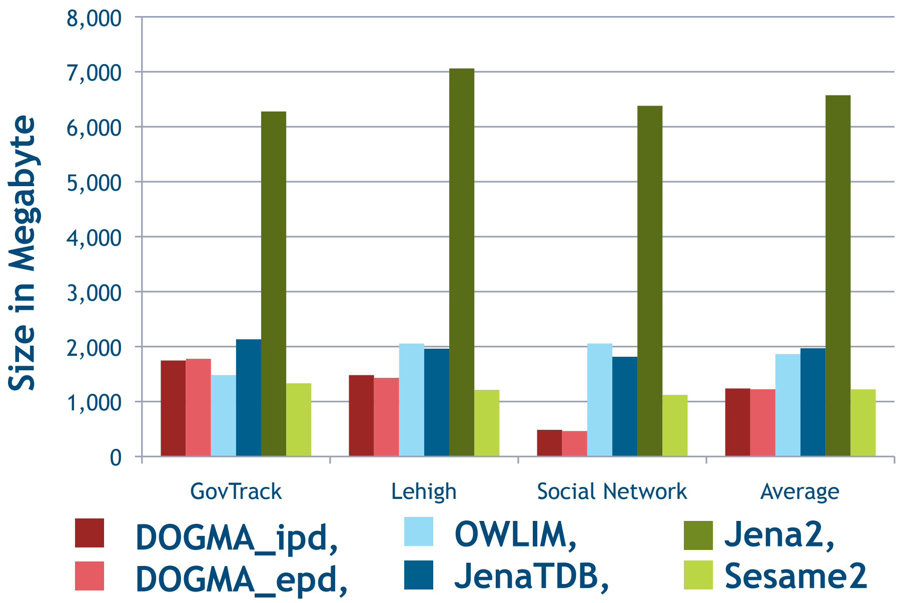


Figure 4.16: Index size (MB) for different datasets

To wrap up the results of our experimental evaluation, we can observe that both DOGMA_ipd and DOGMA_epd are significantly faster than all other graph database systems under comparison on complex graph queries over non-degenerate graph datasets. Moreover, they can efficiently answer complex queries on which most of the other systems do not terminate or take up to 400 times longer, while maintaining a satisfactory storage footprint. DOGMA_ipd and DOGMA_epd have similar performance, yet differences exist which suggest that each index has unique advantages for particular queries and RDF datasets. Investigating these is subject of future research.

Chapter 5
Budget Match Algorithm for Subgraph Queries

To efficiently answer subgraph matching queries on very large real world social network databases, we must devise subgraph matching algorithms which aggressively prune the search space. Existing work on subgraph matching algorithms and index structures typically employ heuristics to predict the cost of answering strategies based on statistics about the database and the current state of query processing and then choose a strategy to minimize cost. However, *static* cost models can be very inaccurate when facing long-tailed degree distributions, where they may suggest suboptimal query plans. These long-tailed degree distributions, such as power-law distributions [6], are very common for real world networks and have been confirmed on numerous databases (e.g. [47, 112]).

In this chapter, we propose the BudgetMatch algorithm to efficiently answer SM-queries without label variables over type-only graphs. The algorithm assigns a "budget" to each query vertex based on its expected cost of processing, then re-costs query parts *adaptively* as it executes and learns more about the search space. This approach allows us to gracefully recover from situations where our cost expectation was inaccurate and to update our cost predictions based on what BudgetMatch encounters during processing. We show experimentally that on a real world social network database with 1.12B edges, BudgetMatch can answer complex subgraph queries in under one second and significantly outperform existing subgraph matching algorithms.

To simplify the presentation of the algorithm, we will represent a type-only social network database $\mathcal{S} = (V, E', \mathit{start}, \mathit{end}, \mathcal{P})$ as a graph $G = (V, E)$ where $E \subseteq V \times \mathbb{L} \times V$ is the set of labeled edges such that $e \in E' \Leftrightarrow (\mathit{start}(e), \mathit{type}(e), \mathit{end}(e)) \in E$. Moreover, we assume that queries do not allow label variables, and represent a query Q simply by a pair (V_Q, E_Q) where $E_Q \subseteq V_Q \times \mathbb{L} \times V_Q$ is the set of query edges.

5.1 Budget Query Answering

The BudgetMatch query answering algorithm is shown in Figure 5.1. It proceeds depth-first, matching query vertices with graph vertices one at a time until either a complete answer substitution has been found or a condition is met which allows the algorithm to abandon the current branch of the search tree and backtrack.

For each vertex z in the query, BudgetMatch maintains a triple (z, R_z, c_z) representing possible "matches" for z. c_z is the current expected cost of processing z. This quantity changes as the algorithm executes and searches more of G, thus gaining knowledge about both, the topology of the graph and the cost of matching z. The expected costs are compared when choosing the next query vertex to match in the search for answer substitutions. R_z stores potential matches for z. When z

is a constant in the query, R_z contains z itself; when z is a variable in the query, R_z contains a list of possible matches for z when z is initialized – some of these matches may be discarded as the algorithm proceeds – and is empty otherwise.

To ensure that our cost expectations are accurate, we assign a budget to each vertex before it is matched. If the true cost of matching the vertex exceeds that budget, we abort (at least temporarily) processing that node, update the expected cost and reconsider the vertex selection.

BudgetMatch works *irrespective* of what mechanism is used to initialize costs, update costs, and assign budgets to query vertices z (as long as those mechanisms are reasonable). We discuss some choices of cost and budget functions which we compare in our experiments.

Cost initialization

We initialize the cost of each query vertex via function *initialCost* in line 3 of BudgetMatch. *initialCost* can be defined in many ways. For instance, we could set it to be a fixed value $\lambda > 0$ (say, $\lambda = 5$ as in Figure 5.2(a)) for each vertex. Alternatively, we could use statistics about G to initialize costs. Specifically, let $degree(\ell, G) = \frac{|\{(v,\ell,w)\in E\}|}{|V|}$ for $\ell \in \mathbb{L}$ denote the average degree across all vertices in G for edges with type label ℓ. Using this statistic, we may define $initialCost(z, V_Q, G) = \lambda \times \sum_{(z,\ell,v)\in E_Q} degree(\ell, G)$ as the sum of the average degrees for all edge labels of edges incident on z in the query graph.

Updating budgets

We use a simple, multiplicative cost update function $updateCost_\mu(c_w) = \mu \times c_w$ for $\mu > 1$.

Assigning budgets

There are many ways to assign budgets to vertices. We propose the following four formulations and compare them in the experiments.

- $assignBudget_1(c_w, w, t, P) = c_w$.
- $assignBudget_2(c_w, w, t, P) =$

$$= \begin{cases} \frac{|R_v|\times c_v}{|R_w|} & \text{if } \exists v = \text{argmin}_{R_x\neq\emptyset, x\neq w}|R_x| \times c_x, \\ \infty & \text{otherwise.} \end{cases}$$

- $assignBudget_3(c_w, w, t, P) = |R_w| \times c_w - t$
- $assignBudget_4(c_w, w, t, P) =$

$$= \begin{cases} |R_v| \times c_v - t & \text{if } \exists v = \text{argmin}_{R_x\neq\emptyset, x\neq w}|R_x| \times c_x, \\ \infty & \text{otherwise.} \end{cases}$$

```
Global variables
$G$: Graph (index)
$A$: Answer set for $Q$ w.r.t. $G$.

Algorithm BudgetMatch
Input: Query $Q$

1   $A \leftarrow \emptyset$
2   for all $z \in V_Q$ /* Initialization of possible matches*/
3       $c_z \leftarrow initialCost(z, V_Q, G)$
4       if $z \in VAR_Q$ then $R_z \leftarrow \emptyset$
5       else $R_z \leftarrow \{z\}$
6   $P \leftarrow \{(z, R_z, c_z) \mid z \in V_Q\}$
7   selectVertex($Q$, $\emptyset$, $P$)

Algorithm selectVertex
Input: Query $Q$, partial substitution $\theta$, possible matches $P$

8   if $\exists (z, R_z, c_z) \in P \ : R_z \neq \emptyset$
9       $w \leftarrow \text{argmin}_{(z,R_z,c_z) \in P \ : R_z \neq \emptyset} |R_z| \times c_z$
10      $t \leftarrow 0$
11      $R'_w \leftarrow \emptyset$
12      for all $m \in R_w$ /* substitute possible match */
13          $b \leftarrow assignBudget(c_w, w, t, P)$
14          $s \leftarrow$ substituteVertex($Q$, $\theta$, $P$, $b, w, m$)
15          if $s > b$ then $R'_w \leftarrow R'_w \cup \{m\}$
16          $t \leftarrow t + s$
17      if $R'_w \neq \emptyset$
18          $c'_w \leftarrow updateCost(c_w)$
19          selectVertex($Q$, $\theta$,$P \setminus \{(w, R_w, c_w)\} \cup \{(w, R'_w, c'_w)\}$)
20  else
21      $A \leftarrow A \cup \{\theta\}$

Algorithm substituteVertex
Input: Query $Q$, partial substitution $\theta$, possible matches $P$,
       cost budget $b$, query vertex $w$, possible match $m$
Output: Consumed budget

22  retrieveVertex($G, m$)
23  if $w \in VAR$ then $\theta' \leftarrow \theta \cup \{w \rightarrow m\}$
24  $Q' \leftarrow Q$ ; for all $z \in V_Q$   $R'_z \leftarrow R_z$ /* copy */
25  $R'_w \leftarrow \emptyset$
26  $s \leftarrow 0$, $s_L \leftarrow \emptyset$
27  for all edges $e = (w, l, v) \in E_{Q'}$
28      if $l \in s_L$ then $r \leftarrow \infty$
29      else $r \leftarrow b - s$ ; $s_L \leftarrow s_L \cup \{l\}$
30      $N \leftarrow$ retrieveNeighbors($G, m, l, r$)
31      if $N =$ null then return $b + 1$
32      else if $r < \infty$ then $s \leftarrow s + |N|$
33      if $w = v$ /* treat self-loops as a special case */
34          if $m \notin N$ then return $s$
35      else
36          if $R'_v = \emptyset$ then $R'_v \leftarrow N$
37          else $R'_v \leftarrow R'_v \cap N$
38          if $R'_v = \emptyset$ then return s
39      remove $e$ from $Q'$
40  $P' \leftarrow \{(z, R'_z, c_z) \mid z \in V_Q\}$
41  selectVertex($Q'$, $\theta'$, $P'$)
```

Figure 5.1: BudgetMatch algorithm

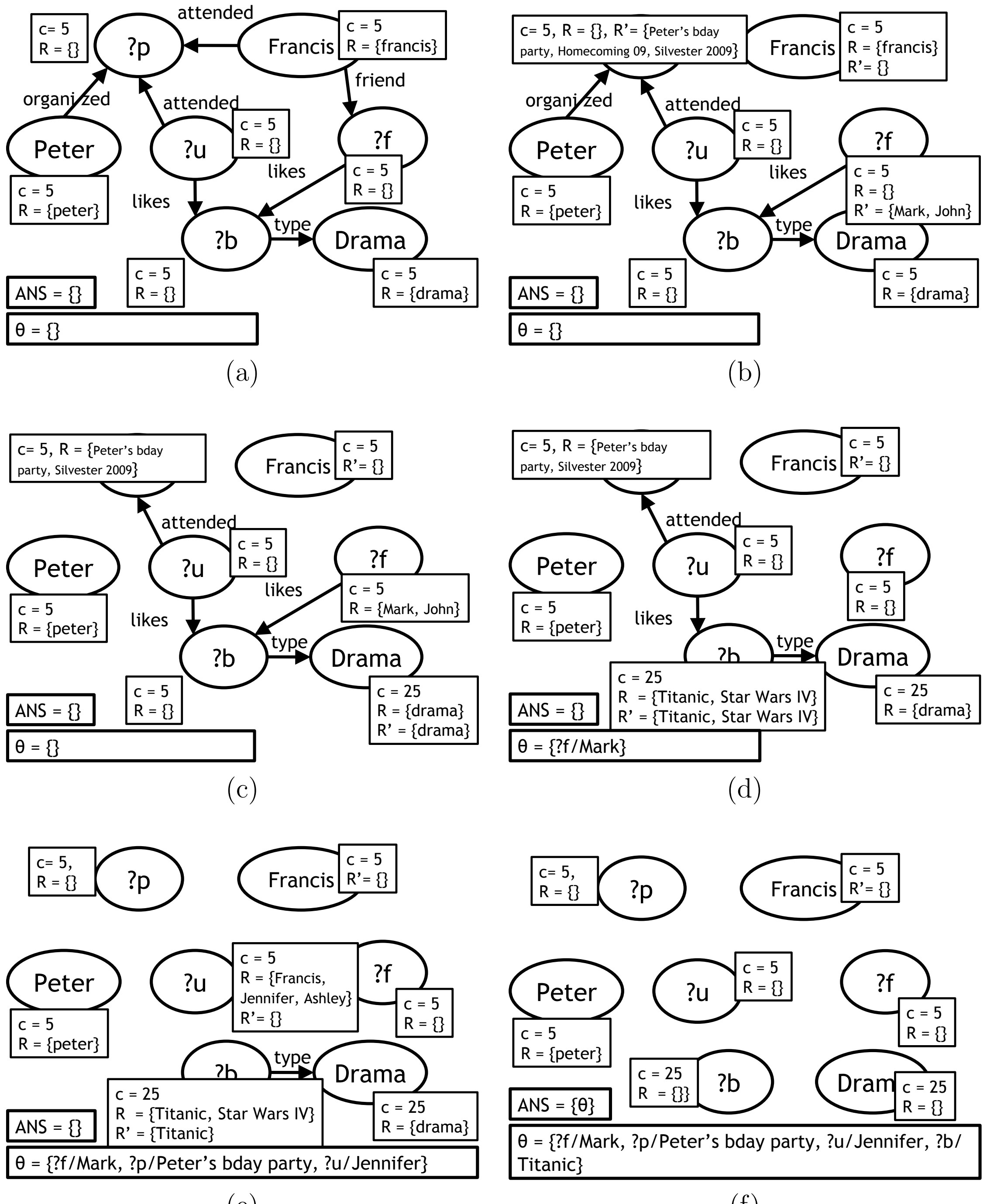
Budget function $assignBudget_1$ uses the expected cost for a query vertex as its budget. $assignBudget_2$ uses the average cost per match of the second lowest cost as the budget or infinity if no second lowest exists. The rationale behind this function is that we should afford our selected vertex with a total budget that equals the cost of the next best substitution vertex. Budget assignment functions $assignBudget_3$ and $assignBudget_4$ assign a total budget across all potential matches by considering the variable t which represents the total budget expense thus far. Function $assignBudget_3$ subtracts from the total expected cost $|R_w| \times c_w$ what has already been expended: t. Similar to function $assignBudget_2$, function $assignBudget_4$ uses the second lowest total cost or infinity.

5.2 Explanation of BudgetMatch

Now that we have introduced some basic terms relating to budgets and costs, we explain BudgetMatch using the example graph of Figure 1.7 and query of Figure 1.8. We use the $assignBudget_3$ function to assign budgets and $\mu = 5$ to update them in our example. To simplify the presentation of the algorithm, we assume undirected edges and the presence of least one constant vertex. Our implementation of BudgetMatch handles directed edges – addressing queries without constant vertices requires an additional index for edge lookups.

Figure 5.2(a) shows that our algorithm initialized all query vertices with cost 5. selectVertex looks at the three query vertices (Francis, Peter, and Drama) with non-empty R's which indicates that these potential match sets have been initialized. All these query vertices z have the same value of $|R_z| \times c_z$ – so one vertex (say Francis) is randomly picked to be w. The loop of Step 12 is executed just once as $R_{\text{Francis}} =$ Francis so m is set to Francis. $assignBudget_3$ assigns a budget of 5. We now execute substituteVertex which retrieves vertex m=Francis. Function retrieveVertex retrieves the label of vertex m from disk (m being its identifier). We now look at the edges incident on the query vertex $m =$ Francis in the query. There are two edges (Francis, friend, $?f$) and (Francis, attended, $?p$). Suppose the loop in line 27 considers the edge (Francis, friend, $?f$) first. By calling retrieveNeighbors(G,Francis, friend,5) it finds that there are two possible values for $?f$ – John and Mark. Line 32 sets $s = 0 + 2 = 2$ as we have two solutions and then $R'_{?f} = \{$John, Mark$\}$. After this, it *deletes* the edge (Francis, friend, $?f$) from the query because this edge has been processed. The second iteration of the loop at line 27 proceeds likewise for edge (Francis, attended, $?p$) but with an adjusted budget of $r = 5 - 2 = 3$, and $s_L = \{\text{friend}, \text{attended}\}$ is the set of retrieved edge labels. We set $R'_{?p} = \{\text{Peter's bday party}, \text{Homecoming 09}, \text{Silvester 2009}\}$ in Line 36 and remove the edge (Francis, attended, $?p$). Figure 5.2(b) shows the status at this time. We update the possible matches P' variable in Line 40 and recursively invoke the selectVertex function in Line 41.

At this stage, we have four query vertices (Peter, Drama, $?f$, $?p$) with initialized potential match sets R to choose from with cost values ($|R_z| \times c_z$) of 15, 5, 10, 5, respectively. Line 9 says we should choose either Peter or Drama – in our example, suppose Peter gets chosen. Processing vertex Peter is similar to processing Francis, with the only difference that the potential match set $R_{?p}$ has already been



attended
friend
organized
likes
type
c= 5, R = {}
c = 5, R = {francis}
c = 5, R = {peter}
c = 5, R = {drama}
c= 5, R = {}, R'= {Peter's bday party, Homecoming 09, Silvester 2009}
c = 5, R = {francis}, R'= {}
c = 5, R = {}, R' = {Mark, John}
c= 5, R = {Peter's bday party, Silvester 2009}
c = 5, R'= {}
c = 5, R = {Mark, John}
c = 25, R = {drama}, R' = {drama}
c = 25, R = {Titanic, Star Wars IV}, R' = {Titanic, Star Wars IV}
c = 25, R = {drama}
c = 5, R = {Francis, Jennifer, Ashley}, R'= {}
c = 25, R = {Titanic, Star Wars IV}, R' = {Titanic}
c = 25, R = {}
?p
?u
?f
?b
Francis
Peter
Drama
ANS = {}
ANS = {θ}
θ = {}
θ = {?f/Mark}
θ = {?f/Mark, ?p/Peter's bday party, ?u/Jennifer}
θ = {?f/Mark, ?p/Peter's bday party, ?u/Jennifer, ?b/Titanic}
(a)
(b)
(c)
(d)
(e)
(f)

Figure 5.2: Behavior of BudgetMatch on the example graph of Figure 1.7 and query of Figure 1.8

initialized and so we intersect it with the retrieved vertices in N (line 37) yielding $R'_{?p} = \{\text{Peter's bday party}, \text{Silvester 2009}\}$. In the next iteration, the lowest cost vertex is Drama. As $R_{\text{Drama}} = \{\text{Drama}\}$, the loop of line 12 is executed only once. *assignBudget* sets budget $b = 5$ and substituteVertex is invoked. We see that Drama has only one neighbor $?b$ in the query (line 27). We set $r = 5$, $s_L = \{\text{type}\}$ and $v = ?b$ in lines 27-29. We then invoke retrieveNeighbors$(G, \text{Drama}, \text{type}, 5)$ where 5 is the budget we allocate to this index retrieval. However, since Drama has a large number of neighbors, retrieveNeighbors returns null, because retrieveNeighbors is implemented to stop retrieving neighbors as soon as the budget is exceeded. *This is a critical pruning step because we do not want to process high degree nodes unless we have to.* Thus, in Line 31, we return 6 and abort the substituteVertex invocation, returning to line 14 where s is set to 6. As $s > b = 5$, we set $R'_{\text{Drama}} = \{\text{Drama}\}$. As the condition on line 17 is true, we update (line 18) the c_{Drama} to $\mu \times 5 = 25$. By assigning a high value to Drama, it is deferred for processing — the edge ($?b$, type, Drama) is *not* deleted. We then call selectVertex recursively in Line 19. Figure 5.2(c) shows the situation at this stage.

As both $?p$ and $?f$ have a cost value (line 9) of 10, we can choose arbitrarily – say $?f$. Processing $?f$ closely resembles the processing of vertex Francis but now we have two possible matches, John and Mark. Suppose we choose Mark first in the for-loop of line 12. In substituteVertex we extend the current (empty) substitution θ to map $?f$ onto Mark because $?f$ is a variable query vertex and initialize $R'_{?b} = \{\text{Star Wars IV}, \text{Titanic}\}$. In the next iteration, we choose $?b$. The first possible match for $?b$ that we explore is $m =$ Titanic and assign a budget of 5. However, when we try to retrieve the neighbors of Titanic from disk that budget is exceeded because Titanic has many fans in the database – so, null is returned. As we did for Drama, we abort the processing of $?b$ and update its expected cost to 25. Figure 5.2(d) shows the current stage of processing.

Now, $?p$ is the lowest cost query vertex which is selected next and we choose $m =$ Peter's bday party. Substituting m extends θ and initializes $R_{?u}$, thus $?u$ is selected next because its expected cost is $15 < 25$. When processing $?u$ we choose $m =$ Jennifer and retrieve all movies Jennifer likes on a budget of 5, intersecting those with $R'_{?b}$ which yields $R'_{?b} = \{\text{Titanic}\}$. This stage is shown in Figure 5.2(e). Finally, we process vertex $?b$ with single potential match Titanic and, lastly, Drama. At this point there are no more non-empty sets R_z and so the current substitution θ must be an answer which is added to A (line 21). This stage is shown in Figure 5.2(f). Now, the algorithm backtracks and tries the remaining potential matches for each query vertex – those will not produce any additional answer substitutions.

In our short sample run of the algorithm, BudgetMatch twice avoided retrieving the neighborhoods of high degree vertices and updated its cost expectations accordingly, thereby saving considerably in query answering time.

5.3 Experimental Evaluation

We implemented BudgetMatch in Java on top of the Neo4j graph database framework. Our implementation of $retrieveNeighbors(G, m, l, r)$ uses degree annotations of the vertices: if the degree of vertex m w.r.t. label l is larger than r, the function returns null, otherwise it iterates over all incident edges to retrieve the neighbors.

We compared BudgetMatch against three baseline subgraph matching algorithms, all built on top of the Neo4j database library to ensure fair comparison. The first baseline, which we denote SN-1, uses the subgraph matching component provided with Neo4j. The other two are implementations of the DOGMA query answering algorithm. The second baseline, denoted SN-2, does not employ statistics about the database; the third one, denoted SN-3, makes use of edge selectivity statistics for vertex selection.

We evaluated BudgetMatch on a large social network database extracted from the Delicious social bookmarking service capturing user posts, various attributes and tag assignments [156]. The database has approximately 1,122 million edges. The query evaluation benchmark consists of 9 subgraph matching queries; the total number of edges in the queries varied from 5 to 12. We designed queries by first fixing an initial structure and designating *constant* vertices. We then initialized those query vertices with vertex identifiers retrieved randomly from the respective databases. During this process, we excluded queries with an empty result set. As vertices were randomly selected, the selectivity of the queries varies with the degree and connectivity characteristics of the randomly chosen vertices.

All experiments were executed on a six-core AMD Opteron processor, a 15K RPM 300 GB SAS hard drive and 256 GB of RAM. However, we only allocated a maximum of 2GB of RAM to the Java virtual machine for all experimental trails. We ran the queries with cold and warm caches. For the cold cache mode, we started the database, used a dummy query to initialize it and then ran the benchmark. To warm the caches, we ran the benchmark once and then immediately ran it again in the same order, reporting the second query time.

We evaluated the BudgetMatch algorithm on all nine queries for each of the four budget assignment functions, each of the two initial cost functions with $\lambda = 100, 500, 2000, 15000$, and with $\mu = 10$. Out of these many configurations, we selected a representative set of 8 configurations. We compare these versions of the BudgetMatch algorithm against the SN-3 baseline (i.e. DOGMA plus selectivity statistics) which consistently outperformed the other two baselines on all queries. The query times (in milliseconds) are reported in Figure 5.3 and Figure 5.4 for cold and warm caches, respectively. All reported times are averaged over 10 independent runs with the two lowest and highest query times discarded to remove possible outliers.

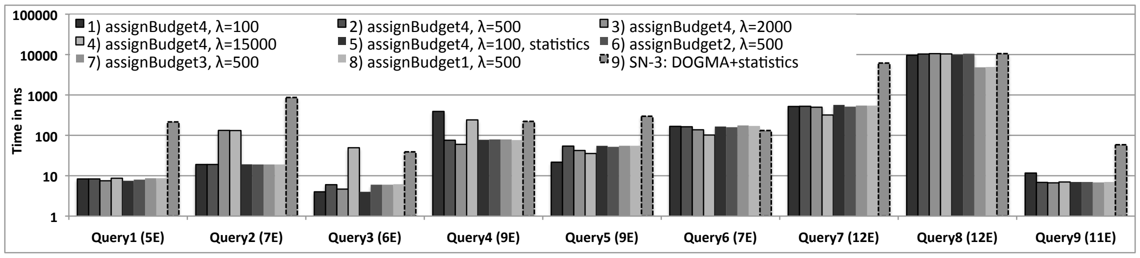


Figure 5.3: Query execution times (in milliseconds) for cold caches. E denotes number of edges in each query

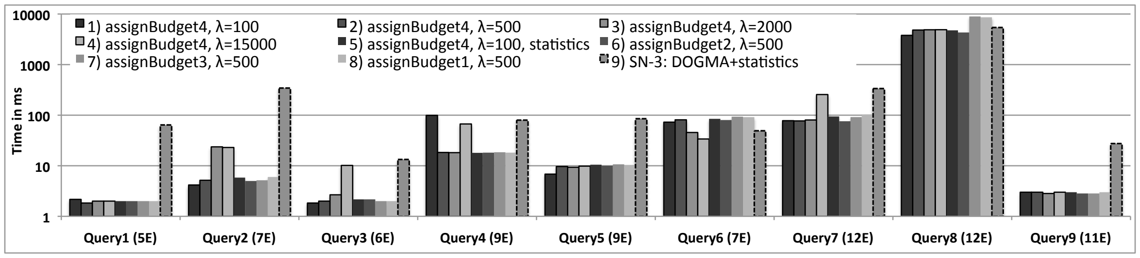


Figure 5.4: Query execution times (in milliseconds) for warm caches

We observe that the BudgetMatch algorithm outperforms the best baseline method by up to two orders of magnitude and is considerably faster on most queries. However, performance varies between algorithm configurations and queries. Comparing all configurations (also those not reported in the figures) for both warm and cold caches on all queries, we observed that the version of BudgetMatch using the $assignBudget_4$ function with constant initial cost function and $\lambda = 500$ (denoted configuration 2) performs best on average. It significantly outperforms all three baselines on all queries but one. The average performance improvements of configuration 2 of BudgetMatch compared to the three baselines for cold and warm caches are reported in Table 5.1.[1]

Table 5.1: Average performance improvements of configuration 2 of BudgetMatch

	SN-1	SN-2	SN-3
Cold	1,286,700%	1,198%	1,105%
Warm	4,479,400%	1,834%	1,418%

5.4 Related Work

Existing work on general graph data management and subgraph matching (*e.g.*, [29, 60, 77, 132, 165, 167]) typically use heuristics to predict the cost of answering strategies based on statistics about the database and the current state of query processing and then choose a strategy to minimize cost. However, due to the highly heterogeneous nature of network data [112] such predictions can become inaccurate. On the other hand, the models developed in the past can suggest appropriate choices for costing. For instance, [170] proposes to transform vertices into points in a vector space, thus converting queries into distance-based multi-way joins over the vector space. In [31] the authors propose a two-step join optimization algorithm based on a cluster based join index. GADDI is proposed in [164] that employs a

[1]The SN-1 baseline (which is Neo4j's subgraph matching component) does not seem to use a cost model and only terminated on 6 out of the 9 queries within one hour. We only compare the query times for those 6 queries.

structural distance based approach and a dynamic matching scheme to minimize redundant calculations. GADDI can handle graphs with thousands of vertices, which are common in many biological applications. In [165] the authors propose SUMMA, which improves over GADDI and employs more advanced indices, becoming capable to handle graphs with up to tens of millions of vertices. The algorithm in [167] employs an aggressive pruning strategy based on an index storing label distributions. In [107], the authors argue that existing indices over *sets* of data graphs do not support efficient pruning when they face graphs with tens of thousands of vertices. They propose an index that is specifically targeted at this query evaluation scenario.

Chapter 6
Cloud-Based Query Processing in Social Networks

In previous chapters, we have presented a social network database system and query answering algorithms operating on a single compute machine. In this chapter, we describe how to *distribute* social network database across multiple compute machines and answer SM-queries in parallel. One may wish to distribute a social network database for any of the following reasons:

1. **Size:** As online social networks grow to millions of users and records billions of interactions between those, the size of the resulting social network dataset can become too large to store on a single machine.

2. **Throughput:** A single machine is limited in the number of database requests it can handle in a certain time period. By distributing the database across multiple machines, the capacity and throughput of the entire database systems is increased. In particular database systems exposed to user via the web can experience very large workloads at times, exceeding more than 1000 requests per second. Clearly, a single machine is not well equipped to handle such a workload.

3. **Latency:** By executing queries in parallel across many machines, the time it takes to retrieve all query answers can be significantly reduced for certain types of queries.

It could be argued that using a more powerful compute machine would also improve storage capacity, throughput, and latency without adding the complexity of data distribution and parallel query answering. This strategy is called *scaling up*: investing in more powerful hardware to address rising database demands. However, the scale-up strategy is limited and costly. It is limited by the current state of the art hardware components. There certainly does not exist a single high performance computing machine that could handle the workloads Google or Facebook are confronted with. More importantly, buying a high performance database server is much more expensive than buying commodity hardware of equal compute power. This is the primary reason that *scaling out*, that is, distributing the workload across multiple, smaller machines, is the preferred strategy for most demanding applications.

In this chapter, we present the COSI system[1] which distributes a social network across multiple compute machines and answers SM-queries in parallel using asynchronous query answering algorithm that does not rely on central orchestration. The COSI system addresses the following two challenges of a distributed social network database:

1. **Data Distribution:** How do we distribute the graph data across the compute machines such that we balance the workload and ensure fast query answering?

[1] COSIstands for "cloud-oriented subgraph identification"

2. **Query Answering:** How do we answer SM-queries efficiently when the data is distributed across multiple machines without introducing synchronization overhead?

Before we answer those questions in Section 6.2 and 6.4 respectively, we first describe the overall architecture of the COSI database system.

6.1 COSI Architecture

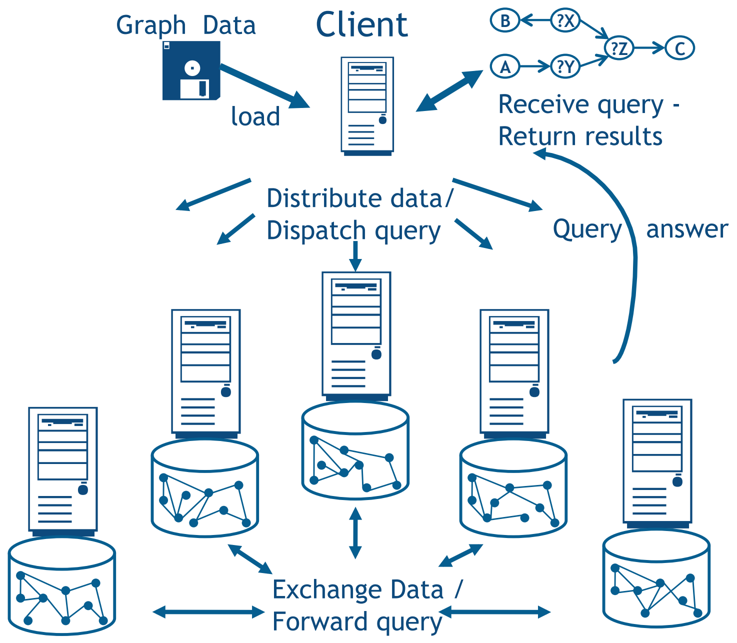


Figure 6.1: Architecture of COSI

Figure 6.1 shows a schematic view of the the architecture of COSI. We assume the existence of k compute machines in our cluster, where k is chosen depending on the performance and scalability requirements of a particular application. In Figure 6.1, $k = 5$ and the compute machines are shown in the lower half of the figure. Each of those machines stores a fragment of the social network in a local social network database and responds to query requests specifically addressed to it. The architecture and system we present in this chapter does not depend on any particular local social network database.
A client machine, shown at the top of Figure 6.1 interacts with the database cluster by either loading and storing graph data in the distributed social network database or asking SM-queries. Once a client issues a request to the database cluster, the compute machines coordinate internally to fulfill the request and return one or

multiple answers to the client as a response upon completion. The communication infrastructure ensures that there are no communication bottlenecks.

6.2 Partitioning Social Networks across a Compute Cluster

We now address the first question: How do we distribute a social network across a compute cloud so that we can efficiently process subgraph matching queries. In partitioning the social network data, we follow two objectives:

1. All k storage machines should store roughly the same amount of data to balance the load across machines.

2. The social network partition should minimize the expected query time.

At a high level, we achieve these objectives as follows. First, we transform the social network $\mathcal{S} = (V, E, start, end, \mathcal{P})$ into a simple weighted graph $\mathcal{W}(\mathcal{S})$. Intuitively, the weight of an edge $e = (u, v)$ in the simple weighted graph $\mathcal{W}(\mathcal{S})$ refers to the sum of the probability that v will be retrieved immediately after u and *vice versa* when an arbitrary query is processed. Intuitively, if this probability is (relatively) high, then the two vertices should be stored on the same storage machine. We then use these to partition the social network across the k storage machines so that expected communication costs are minimized.

Throughout this section, we assume there is a probability distribution $\mathbb{P}$ over the space of all queries. Intuitively, $\mathbb{P}(Q)$ is the probability that a random SM-query posed to a social network is Q. For any real world online social network like Facebook or Orkut, $\mathbb{P}$ can be easily learned from frequency analysis of past query logs. We now formally derive an optimal partitioning strategy from our knowledge of $\mathbb{P}$.

6.2.1 Probability of Vertex (Co-)Retrievals

A *query plan* $qp(Q)$ for a SM-query Q is a sequence of two types of *operations*: the first type retrieves the neighborhood of vertex v (from whichever compute machine it is on), and the second type performs some computation (e.g. check a selection condition or perform a join) on the results of previous operations. This definition is compatible with most existing definitions of query plans in the database literature.

Definition 6.1 (Query trace). Suppose $x = qp(Q)$ is a query plan for a query Q on a social network database $\mathcal{S}$. The *query trace* of executing x on $\mathcal{S}$, denoted $qt(x, \mathcal{S})$, consists of (i) all the vertices v in $\mathcal{S}$ whose neighborhood is retrieved during execution of query plan x on $\mathcal{S}$, and (ii) all *pairs* (u, v) of vertices where immediately after retrieving u's neighborhood, the query plan retrieves v's neighborhood (in the next operation of x). □

Traces contain consecutive retrievals of vertex neighborhoods. Note that our definition of query trace not only include the retrieval of individual vertices, but also consecutive retrievals of vertex neighborhoods. The reason is that this allows us to potentially store neighborhoods of both u and v on the same storage node, thus avoiding unnecessary communication when executing the query plan incurred by "jumping" from one storage machine to another.

When processing a query, we make the reasonable assumption that index retrievals are cached so that repeated vertex neighborhood retrievals are read from memory and hence the query trace $qt(x, \mathcal{S})$ can be defined as a set rather than as a multiset. The probability distribution $\mathbb{P}$ on queries can be used to infer a probability distribution $\tilde{\mathbb{P}}$ over the space of feasible query plans. $\tilde{\mathbb{P}}(x) = \sum_{Q \in \mathcal{Q}: qp(Q)=x} \mathbb{P}(Q)$. This says that the probability of a query plan is the sum of the probabilities of all queries which use that query plan.[2] We now define the probabilities of retrieval and co-retrieval as follows.

Probability of retrieving vertex v. The probability, $\mathbb{P}(v)$, of retrieving v when executing a random query plan is $\sum_{x \in qp(\mathcal{Q}): v \in qt(x,\mathcal{S})} \mathbb{P}(x)$. Thus, the probability of retrieving v is the sum of the probabilities of all query plans that retrieve v.

Probability of retrieving v immediately after u. The probability $\mathbb{P}(u, v)$ of retrieving v immediately after u is $\sum_{x \in qp(\mathcal{Q}): (u,v) \in qt(x,\mathcal{S})} \mathbb{P}(x)$. This says that the probability of retrieving v immediately after u is sum of the probabilities of all query plans that retrieve v immediately after u. Neither definition requires that u or v actually occur in the result set of query Q – just that they were considered or "visited" during query execution (and possibly discarded).

6.2.2 Optimal Partitioning

Based on the probabilities of retrieval, we can associate a simple, weighted, undirected graph $\mathcal{W}(\mathcal{S})$ with the social network $\mathcal{S} = (V, E, \mathit{start}, \mathit{end}, \mathcal{P})$. The simple graph $\mathcal{W}(\mathcal{S}) = (V, E_{\mathcal{W}}, \omega_{\mathcal{W}})$ is a complete graph on V, i.e. $E_{\mathcal{W}} = \{(u, v) \mid u, v \in V\}$, where $\omega_{\mathcal{W}}((u, v)) = \mathbb{P}(u, v) + \mathbb{P}(v, u)$.

The following important theorem shows that the minimal edge cut partitions of $\mathcal{W}(\mathcal{S})$ correspond to the partitions of $\mathcal{S}$ across k storage machines that minimize expected cost of executing SM-queries based on the probability distribution $\mathbb{P}$. A partition of a graph or network is a partition of its vertices V into blocks. A collection of blocks $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a partition of V iff 1) blocks contain only contain vertices and all of them: $\bigcup_{i=1}^{k} P_i = V$ and 2) all blocks are disjoint: $P_i \cap P_j = \emptyset \ \forall\, i \neq j$. Given a partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ for the vertices in a simple graph $G = (V, E, \omega)$, we say that an edge $e = (u, v)$ is *cut* if its end vertices are contained in different partition blocks, i.e. $u \in P_i$, $v \in P_j$ and $i \neq j$. The *edge cut* of a partition is the set of all edges that are cut. The size of an edge cut is the sum of the weights of the edges that are in the edge cut. A partition $\mathcal{P}$ is said to be a *minimum edge cut partition* iff there is no other partition $\mathcal{P}'$ such that the the size of the edge cut of $\mathcal{P}'$ is less than that of $\mathcal{P}$. Minimum edge cut partitions may not be unique for a given graph G.

Since $\mathcal{W}(\mathcal{S})$ and $\mathcal{S}$ share the same set of vertices, any partition of $\mathcal{W}(\mathcal{S})$ is

[2]In the rest of the paper, we will abuse notation and denote both PDFs by $\mathbb{P}$.

also a partition of $\mathcal{S}$.

Theorem 6.1. *Assuming uniform costs (across all storage machines) for retrieving a vertex neighborhood and sending a message to a remote machine, the partition of $\mathcal{S}$ which minimizes the total query execution time of a random query coincides with the partition that minimizes the cost of a minimal edge cut of $\mathcal{W}(\mathcal{S})$.* □

Proof. Suppose we choose a query Q uniformly at random. The total query execution time for Q is the sum of computation time each of the k storage machines devotes to finding all answer to query Q. The time each storage machine spends on answering Q can be decomposed into three parts: 1) the time to retrieve vertex information and vertex neighborhoods of all vertices $v \in qt(qp(Q), \mathcal{S})$ from disk, 2) local processing of query Q (see Section 6.4 for more details), and 3) communication with other storage machines when forwarding or receiving a query. The total time needed to retrieve vertex information and neighborhoods as well as the time for local processing is independent of the partition, assuming all storage machines are equally fast at retrieving vertex neighborhoods from disk and equally fast at local processing. Thus we need a constant amount of time to retrieve $ngh(v)$ for all $v \in qt(qp(Q), \mathcal{S})$ irrespective of the partition block containing v. In other words, how we partition the graph does not impact the first two parts of the total query answering time – those remain constant. Hence, the only variable component of the total query time is part 3, the time spend communicating with other machines, and the partition that minimizes that summand of the overall time equation is the one that minimizes the overall query execution time across the k storage machines.

Let $TC(Q)$ denote the total communication time across all storage machines for answering query Q. We fix some partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ of $\mathcal{S}$. Suppose $(u, v) \in qt(qp(Q), \mathcal{S})$ with $u \in P_i$, $v \in P_j$, and $i \neq j$, that is, we consider the case of vertex co-retrieval where the co-retrieved vertices are in different partition blocks. In this case, the neighborhood retrieval for vertex u is followed by that of v. As v is in a different partition block residing on another storage machine, we need to send a message to this node at a communication cost of c, which adds to the total query execution time and depends on the partition $\mathcal{P}$. We assume uniform communication cost across all storage machines and c denotes the sum of communication costs of both storage machines i and j. We define the indicator random variable $X_{(u,v)} = c$ if $(u, v) \in qt(qp(Q), \mathcal{S})$, and $X_{(u,v)} = 0$ otherwise. Let $E_{\mathcal{P}} = \{(u, v) | u, v \in V, u \in P_i, v \in P_j, i \neq j\}$ denote the set of all pairs of vertices in the social network that are assigned to distinct partition blocks. Using standard expected values, the total expected cost of communication for Q is

$$\begin{aligned}
\mathbb{E}\left[TC(Q)\right] &= \mathbb{E}\left[\textstyle\sum_{(u,v)\in E_{\mathcal{P}}} X_{(u,v)}\right] \\
&= \textstyle\sum_{(u,v)\in E_{\mathcal{P}}} \mathbb{E}\left[X_{(u,v)}\right] \\
&= \textstyle\sum_{(u,v)\in E_{\mathcal{P}}} \mathbb{P}(u, v) \times c
\end{aligned}$$

where we use the linearity of expectation in the derivation.

Since c is a constant, minimizing $\mathbb{E}\left[TC(Q)\right]$ is equivalent to minimizing $\sum_{(u,v)\in E_{\mathcal{P}}} \mathbb{P}(u,v)$. Recall that $\mathcal{W}(\mathcal{S})$ is a fully connected, simple, undirected, weighted graph on the vertices V of $\mathcal{S}$. Hence, $\mathcal{P}$ is also a partition of $\mathcal{W}(\mathcal{S})$. Therefore, for each $(u,v),(v,u) \in E_{\mathcal{P}}$ (not that $E_{\mathcal{P}}$ must be symmetric by definition) there is an undirected edge $(u,v) \in \mathcal{W}(\mathcal{S})$ with weight $\omega_{\mathcal{W}}((u,v)) = \mathbb{P}(u,v) + \mathbb{P}(v,u)$. Hence, $\sum_{(u,v)\in E_{\mathcal{P}}} \mathbb{P}(u,v)$ is equal to the edge cut of $\mathcal{W}(\mathcal{S})$ with respect to partition $\mathcal{P}$. Hence, the partition that minimizes the edge cut of $\mathcal{W}(\mathcal{S})$ is the one that minimizes the $TC(Q)$ and therefore minimizes the total query execution time, which completes the proof. $\square$

6.3 Partitioning in Practice

The problem of finding a graph partition with minimal edge-cut is known to be NP-complete [54]. Hence, the focus of this section lies on the development of tractable algorithms which develop *good* partitions with respect to the cost model derived in Theorem 6.1. A particular challenge is balancing the computational complexity with the quality of the partitions produced. The higher the quality of the partitions produced, the higher the query throughput. But higher complexity of the partitioning algorithm entails a significant up front cost, since the size of the social network database to be partitioned can be very large. In this section we first consider the scenario of individual edge insertion before we develop partitioning algorithms for the more interesting case of batch edge insertion.

For both scenarios we focus on partitioning strategies that permanently assign vertices to storage machines. Once vertex v gets assigned to storage node i, v cannot be moved to another storage machine during the lifetime of the database. Permanent vertex assignment greatly simplifies database maintenance. More importantly, it allows for faster query execution because we can store the storage node identifier i with each reference to vertex v thereby avoiding the cost of an additional lookup. On the other hand, fixing vertices to partition blocks obviously constrains partition algorithms in their ability to minimize the edge cut. On complex subgraph queries, additional retrieval operations needed to look up the storage machine of vertex v add significantly to the overall query execution time which justifies our constraint on vertex placement despite its adverse affects on edge cut minimization.

6.3.1 Individual Edge Insertion

First, we analyze the simple case where edges are inserted one at a time with significant delay. When edges are added individually, any insertion algorithm needs to update the network partition considering just a single edge. Suppose some edge e with $start(e) = u, end(e) = v$ is inserted into $\mathcal{S}$. Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a partition of the social network $\mathcal{S}$. The partial vertex assignment function which assigns vertices to their partition block is denoted by $a : V \to \mathcal{P}$. If $a(v)$ is undefined for a vertex v we call v *new*. If both, $a(u)$ and $a(v)$ are defined, then the permanent vertex assignment assumption dictates that the triple is inserted on the storage machines that host partition blocks $a(u)$,$a(v)$ respectively. If either end vertex is new, we must assign it to a partition block, for which we suggest two approaches.

The *random assignment strategy* chooses one of the k partition blocks uniformly at random for each edge that has at least one new end vertex (as start or end vertex). It then assigns all new vertices introduced by the edge to that partition block. The random strategy is simple and leads to balanced partitions under mild assumptions about the social network database $\mathcal{S}$. However, the random strategy is naive, because it does not aim at minimizing the edge cut, and therefore only serves as a baseline for our experiments in Section 6.5.

Theorem 6.1 clearly shows that we can transform our problem into one of minimal edge cut computation. In order to actively minimize the edge cut, we need to take the connectedness of newly introduced vertices into account. For instance, suppose an edge $e = (u, v)$ is inserted where v has already been assigned to a partition block, then we should assign vertex u to $a(v)$ as well. However, such an approach would lead to extremely unbalanced partitions. To reflect the competing objectives of balance and minimal edge cut, we introduce the novel notion of a *vertex force vector*.

Definition 6.2 (Vertex force vector)**.** Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be a partition of $\mathcal{W}(\mathcal{S})$ and consider any block P_i. The *vertex force vector*, denoted $|\vec{v}|$, of any vertex $v \in V$ is a k-dimensional vector where $|\vec{v}|[i] = f_{\mathcal{P}}(\sum_{x \in ngh_{\mathcal{W}}(v) \cap P_i} \omega_{\mathcal{W}}(v, x))$ and $f_{\mathcal{P}} : \mathbb{R}^+ \to \mathbb{R}$ is a function called the *affinity measure*. □

A vertex force vector intuitively specifies the "affinity" between a vertex and each partition block as measured by the affinity measure $f_{\mathcal{P}}$. An affinity measure takes the connectedness between a vertex v and the respective partition block as an argument. The vertex force vector captures the strength with which each partition block "pulls" on the vertex and is used as the basis for a vertex assignment decision. If an inserted edge introduces a new vertex v, we first compute the vertex force vector $|\vec{v}|$ and then assign v to the partition block P_j where $j = argmax_{1 \leq i \leq k} |\vec{v}|[i]$. COSI uses an affinity measure that is a linear combination of three factors. We discuss the choice of coefficients in the experimental section.

Connectedness Obviously, evaluating the connectedness of a vertex to a partition block, i.e. the argument to the affinity measure, is crucial for edge cut minimization.

Imbalance Balanced partitions lead to even workload distribution across the storage nodes which entails maximum resource utilization. In order to achieve balanced partitions, we need a measure for partition imbalance. Let $|P_i|_E = \sum_{x \in P_i} deg(x)$ denote the number of edges in partition block P_i and therefore the storage consumption on storage machine i. A reasonable measure of partition imbalance would be the standard deviation of $\{|P_i|_E\}_{1 \leq i \leq k}$ across all partition blocks. However, measuring imbalance in absolute terms can be misleading when data is being loaded. Suppose we have a partition with just two blocks $\mathcal{P} = P_1, P_2$ where P_1 contains 20 and P_2 60 edges. Hence, the standard deviation of partition block size is 20. Is $\mathcal{P}$ an imbalanced partition? If we expect only a hundred edges, then it is acutely

imbalanced. However, if we know the database to grow to millions of edges, then this initial statistic does not suggest imbalance at all.
For a better measure of imbalance we assume that the user provides a ball-park estimate of the total number of edges, T to be loaded into the database. This estimate need not be accurate and can be changed during the lifetime of the database. Then a reasonable measure of imbalance is the standard deviation of $\{\frac{|P_i|_E}{T}\}_{1\leq i\leq k}$.

Excessive Size In addition to imbalance, we regulate the size of partition blocks by comparing the actual size of a block to its expected one. If a block grows beyond its expected size, we want to punish such growth more aggressively than imbalance does alone by reducing the affinity further according to the metric $\min(-\frac{|P_i|_E - \frac{T}{k}}{T}, 0)$.

In Section 6.5 we define the affinity measures used in our experiments based on the three metrics introduced above.

6.3.2 Batch Edge Insertion

Individual edge partitioning strategies are severely restricted due to the limited information about the connectedness of a newly inserted vertex. However, in most applications edges are inserted in groups rather than one at a time. This is particularly true, when large social network datasets is loaded into the database at once. Hence, we now turn our attention to partitioning strategies for batch insertion.

In the batch edge insertion scenario, the user adds an entire social network subgraph S to the database at once. As before, the challenge is to efficiently assign newly introduced vertices to partition blocks such that we minimize edge cut and achieve balanced partitions.

A naive GreedyInsert insertion algorithm iterates over all new vertices v: for each vertex v it computes the vertex force vector and assigns v to the block P_i such that $|\mathbf{v}|[i]$ is maximal. This greedy algorithm is used as a baseline in our experiments.

Prior studies on finding minimal edge cuts demonstrate that greedy approaches do not compute good partitions. Graph partitioning is a widely studied problem with many application. Related work on graph partitioning include spectral clustering [69], randomized minimum cut schemes [75], mathematical programming [117], and a variety of other approaches (see [49] for a survey). Most proposed graph partitioning algorithms, however, are inapplicable to social network datasets since their runtime complexity is quadratic in the number of vertices or worse making them infeasible w.r.t. the size of social networks. Multi-level partitioning schemes have been shown to be fast and produce good partitions on a large range of graphs [76]. However, these algorithms partition the entire graph at once and do not operate incrementally as we require. We now present a multi-level social network partitioning algorithm that is designed to build a partition incrementally by distinguishing new from previously assigned vertices and generalizing the vertex force vector concept.

Our COSI_Insert algorithm leverages graph *modularity* [13] to identify a strongly connected subgraph that is loosely connected to the remaining graph in $\mathcal{W}(\mathcal{S})$. However, modularity cannot be used blindly as our balance requirement must also be met.

Definition 6.3 (Modularity)**.** The *modularity* of a partition $\mathcal{P}$ of a simple, undirected, weighted graph $G = (V, E, \omega)$ is defined as

$$mod(\mathcal{P}) \quad = \quad \sum_{P \in \mathcal{P}} \left(\frac{W(P,P)}{2\,|E|} - \frac{deg_\omega(P)^2}{(2\,|E|)^2} \right)$$

where $deg_\omega(v) = \sum_{x \in V} \omega(v,x)$ is the weighted degree of vertex v, $W(X,Y) = \sum_{x \in X, y \in Y} \omega(x,y)$ is the sum of edge weights connecting two sets of vertices $X, Y \subset V$, and $deg_\omega(X) = \sum_{x \in X} deg_\omega(x)$ is the weighted degree of a set of vertices $X \subset V$.$\square$

Intuitively, blocks with high modularity are densely connected subgraphs which are isolated from the rest of the graph. Our algorithm iteratively builds high modularity blocks starting on $\mathcal{W}(\mathcal{S})$ and then assigns all vertices in a block to one storage machine based on the vertex force vector. Let $B \subset V$ be a set of vertices. We generalize the notion of a vertex force vector to sets of vertices B by defining $\left|\vec{B}\right|[i] = f_{\mathcal{P}}\left(\sum_{v \in B} \sum_{x \in ngh_{\mathcal{W}}(v) \cap P_i} w_{\mathcal{W}}(v,x)\right)$. The intuition behind our partitioning algorithm is that assigning vertices at the aggregate level of isolated and densely connected blocks yields good partitions because (i) we respect the topology of the graph, (ii) most edges are within blocks and therefore cannot be cut, and (iii) force vectors of sets of vertices combine the connectedness information of many vertices leading to better assignment decisions.

The COSI_Partition algorithm (Fig. 6.2) constructs two assignments functions:

1. **Vertex assignment function** c assigns vertices to modularity blocks in $\mathcal{C}$ aiming to maximize the modularity measure defined above.

2. **Cluster assignment function** b assigns modularity blocks in $\mathcal{C}$ to partition blocks in $\mathcal{P}$ according to the vertex force vector.

COSI_Partition initially assigns each vertex to be its own block via the assignment function c. The assignment function from initial blocks to the partition $\mathcal{P}$ is initialized to $\emptyset$ and subsequently updated to account for prior vertex assignments according to the parameter function α. The algorithm then repeatedly iterates over all new vertices u and determines whether moving u into any neighboring block increases modularity. If so, u is moved into the block which yields the largest increase and the assignments are updated. We continue iterating over all new vertices until the number of vertex moves l in the previous iteration falls below some threshold δ. This first part of the algorithm determines modular blocks by iteratively moving individual vertices to maximize modularity improvement. $\Delta M(u,t)$ denotes the change in modularity resulting from moving vertex u into the block containing vertex t. $\Delta M(u,t)$ can be computed efficiently as:

$$\Delta M(u,t) = \frac{2W(\{u\},c(t)) - 2W(\{u\},c(u)-\{u\})}{2|E|} - \frac{2deg_w(u)[deg_\omega(c(t)) - deg_\omega(c(u)-\{u\})]}{(2|E|)^2}$$

```
Algorithm COSI_Partition
Input: Simple, undirected, weighted graph $G = (V, E, \omega)$,
       partial assignment function $\alpha : V \to \mathcal{P}$
Output: Total assignment function $\alpha : V \to \mathcal{P}$

1  $(c : V \to \mathcal{C}) \leftarrow \emptyset$; $c(v) = \{v\} \forall v \in V$
2  $(b : \mathcal{C} \to \mathcal{P}) \leftarrow \emptyset$
3  for all $v \in$ domain$(\alpha)$
4    $b(c(v)) = \alpha(v)$
5  repeat
6    $l \leftarrow 0$
7    for all $u \in V -$ domain$(\alpha)$
8      $x \leftarrow \text{argmax}_{t \in ngh(u)} \Delta M(u,t)$
9      if $\Delta M(u,x) > 0$
10       $c(x) \leftarrow c(x) \cup \{u\}$
11       $c(u) \leftarrow c(u) - \{u\}$
12       $c(u) \leftarrow c(x)$
13       $l \leftarrow l + 1$
14 until $l < \delta$
15 $G_\mathcal{C} \leftarrow (\mathcal{C}, E_\mathcal{C}, \omega_\mathcal{C})$ where $E = \{(x,y) \mid \exists u \in c^{-1}(x), v \in c^{-1}(y) : (u,v) \in E\}$
16 $\omega_\mathcal{C} \leftarrow E_\mathcal{C} \to \mathbb{R}$ where $\omega_\mathcal{C}((x,y)) = \sum_{u \in c^{-1}(x)} \sum_{v \in c^{-1}(y)} \omega((u,v))$
17 if $|\mathcal{C}| < \theta_1$ or $\frac{|\mathcal{C}|}{|V|} > \theta_2$
18   for all $i = 1, \ldots, n$
19     $e_i \leftarrow 0$
20     for all $C$ randomly chosen from $\mathcal{C} -$ domain$(b)$
21       $b_i(C) \leftarrow P_m$ where $m = argmax_{1 \le j \le k} \left|\vec{C}\right| [j]$
22   $b \leftarrow b_j$ where $j = \text{argmin}_{1 \le i \le n} \sum_{C_s, C_t \in \mathcal{C}, b_i(C_s) \neq b_i(C_t)} \omega_\mathcal{C}((C_s, C_t))$
23 else
24   $b \leftarrow$ COSI_Partition$(G_\mathcal{C}, b)$
25 for all $v \in V$
26   $\alpha(v) \leftarrow b(c(v))$
27 return $\alpha$
```

Figure 6.2: COSI_Partition algorithm

The algorithm constructs a new graph $G_\mathcal{C}$ from the original graph G by collapsing all vertices assigned to the same block during the modularity finding phase.

If the graph $G_{\mathcal{C}}$ has less than some threshold number of vertices θ_1 or if its size is not significantly smaller than the original graph G, we stop and assign vertices to partition blocks. Otherwise, we call COSI_Partition recursively on the collapsed graph $G_{\mathcal{C}}$ to find modular blocks comprised of modular blocks, thereby building multiple levels of modular graphs. If the collapsed graph is small enough, we sequentially assign each vertex in $G_{\mathcal{C}}$ to the partition block which maximizes the force vector component as we did before. We repeat this process n times using different random permutation of the vertices and choose the assignment that minimizes the edge cut. Finally, we map the vertex assignment onto the original graph G, thereby projecting it down one level. COSI_Partition is guaranteed to respect prior vertex assignments (as specified by parameter α). Moreover, the size of the collapsed graph $G_{\mathcal{C}}$ is a constant factor smaller than the original graph and hence the size of the input graph decreases exponentially as the function calls itself recursively which contributes to the speed of the algorithm.

6.4 Parallel SM-Query Answering

The COSI_basic parallel algorithm, shown in Fig. 6.3, operates asynchronously and in parallel across all storage machines. A user issues query Q to the client machine which "prepares" the query. In particular, it selects one constant vertex c from Q and determines the storage machine S that hosts c using the function call location(c). The prepared query is then forwarded to S.

The algorithm proceeds depth first, substituting vertices for variables in Q one at a time. We maintain a set of result candidates R_z for each variable vertex z in Q. This set is either uninitialized or is a superset of all result substitutions for that particular variable vertex. The storage machine query algorithm assumes there is an index retrieval function retrieveNeighbors(D, v, l) that retrieves $ngh_l(v)$ in sorted order from the local index D (which could be implemented many ways) on the slave. For now, h_{opt} arbitrarily chooses the next vertex to be substituted. A better definition of h_{opt} will be provided in the COSI_heur algorithm.

Incoming queries come with a selected variable vertex to be instantiated with a vertex ID. The algorithm updates the candidate result sets by retrieving the neighborhood of the newly substituted vertex from the index. Since the result sets are sorted, this operation takes linear time. It then checks if any results have been found or whether the current substitution cannot yield a valid result. All query results are sent to the client which returns them to the user. If neither condition holds, the algorithm selects the next variable vertex v' to be substituted and forwards the query to those storage machines that host potential substitution candidates for v'.

The COSI_basic algorithm uses two optimizations to reduce messaging costs: (i) if the source and destination of the message in Lines 22-23 coincide, then the algorithm recursively calls itself with updated data structures rather then sending the message; (ii) the query processor groups all messages with the same query ID targeted at the same storage machine and sends them in one message, thus reducing

```
     Algorithm COSI_basic

     On client machine
     Input: Graph query $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q)$
     Output: Answer substitutions $A$, i.e. set of substitutions $\theta$ s.t. $Q\theta$ is a subgraph of $\mathcal{S}$

1    for all $z \in V_Q \cap VAR$
2      $R_z \leftarrow$ null /* no candidate substitutions for any vars in the query initially */
3    for all $z \in V_Q \cap (\mathcal{S} \cup \mathcal{V})$
4      $R_z \leftarrow \{z\}$/* constant vertices only have themselves
                  as possible substitutions */
5    $qid \leftarrow$ next query ID /* uniquely identifies query */
6    $c \leftarrow \text{argmin}_{v \in V_Q} h_{opt}(v)$/* pick optimal vertex to process next */
7    send $(Q, qid, c, \{(c \rightarrow c)\}, \{R_z\})$ message to location($c$) storage machine

     On storage machine
     Input: Graph query $Q$, query ID $qid$, designated vertex $c$,
          partial substitution $\theta$, candidate sets $\{R_z\}$, local index $D$

8    for all edges $e$ with $start(e) = c$ and $end(e) = v \in V_Q \cap VAR$
9      if $R_v =$ null
10       $R_v \leftarrow$ retrieveNeighbors($D, c, \lambda_Q(e)$) /* use index to retrieve
                  all nbrs of $c$ with same label as $e$ */
11     else
12       $R_v \leftarrow R_v \cap$ retrieveNeighbors($D, c, \lambda_Q(e)$) /* restrict space
                  of possible subst. for $z$ */
13   if $\nexists e \in E_Q : start(e), end(e) \in V_Q \cap VAR$/* have we found an answer? */
14     $\{v_1, \ldots, v_l\} \leftarrow \{v \mid v \in V_Q \cap VAR \wedge v \notin \text{domain}(\theta)\}$
15     for $(s_1, \ldots, s_l) \in R_{v_1} \times R_{v_2} \times \ldots \times R_{v_l}$
16       $\theta' \leftarrow \theta \cup (v_1 \rightarrow s_1) \cup \ldots \cup (v_l \rightarrow s_l)$
17       Send $(qid, \theta')$ to client
18   else if $\exists w \in V_Q : R_w = \emptyset$/* reached dead end? */
19     return "NO"
20   else
21     $w \leftarrow \text{argmin}_{v \in V_Q \wedge |R_v| > 0} h_{opt}(v)$/* pick optimal vertex to process next */
22     for all $m \in R_w$
23       $\theta' \leftarrow \theta \cup \{w \rightarrow m\}$
24       send $(Q, qid, m, \theta', \{R_z\})$ message to location($m$) storage machine
```

Figure 6.3: COSI_basic algorithm

communication cost. COSI_basic does not rely on central orchestration – it uses depth first search so the branches of the search tree are traversed in parallel while ensuring that no branch gets explored multiple times. After forwarding the prepared query to a storage machine, the client waits for incoming results of that query and forwards those to the user. As we explore branches in parallel, the client cannot be notified when the search for query results has completed. Keeping track of the current number of parallel executions for each query would introduce significant synchronization cost. Instead, the client keeps track of the time t_{last} at which the last result of a running query has come in. If the difference between the current time and t_{last} exceeds a threshold, the client asks all storage machines for a list of query IDs of all currently running queries. The client merges these lists and closes all queries whose IDs are not contained. To avoid the case where a query is being forwarded to another storage machine at the very moment that the client asks for all query IDs, each storage machine keeps query IDs in their local list up to a certain grace period.

6.4.1 The COSI_heur algorithm

The choice of the next variable to be instantiated has profound implications on the running time of COSI_basic, as some substitutions yield larger branching factors in the search than others. COSI_heur handles this by choosing the variable vertex v' which has the lowest cost according to function h_{opt} . We now discuss the some statistics that are related to the cost of a variable vertex in terms of query execution time.

First, to reduce branching factor, we could choose the variable vertex v' with the smallest number of result candidates. This heuristic only considers the branching factor of the immediate next iteration, but is nevertheless an important metric to consider in the cost heuristic. Second, whenever we instantiate a vertex on a remote partition block, we have to send a message to the appropriate slave which is expensive. Therefore, we consider the fraction of result candidates which are not stored locally as a cost metric. When we have to send a query to remote slaves for further processing, we would like to distribute the workload evenly across all slaves. Hence, we also analyze the distribution of result candidates by slave via the cost metric

$$ds(v) = \sqrt{\sum_{1 \leq i \leq k} \left(|R_v^i| - \frac{|R_v|}{k} \right)^2}$$

where R_v^i is the set of result candidates for vertex v restricted to those which reside on slave node i. Finally, we define

$$h_{opt}(v) = |R_v| \times (1 - \frac{|R_v^l|}{\alpha \times |R_v|}) \times (1 + \beta \times \frac{ds(v)}{|R_v|})$$

where l is the ID of the local slave node and α and β are constants that determine how much the model favors locality over parallelism. Our experiments study how α, β impact query run-times.

6.5 Experimental Evaluation

COSI is implemented in Java. We developed a communication infrastructure for the compute nodes based on the Java NIO libraries which is used to send the graph data during the loading and the queries during the query answering stages. The communication infrastructure handles contention at individual nodes and variations in network latency. It is optimized to ensure that the client's requests for outstanding queries are answered quickly. COSI uses a modified version of the DOGMA graph database [17] for local storage of the graph data. DOGMA itself interfaces against BerkeleyDB 4.8 [12] for on-disk storage. However, we emphasize that COSI is implemented independently from the underlying graph storage backend and can utilize alternative databases. The client handles all user requests and maintains a vertex label lookup table on disk using BerkeleyDB. To reduce storage size as well as message size, all vertex labels are mapped onto unique IDs at the client. When a query is issued to the client, it first retrieves the vertex ids for the labels before forwarding the query to the storage machine. Conversely, it looks up the vertex labels for all ids contained in a query answer before forwarding it to the issuing client. Moreover, to facilitate efficient retrieval, vertex IDs include a namespace which uniquely identifies the storage machine; thus, vertices can be directly located without a routing table.

In our experiments, we used a cluster of 16 compute nodes out of which one served as a client and the remaining 15 nodes served as storage nodes. All storage nodes had an identical hardware configuration with two Intel Xeon Quad Core 2.3 GHz Processors, 8 GB of RAM, and 73 GB SAS 10k RPM hard drive. The client's hardware differed slightly with 16 GB of RAM and two 146 GB 10k RPM SAS disks in RAID1 mirror configuration.

We fixed the coefficients for the affinity measure by hand. Both, the imbalance and excessive size metric, were given an equal weight of 1. The connectedness measure was set relative to the number of edges we considered per batch. We experimented with different batch sizes and found best performance for half a million edges.

By the definition of COSI_basic query answering algorithm in Figure 6.3, any two co-retrieved vertices must be connected. Hence, we set the probability of vertex co-retrieval to 0 for all unconnected pairs of vertices. The co-retrieval probability for connected vertices was set to 1.[3] Ideally, these probabilities would be derived from an analysis of query execution logs, however, such data was not available to us.

We used the social network data set studied in [103] for experiments. This data set contains 778M edges and describes personal relationships and group memberships crawled from Facebook, Orkut, Flickr, and LiveJournal.

To evaluate the performance of our proposed partitioning and query answering strategies, we designed 11 queries of varying size. In designing the queries, we first fixed the query graph topology and then randomly chose edge and vertex labels from the social network dataset while ensuring that the resulting query has a non-empty result set. The size of a query graph is measured by the number of edges and vertices it contains. We list some of the queries used in our experiments in the appendix.

[3]Note, that multiplying the probabilities of co-retrieval by a constant factor does not affect the edge cut minimization problem.

6.5.1 Performance of COSI_Partition

Fig. 6.4 compares COSI_Partition's performance with that of the GreedyInsert algorithm. To validate our experiments, we used a random partitioning scheme, which assigns vertices to storage machines uniformly at random, as the naive baseline in our experiments and report all results in comparison to this baseline. COSI_Partition achieves a substantial 36% improvement in edge cut over the naive baseline at a total running time of 10.5 hours for all 778M edges. GreedyInsert only achieves a marginal improvement in edge cut. COSI_Partition significantly outperforms greedy batch insertion by 33% with only slightly higher imbalance as measured in the standard deviation in partition block size relative to average size of a block. We observe that COSI_Partition substantially outperforms both baselines on the important edge cut quality metric.

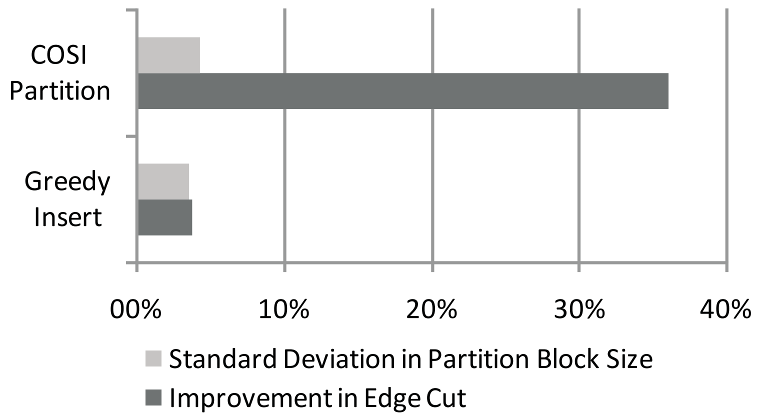


Figure 6.4: Comparison of partitioning methods

6.5.2 Query Answering

Fig. 6.5 compares COSI_basic against COSI_heur for three different parameter settings of the heuristic h_{opt}: ($\alpha = 1.2, \beta = 0.1$) which strongly favors locality over parallelism, ($\alpha = 8.0, \beta = 5.0$) which strongly favors parallelism over locality, and ($\alpha = 2.0, \beta = 0.5$) which balances locality and parallelism. The queries have increasing complexity as measured by the number of edges (E) and variables (V) in the query graph. All query times were averaged across 6 independent runs with complete system restarts after each run to empty caches. Note, that the graph is plotted in logarithmic scale to accommodate the huge differences in query times.

COSI_heur drastically outperforms COSI_basic by up to 4 orders of magnitude on all but two queries, and the performance gap seems to grow exponentially with the query complexity. A close look at the difference in performance between the variants of COSI_heur reveals that the third configuration outperforms the first one on 9 queries, with a tie on the remaining 2, and outperforms the second configuration on 8 queries, being slower only on 3. These results suggest, that a balanced choice

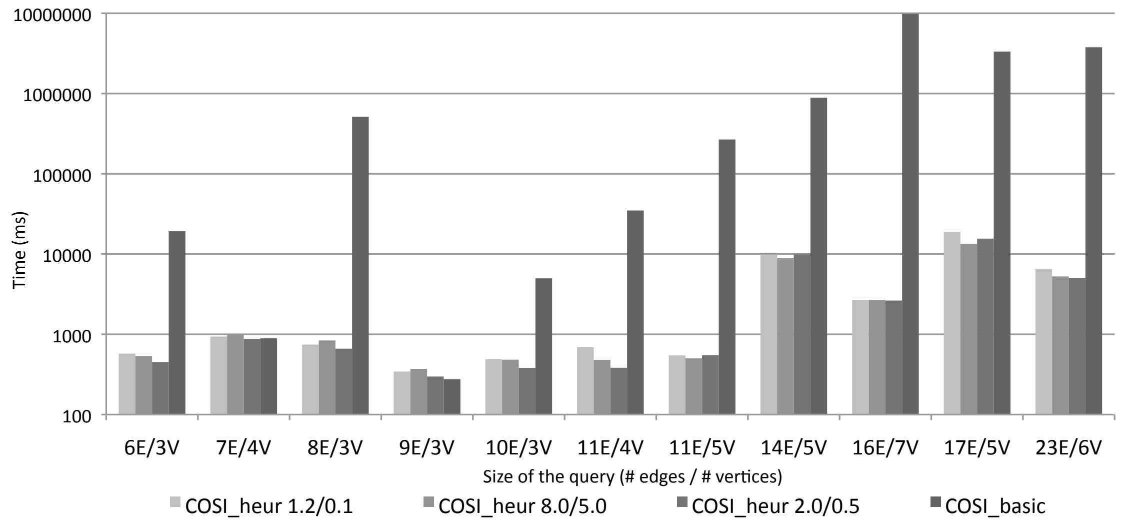


Figure 6.5: Query times by query answering algorithm on the 778M edge Facebook/LiveJournal/Orkut data set

of parameters leads to a better h_{opt}.

6.5.3 Partition Impact

We derived theoretically in Theorem 6.1 that smaller edge cut leads to shorter expected query times. To verify the theorem experimentally, we compare the average query answering times over the partition generated by COSI_Partition against that produced by GreedyInsert. We have shown above that COSI_Partition produces partitions with lower edge cut which should reflect in shorter query times. We used the COSI_heur query answering with the third parameter configuration to compute the times for our set of queries and report the results in Fig. 6.6. The results support our hypothesis by showing that the partition produced by COSI yields significantly better query times for complex queries.

6.6 Related Work

COSI is the first cloud-oriented, distributed graph database we are aware off. YARS2 [67] is a parallel RDF triple store which utilizes multiple machines to store large RDF data sets. However, query answering is executed on a central machine which merely communicates with the storage machine to retrieve sets of triples. As such, query answering relies on central orchestration and is not parallelized. Also, YARS2 [67] does not address the issue of data partitioning for fast query execution.

Recently, the commercial OrientDB database suite extended its offering to include a cloud enabled graph database. Much like COSI, the network data is distributed flexibly across multiple storage machines. However, OrientDB does not support subgraph matching queries nor does it provide graph partitioning strategies

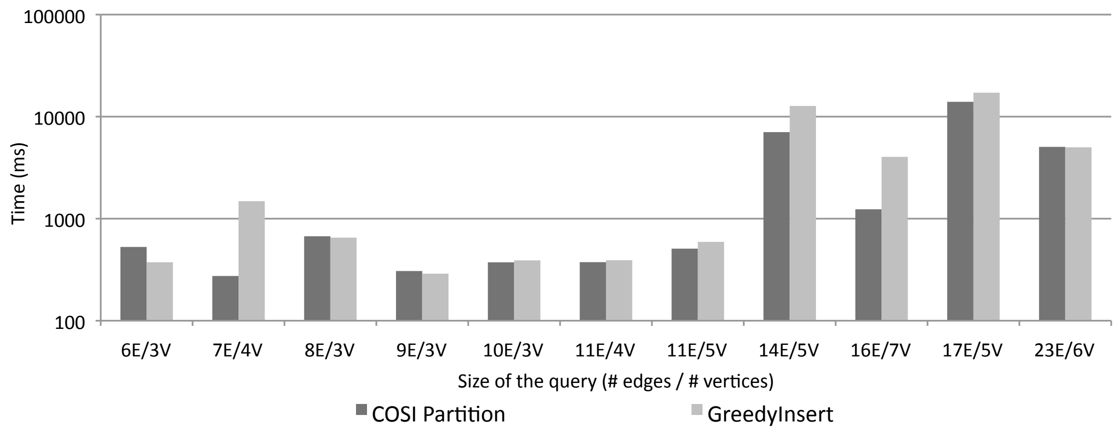


Figure 6.6: Query times by partitioning method on the 778M edge Facebook/LiveJournal/Orkut data set

to optimize throughput.
Other commercial graph database systems, such as Neo4j, or AllegroGraph can be "sharded", that is, partitioned, across multiple machines to increase scalability. However, sharding strategies have to be implemented by the database administrator and are not optimized automatically.

Increasing scalability through parallelization has been extensively studied for relational database systems [38]. However, for RDBMS the unit of analysis is the relational tuple and, hence, one aims at dividing relational tables across multiple machines to enable parallel execution of SQL operations. Such partitioning strategies typically "decluster" the data set [99]. In contrast, relationships between resources are the unit of analysis in the RDF model and therefore we aim at "clustering" resources to avoid excessive communication cost.

To complete our review of related work, we point out that our problem formulation is significantly different from distributed RDF stores and federated query answering (e.g. [123, 145]) as well as P2P RDF stores [25] because we assume control over data placement and the communication framework.

Chapter 7
View Maintenance

Some subgraph matching queries, like the movie recommendation query on the social network shown in Figure 1.8, are *one-time* queries: A user creates the query to search for some specific information in the social network and then discards it or reuses it infrequently. Many user issued queries are of this nature. However, there are subgraph matching queries for which we want to maintain the answer as the underlying graph data changes. Consider the *72 fraud detection query shown in Figure 1.6. A phone company would like to track the answers to this query on a continuous basis and be alerted to new answers almost immediately, rather than having to query over and over again. Maintaining the answer to a subgraph matching query is called *view maintenance* and the subject of this chapter.

There are primarily two reasons for maintaining the answer to a query:

1. **Monitoring** the answers to a subgraph matching query as the social network changes in order to be immediately alerted to changes in the answer set or to automatically process new or removed answers.

2. **Caching** the answer to a frequently issued subgraph matching query. In scenarios where multiple users interact with one social network database, the same query might be issued many times by various users and applications. Recomputing the answer for each query request can be too costly and exhaust the database's capacity. By maintaining the answer set for such a query, any query request can be directly satisfied by returning the materialized view.

In this chapter, we are interested in developing the techniques needed to support social network *view servers* that *simultaneously* monitor large numbers of diverse views over large social networks. For instance, a marketer for Sony's Bravia TVs in India might want to continuously track all mentions of the Bravia made on Twitter by people in India who have more than 1000 followers and who work for a newspaper. A law enforcement officer might wish to monitor a photo-sharing or tagging site for all people who have "favorited" three of more photos that have been reported by at least two others as containing inappropriate child content. Such people may be potential pedophiles. A music company might wish to identify all individuals who have shared more than k music files containing a certain watermark. This might indicate people who are engaging in illegal file sharing.

Views and view maintenance have also been studied in the context of graph and/or RDF databases [64, 168, 152, 71, 97]. In this chapter, we build upon this past work on view maintenance in graph data to solve a different problem: suppose an social network database $\mathcal{S}_t$ (at time t) is stored on a view server with a set $\mathbf{Q} = \{Q_1, \ldots, Q_k\}$ of registered views that need to be tracked over time. Furthermore, suppose we know the answer to each view Q_i at time t. Suppose now that some *updates* occur at time t – how can the view server incrementally compute the answer to the views in $\mathbf{Q}$ change so that at time $(t+1)$, they represent the correct answer w.r.t. the database $\mathcal{S}_{t+1}$ that results after the updates? This problem is extremely important because, e.g., over 60M tweets are posted daily – we would like to incrementally compute these results rather than computing them from scratch.

7.1 Motivating Example

Figure 7.1 shows a small social network whose nodes represent individuals, companies, articles, messages, and topics. The links represent relationships between nodes such as employee-employer relationships, reference relations, publish relations, and tweet/retweet relationships. The example network is similar to the company social network in Figure 1.9 but is now presented in the context of view maintenance.

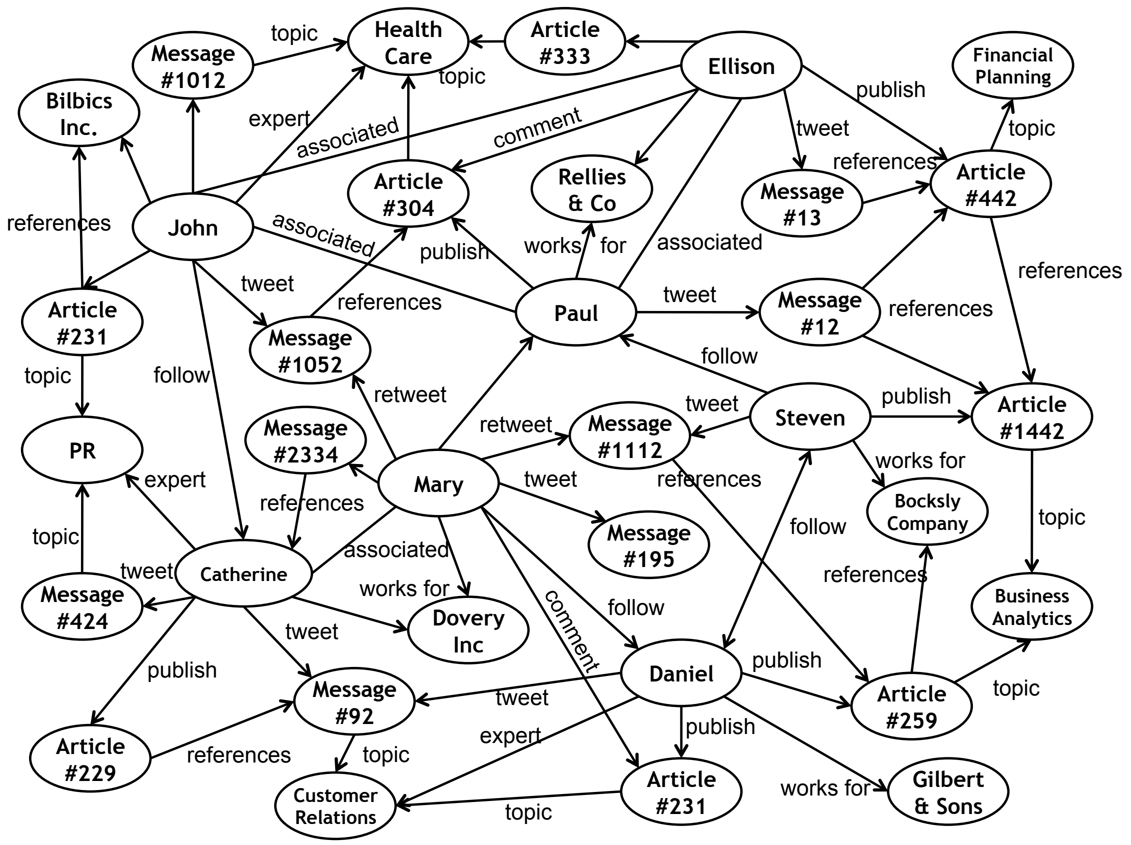


Figure 7.1: Business communication social network example

A recruiter might want to find individuals with expertise in *health care* and *business analytics* using query Q1 shown in Figure 7.2. The person of interest is denoted by the vertex labeled *?person* (question marks denote variables). The query considers having published an article on the topic of health care which is referenced by an expert on the subject as an indication of knowledge (left part of the query graph). Furthermore, it considers referencing an article on the topic of business analytics by a person who follows the individual of interest as an indication of being connected to the business analytics community (right part of the query graph).

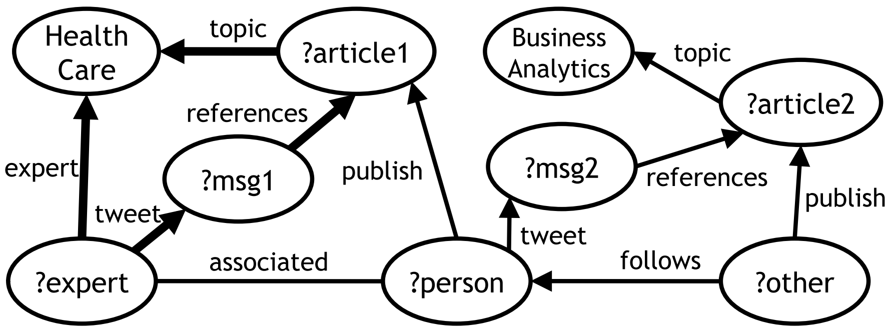


Figure 7.2: Example query Q1

Subgraph matching can also be used by an analyst for knowledge discovery. Query Q2 in Figure 7.3 searches for all authors and articles on the topic of Health Care which have been commented on by an individual who publishes on the subject and which has been referenced by an expert on health care.

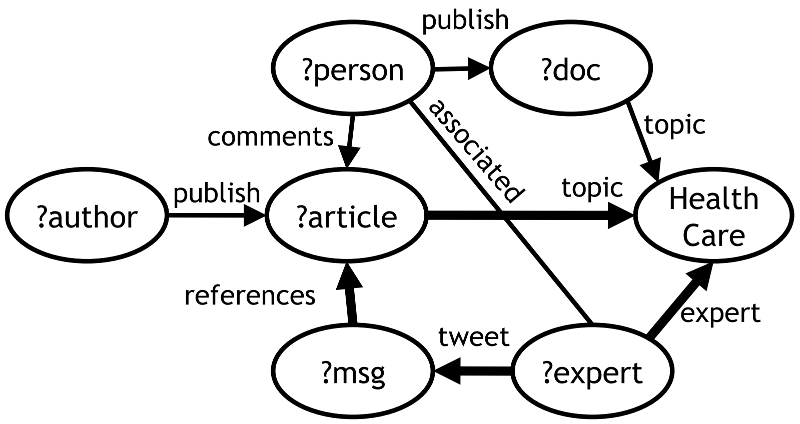


Figure 7.3: Example query Q2

To date, there is no approach to the problem of simultaneous view updates of multiple views. Past work only deals with one view at a time. In this paper, we make the following main contributions. In Section 7.3, we develop the concept of a *merged view* that leverages *common subgraphs* amongst the views in **Q** to identify a "center" edge that view maintenance will focus on. There can be many possible merged views – we define the concept of an optimal merged view and show that finding it is NP-hard. We then define the AddView algorithm that can be used to compute an optimal merged view. In Section 7.4 we propose a MultiView algorithm that uses the merged graph to incrementally update multiple materialized views. Section 7.5 describes the results of detailed experiments we have conducted using 6 real-world datasets, the largest of which has over 540M edges and 11M vertices.[1] We also describe the query generator we used to generate a wide range of queries

[1]We consider two small networks – a physics collaboration network and the Enron email network,

for testing. On average, our algorithms are 570% faster than if we use a naive view maintenance strategy.

7.2 Basic View Maintenance

In this section, we first define views on social network databases before we review a naive view maintenance algorithm to update *one* view when the social network database changes. Throughout this chapter, we assume that the social network database $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is fixed.

Definition 7.1 (Subgraph Matching View)**.** A *subgraph matching view* Q is defined via a subgraph query Q for which all query answers are maintained and materialized for the lifetime of the defined view[2].

Thus, a subgraph view is a query for which we can directly retrieve all answers without further processing of the query or accessing the core database. We will often abuse notation and refer to a subgraph query Q as a view when justified by the context. When Q is a view, the query's answer set $\mathsf{AS}(Q, \mathcal{S})$ is stored and maintained on the view server while the underlying social network database $\mathcal{S}$ changes.

When a view Q is first registered with a social network database $\mathcal{S}$, it has to be executed to find the initial query answers $\mathsf{AS}(Q, \mathcal{S})$. For the initial view construction, any existing subgraph matching algorithm can be used. To avoid recomputing a view's answer set entirely when the social network database changes, *incremental view maintenance* strategies have been proposed [64].

Suppose an edge e is added to the database $\mathcal{S}$ with registered view Q. The basic incremental view maintance algorithm ViewBasic shown in Figure 7.4 first determines if the answer set of Q is affected by the addition of e and, if so, substitutes e into Q, executes the partially substituted query on $\mathcal{S}$, and adds the result to the answer set of the view Q.[3]

The ViewBasic algorithm updates the current answer set of Q by iterating over all edges e_Q in Q which have the same edge label as the added edge e. For each edge, the algorithm first checks whether the vertices of e_Q are constants and – in this case – whether they are identical to the respective vertices of e (lines 3–4). If not, the edge e cannot possibly be a match for e_Q and we continue. Otherwise, an identical copy $\tilde{Q}$ of Q is created without e_Q. Since e is a match for e_Q, we replace the vertices of e_Q by the respective vertices from e in $\tilde{Q}$ (lines 6–7). Now, we execute $\tilde{Q}$ against the updated social network database $\tilde{\mathcal{S}}$ using *any subgraph matching algorithm* subgraphMatch to find all matches to add to the view increment ΔAS. Finally, the view is updated by adding ΔAS to the current answer set.

As an example, suppose the edge $e =$ (*John, tweet, Message#1012*) has just been added to the social network database shown in Figure 7.1 and we update the

two medium networks from Youtube and Flickr, and two large networks from LiveJournal and Orkut.

[2]Views are also referred to as *standing queries*.

[3]For compactness, this work focuses on incremental view maintenance when edges are added to the social network database. View maintenance for edge removal is completely analogous.

```
Algorithm ViewBasic
Input: View $Q = \langle Q \rangle$, updated database $\tilde{\mathcal{S}} = \langle \tilde{V}, \tilde{E} \rangle$,
       added edge $e = (u, l, v)$

1  $\Delta AS \leftarrow \emptyset$
2  for all $e_Q \in E_Q$ s.t. $label(e_Q) = l$
3     if $start(e_Q) \in \tilde{V} \ \wedge \ start(e_Q) \neq u$ then continue
4     if $end(e_Q) \in \tilde{V} \ \wedge \ end(e_Q) \neq v$ then continue
5     $\tilde{Q} \leftarrow$ copy($Q$) $\setminus \{e_Q\}$
6     replace $start(e_Q)$ by $start(e)$ in $\tilde{Q}$
7     replace $end(e_Q)$ by $end(e)$ in $\tilde{Q}$
8     $\Delta AS \leftarrow \Delta AS \cup$ subgraphMatch($\tilde{Q}, \tilde{\mathcal{S}}$)
9  AS($Q, \tilde{\mathcal{S}}$) $\leftarrow$ AS($Q, \mathcal{S}$) $\cup \Delta AS$
```

Figure 7.4: ViewBasic algorithm

view defined by query Q2 in Figure 7.3. The edge $e_Q = (?expert, tweet, ?msg)$ is the only one in Q2 which matches the label of e. Since both vertices of e_Q are variables, we match it with e and remove it from the copied query $\tilde{Q}2$. The resulting query $\tilde{Q}2$ is shown in Figure 7.5. Executing the subgraph matching query $\tilde{Q}2$ against the updated social network database yields exactly one match (θ_1 = {*?author/Paul,?person/Ellison, ?expert/John, ?article/Article #304, ?msg/Message #1012, ?doc/Article #442*}) which is added to the materialized view.

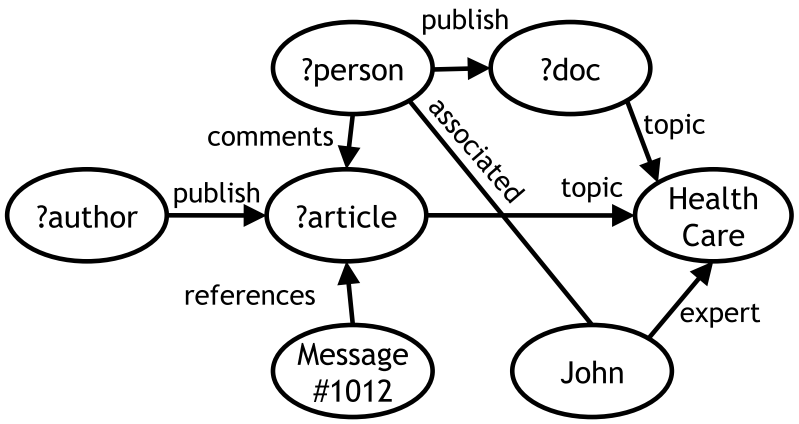


Figure 7.5: Modified query $\tilde{Q}2$

Proposition 7.1. *The ViewBasic algorithm terminates and correctly updates the answer set of Q. Its worst-case time complexity is $O(2^{|VAR_Q|})$.*

It should be noted that any modified query $\tilde{Q}$ contains at least two constant vertices (from the matched edge e) and is therefore anchored. This allows us to use the efficient subgraph matching algorithms and index structures proposed, e.g., in [157, 67, 17, 18, 23, 2].

ViewBasic is a *standard* incremental view maintenance algorithm similar to those proposed in prior work on view maintenance for graphs, with minor deviations due to differences in data model (e.g. graph or tree structured) and motivation (e.g. focus on particular classes of queries). It is the state of the art algorithm for maintaining subgraph matching views on arbitrary graph-structured databases and used as the baseline in this work. In our experiments in Section 7.5 we compare this incremental view maintenance strategy with our novel MultiView algorithm, which we will present next.

7.3 View Merging

Given a set $\mathbf{Q}$ of views, we now study how these views can be "merged" into a single view that leverages isomorphic subgraphs amongst different views in order to (hopefully) reduce the expected cost of maintaining the views when the social network database is updated.

Definition 7.2 (Edge Annotated View)**.** If Q is a view and $e \in E_Q$ is an edge, then $\langle Q, e\rangle$ is an *edge annotated view* (EA-view). For a label $l \in L$, $Q^l = \{\langle Q, e\rangle \mid e \in E_Q \text{ s.t. } label(e) = l\}$ denotes the set of EA-views for Q w.r.t. l. If $\mathbf{Q}$ is a set of views, then $\mathbf{Q}^l = \bigcup_{Q \in \mathbf{Q}} Q^l$.

For example, suppose $\mathbf{Q} = \{Q1, Q2\}$ as described in Section 7.1. In this case, $\mathbf{Q}^{tweet} = \{\langle Q1, (?expert, tweet, ?msg1)\rangle, \langle Q1, (?person, tweet, ?msg2)\rangle, \langle Q2, (?expert, tweet, ?msg)\rangle\}$.

We now come to the most important definition of this paper which uses a notion of a "center edge".

Definition 7.3 (Merged View)**.** Suppose $\mathbf{Q}$ is a set of views and l is an edge label. A *merged view of* $\mathbf{Q}^l$ is a 5-tuple $M^l = \langle V_M, E_M, \mathsf{c} \in E_M, \lambda, \Phi = \{\phi_i\}_{i=1}^m\rangle$ where:

- $V_M \subset V \cup VAR$, $E_M \subseteq V_M \times L \times V_M$ is a labeled graph.
- $\phi_i : VAR_{Q_i} \to V_M \cap VAR$ is an injective mapping that takes each variable vertex in view Q_i and specifies a corresponding variable vertex in V_M such that $E_{Q_i}\phi_i \subseteq E_M$.[4] Intuitively, this ensures that each edge in Q_i is "mapped" by ϕ_i to an edge in E_M;
- $\lambda : E_M \to 2^{\mathbf{Q}^l}$ assigns a nonempty set of EA-views from $\mathbf{Q}^l$ to each edge $e \in E_M$; $\lambda(e) = \{\langle Q_i, e_i\rangle \mid \exists \tilde{e} \in E_{Q_i} \text{ s.t. } \tilde{e}\phi_i = e\}$. This condition says that $\lambda(e)$ is the set of all EA-views $\langle Q_i, e_i\rangle$ such that Q_i contains an edge that is mapped to e by ϕ_i;

[4]$E_Q\phi$ is the result of applying the substitution ϕ to both vertices involved in each edge in E_Q.

- $e_i\phi_i = \mathsf{c}$ for all $i \in [1, m]$ – i.e. the edge e_i labeled l in view Q_i must be mapped to the "center edge" c by ϕ_i (as a consequence, the center edge must also have the label l). Note that ϕ_i is free to map other edges in Q_i (even those labeled l) to other edges in E_M.

The above definition allows multiple possible merged views of a set $\mathbf{Q}^l$ as shown in the example below.

Example 7.1. Suppose $\mathbf{Q} = \{Q1, Q2\}$ are the two views in Section 7.1. We denote the 3 EA-views in $\mathbf{Q}^{tweet}$ as $AV_1 = \langle Q1, (?expert, \text{ tweet}, \ ?msg1)\rangle$, $AV_2 = \langle Q1, (?person, \text{ tweet}, \ ?msg2)\rangle$, and $AV_3 = \langle Q2, (?expert, \text{ tweet}, \ ?msg)\rangle)$. Figure 7.6 shows merged views of $\mathbf{Q}^{tweet}$. Each mapping ϕ_i in the first merged view (ϕ'_i in the second) is associated with EA_i. These mappings map variable vertices in the views to variable vertices in the merged view – and by extension, they map edges as well. These edge mappings are drawn in the figure.

As a set $\mathbf{Q}$ of views can be merged in many ways, we need to define how good a merged view is from the point of view of minimizing compute time. In order to define the score of a merged view to capture this intuition, we first capture the score of an edge. First, a method to score a merged view should capture edge overlap – the larger the sets $\lambda(e)$ are, the more edges from the different views in $\mathbf{Q}$ we can evaluate by one edge evaluation in the merged view. Second, we plan to maintain views when updates occur by looking first at the "center edge". Thus, we need to look at the edge overlaps of paths from the center edge to the edge whose score we are computing.

Definition 7.4 (Edge Score)**.** The *score* of an edge $e \in E_M$ for a merged view M^l is defined as $score_{M^l}(e) = |S|$ where S is the maximal subset of $\mathbf{Q}^l$ such that (i) $S \subseteq \lambda(e)$ and (ii) there exists an undirected path $p = \langle \mathsf{c}, \dots, e_i, \dots, e\rangle$ in M^l such that $\forall e_i \in p$, $S \subseteq \lambda(e_i)$.

The edge score measures the overlap quality of a single edge in the merged view. We aggregate edge scores to compute the overall merged view score as follows.

Definition 7.5 (Merged View Score)**.** The *score* of a merged view M^l is defined as $score(M^l) = \sum_{e \in E_M} score_{M^l}(e)^2$.

The score of a merged view squares edge scores to reward high overlap. A merged view of $\mathbf{Q}^l$ is *optimal* if there is no other merged view of $\mathbf{Q}^l$ with a strictly higher score.

Example 7.2. Returning to our example, the solid edges in Figure 7.6(top) have an edge score of 2 because these edges are labeled with both views Q1 and Q2, the double-line edges have an edge score of 1, while all other edges have a score of 0. The score of this merged view is therefore $3 * 2^2 + 2 * 1^2 = 14$ – it is the optimal merged view for $\mathbf{Q}^{tweet}$. The score of the merged view in Figure 7.6(bottom) is 4.

Unfortunately, finding an optimal merge is NP-hard.

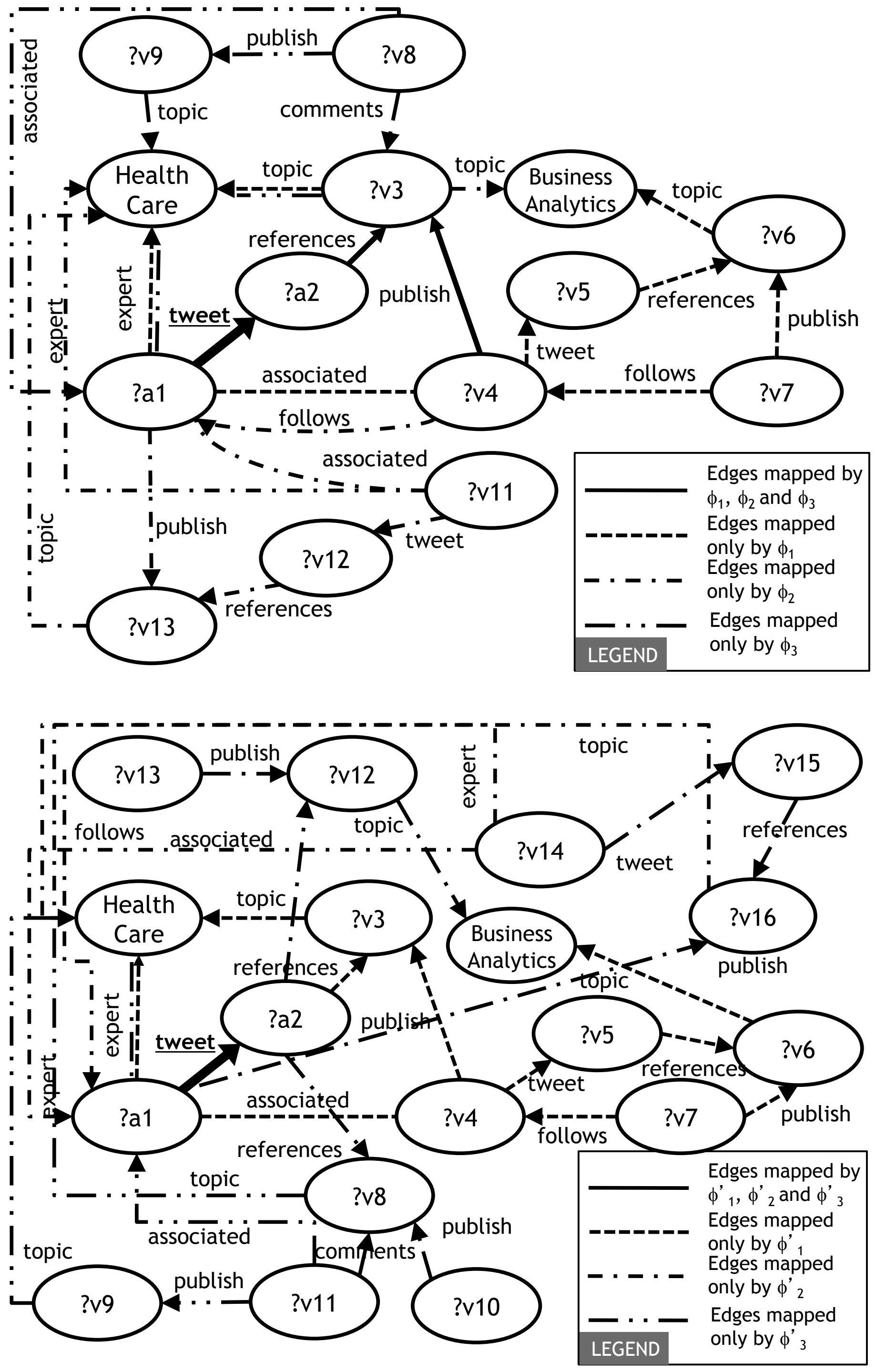


Figure 7.6: Optimal (top) and suboptimal (bottom) merged view of $\{Q1, Q2\}^{tweet}$

Theorem 7.1. *Given a set* $\mathbf{Q}$ *of views and a label* $l \in L$, *computing the optimal merged view of* $\mathbf{Q}^l$ *is NP-hard.*

Theorem 7.1. We start by rephrasing the problem into its decisional version: *Given a set* $\mathbf{Q}$ *of views, a label* $l \in L$, *and a natural number* k, *does there exist a merged* *view M^l of $\mathbf{Q}^l$ with score(M^l) >= k?* We prove NP-hardness of this problem by reduction from SUBGRAPH ISOMORPHISM [54]. Suppose we are given two directed, connected graphs A and B such that A has less than or equal vertices and edges than B (otherwise the problem would be trivial). We translate graphs A and B into views Q_A and Q_B by just assigning the label l to each edge. We then add two additional vertices u, v and an edge (u, l', v), with $l' \in L$, to both views. Finally, for all vertices $x \in V_{Q_A} \setminus \{u, v\}$ we add edges (u, l, x) and (v, l, x) to Q_A and we do the same for Q_B. Now, we can ask the question: *Does there exist a merged view $M^{l'}$ of $\mathbf{Q}^{l'}$ such that score(M) >= $2 * |V_A| + |E_A|$?* The answer to this question is "yes" if and only if A is a isomorphic subgraph of B. To see why this is true, observe that a merged view preserves the properties of a (sub-)isomorphism for those vertices and edges connected to the center edge by virtue of the substitutions ϕ. If A is a subgraph isomorphism, then there must exist a mapping ϕ that maps all edges to B, which contributes the edge score total of $|E_A|$. In addition, all the auxilliary edges map to each other, of which there exist $2 * |V_A|$. If no subgraph isomorphism exist, there is no way to attain this maximum score. □

Fortunately, the AddView search algorithm presented in Figure 7.7 is efficient in practice because most view queries contain relatively few vertices.[5]

Suppose we want to maintain a merged view M^l for every label l in a set $\mathbf{Q}$ of views, so that when a new edge is added to $\mathcal{S}$ with label l, we can just use M^l to determine all view updates. Suppose now we want to add a new view Q to $\mathbf{Q}$. The AddView algorithm updates the set of merged views as follows. First, we create an EA-view AV for each edge e in Q. If no merged view exists e's label, we initialize M^l to Q by adding a trivial annotation function λ and the identity variable substitution id (lines 4–5). If M^l already exists, we merge Q with M^l. To do so, we create a new variable substitution ϕ which aligns the center edge c of M^l with e (line 7) and call OverlayView to extend it to the highest scoring graph isomorphism ϕ^*, mapping a subgraph of Q to a subgraph of M^l centered around c. On lines 9-12, the merged view is updated to include AV and ϕ^* is extended to a full subgraph isomorphism for Q (line 11).[6]

Algorithm OverlayView computes the highest scoring variable substitution ϕ^* by iterating over all possible pairs of vertices in $N_M \times N_Q$, which is the cross product of neighborhoods of any previously aligned vertices. For each pair (v_M, v_Q), we compute the increase in score, $\Delta s(v_M, v_Q)$, that would result from mapping v_Q to v_M (lines 15–19). After computing the score increase for all pairs, we iterate over them in order of decreasing score (line 22), extending the current substitution ϕ for each pair and call OverlayView recursively (line 23) while keeping track of the highest scoring substitution ϕ^* (lines 24–25) which is ultimately returned (line 26). If no pair with positive score increase exists, we return the current substitution ϕ since it cannot be extended to increase its score $s(\phi)$. Iterating over the pairs of vertices in decreasing

[5]In Figure 7.7 we only consider edges with variable end points for simplicity. Edges with constant vertices are a special case and addressed in our implementation. Moreover, we only consider the case of outgoing edges since incoming edges are treated equivalently.

[6]Note, that such extension might require variable renaming which we omit for simplicity.

```
Algorithm AddView
Global: Registered views $\mathbf{Q}$, merged views $\{M^l\}_{l \in L}$
Input: View $Q$

1  for all $e \in E_Q$ s.t. $start(e) \in \mathit{VAR} \wedge end(e) \in \mathit{VAR}$
2      $AV \leftarrow \langle Q, e \rangle$ ,  $l \leftarrow label(e)$
3      if $M^l = \mathbf{null}$ /* Merged view is uninitialized */
4          $\lambda(\tilde{e}) \leftarrow AV$, $\forall \tilde{e} \in E_Q$
5          $M^l \leftarrow \langle \{AV\}, V_Q, E_Q, e, \lambda, \phi_1 = id \rangle$
6      else /* Add new view to merged view */
7          $\phi \leftarrow \{start(e) \mapsto start(\mathsf{c}), end(e) \mapsto end(\mathsf{c})\}$
8          $\phi^* \leftarrow$ OverlayView($M^l = \langle V_M, E_M, \mathsf{c}, \lambda, \Phi \rangle, Q, \phi$)
9          for all $f = (u, l, v) \in E_Q$
10             if $\exists h \in E_M : f\phi^* = h$ then $\lambda(h) \leftarrow \lambda(h) \cup \{AV\}$
11             else $E_M \leftarrow E_M \cup \{f\phi^*\}$, $\lambda(f) \leftarrow \{AV\}$
12         $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{AV\}$, $\Phi \leftarrow \Phi \cup \{\phi^*\}$

Algorithm OverlayView
Input: Merged View $M$, view query $Q$, variable substitution $\phi$
Output: optimal variable substitution $\phi^*$

13 $N_M \leftarrow \left(\bigcup_{v \in \mathsf{range}(\phi)} ngh(v)\right) \setminus \mathsf{range}(\phi)$
14 $N_Q \leftarrow \left(\bigcup_{v \in \mathsf{domain}(\phi)} ngh(v)\right) \setminus \mathsf{domain}(\phi)$
15 for all $(v_M, v_Q) \in N_M \times N_Q$
16     $\Delta s(v_M, v_Q) \leftarrow 0$
17     for all $e_M = (v_M, l, u_M) \in E_M$
18         if $\exists\, e_Q = (v_Q, l, u_Q) \in E_Q$ s.t. $\phi(u_Q) = u_M \vee u_M = u_Q$
19             $\Delta s(v_M, v_Q) \leftarrow \Delta s(v_M, v_Q) + 2|\lambda(e_M)| + 1$
20 if $\exists (v_M, v_Q) \in N_M \times N_Q$ with $\Delta s(v_M, v_Q) > 0$
21     $\phi^* \leftarrow$ null, $s(\phi^*) \leftarrow 0$
22     for all $(v_M, v_Q)$, $\Delta s(v_M, v_Q) > 0$ by decreasing $\Delta s(v_M, v_Q)$
23         $\tilde{\phi} \leftarrow$ OverlayView($M, Q, \phi \cup \{v_Q \mapsto v_M\}$)
24         if $s(\tilde{\phi}) + \Delta s(v_M, v_Q) > s(\phi^*)$
25             $\phi^* \leftarrow \tilde{\phi}$, $s(\phi^*) \leftarrow s(\tilde{\phi}) + \Delta s(v_M, v_Q)$
26     return $\phi^*$
27 else return $\phi$ with $s(\phi) = 0$
```

Figure 7.7: AddView algorithm

order allows us to turn OverlayView into a greedy algorithm by terminating the for-loop in line 22 after a fixed number of iterations. Our experimental results will show that, for scenarios where views are frequently registered with the database or only have a short lifetime, a greedy version of the algorithm is preferable to reduce the cost of adding views.

Proposition 7.2. *The* AddView *algorithm terminates and correctly updates all merged views M^l for some view Q. Its worst-case time complexity is $O(|E_Q|(\max_l |V_{M^l}|)^{|V_Q|})$.*

Sketch. The AddView invokes OverlayView for each edge in Q and therefore covers all edge annotated views associated with Q. OverlayView is essentially an exhaustive search algorithm over all pairs of vertices from M^l and Q. The pruning condition $\Delta s(v_M, v_Q) > 0$ is a valid restriction of the search space, because any vertex pair with $\Delta s(v_M, v_Q) = 0$ cannot possibly contribute positively to the edge score due to the connectedness condition in Definition 7.4 and hence may be safely ignored. Other than that, the search is exhaustive and hence bound to find the highest scoring merge.

The first factor in the worst-case time complexity is due to the fact that OverlayView is invoked $|E_Q|$ times by algorithm AddView. Each iteration of OverlayView considers on the order of $O(|V_{M^l}| \times |V_Q|)$ vertex pairs (for some label l) and the depth of the search tree is at most $|V_Q|$ since we are merging one vertex from Q at a time until none are left or no positive merge is possible. Thus, the total worst-case time complexity is $O(|E_Q|(\max_l |V_{M^l}|)^{|V_Q|})$ (assuming without loss of generality $\max_l |V_{M^l}| \gg |V_Q|$). □

7.4 Multi-View Maintenance

The MultiView algorithm (Figure 7.8) uses a merged view to simultaneously update materialized views affected by the insertion of a single edge.[7]

Lines 1–10 initialize the data structures used, while lines 11–39 finds all answer substitutions that need to be added to the materialized views as a result of e's insertion. For each vertex z in a working copy M of M^l, the algorithm maintains a set of substitution candidates R_z, that is, vertices from the social network database that are potential matches for the query vertex z. For the start and end vertex of the center edge c of M, we initialize the substitution candidates to, respectively, the start and end vertices of e. A constant vertex in M has its substitution candidates initialized to the constant value of the vertex. The remaining variable vertices are uninitialized and their set of substitution candidates is empty.

After initialization, we call the selectEdge function to find all answer substitutions θ for M starting from the empty substitution. The fourth argument to selectEdge is the *active set* A that contains all EA-views for which we are currently finding answers. The active set initially contains all EA-views – but as we extend the substitution θ, EA-views for which θ is no longer a match are removed from the active set. The algorithm proceeds depth first, processing one edge from E_M at a time until all active edges have been processed and it has either found an answer or the substitution θ does not match any views (in which case it backtracks). We say that an edge $e \in E_M$ is *active* if its associated set of EA-views has a non-empty intersection with the active set, i.e. $\lambda(e) \cap A \neq \emptyset$.

[7] In Figure 7.8 we assume edges to be undirected and leave out the special case of loop edges (i.e. edges having the same start and end vertex) to keep the pseudo code concise. Our implementation makes no such assumption.

```
Global variables
$\mathcal{S}$: social network database (index)
$\mathbf{Q}$: Registered views
$\{M^l\}_{l \in L}$: Registered merged views
AS$(Q, \mathcal{S})$: Materialized answer sets for all views in $\mathbf{Q}$

Algorithm MultiView
Input: Added edge $e = (u, l, v)$

1   $M = \langle V_M, E_M \setminus \{\mathsf{c}\}, \mathsf{c}, \lambda, \Phi \rangle \leftarrow M^l$
2   for all $z \in V_M$ /* Initialization of possible matches */
3     if $z = start(\mathsf{c})$ then $R_z \leftarrow \{u\}$
4     else if $z = end(\mathsf{c})$ then $R_z \leftarrow \{v\}$
5     else if $z \in \mathit{VAR}_M$ then $R_z \leftarrow \emptyset$
6     else $R_z \leftarrow \{z\}$
7   selectEdge($M$, $\emptyset$, $\{R_z\}_{z \in V_M}$, $\mathbf{Q}^l$)

Algorithm selectEdge
Input: Merged view $M$, partial substitution $\theta$,
       candidate sets $\{R_z\}$, active EA-views $A$

8   if $\exists e \in E_M$ s.t. $\lambda(e) \cap A \neq \emptyset$
9     $(e, w) \leftarrow \text{argmin}_{(e,z): e \in E_M, z \in \{start(e), end(e)\}, \lambda(e) \cap A \neq \emptyset}\, \text{cost}(e, z)$
10    $A_1 \leftarrow A \cap \lambda(e)$
11    $A_2 \leftarrow A \setminus \lambda(e)$
12    if $A_2 \neq \emptyset$
13      selectEdge($M$, $\theta$, $\{R_z\}$, $A_2$)
14    $x \leftarrow end(e)$ if $w = start(e)$ else $start(e)$
15    if $w \in$ domain($\theta$)
16      $C \leftarrow \{\theta(w)\}$
17    else
18      $C \leftarrow R_w$
19      $R'_w \leftarrow \emptyset$
20    for all $m \in C$ /* substitute possible match */
21      retrieveVertex($\mathcal{S}, m$)
22      if $w \in \mathit{VAR}$ then $\theta' \leftarrow \theta \cup \{w \rightarrow m\}$
23      $M' \leftarrow M$ /* copy query */
24      remove $e$ from $E_{M'}$
25      $\forall z \in V_{M'} : R'_z \leftarrow R_z$ /* copy data structures */
26      $N \leftarrow$ retrieveNeighbors($\mathcal{S}, m, l$)
27      if $R'_x = \emptyset \wedge x \notin$ domain($\theta$)
28        $R'_x \leftarrow N$
29      else
30        $R'_x \leftarrow R'_x \cap N$
31        if $R'_x = \emptyset \wedge \theta(x) \notin N$ then return null /* backtrack */
32      selectEdge($M'$, $\theta'$, $\{R'_z\}$, $A_1$)
33  else
34    collect remaining candidates for $R_z \neq \emptyset$ in substitution $\theta$
35    for all $\langle Q_i, e_i \rangle \in A$ with mapping $\phi_i$ in $M$
36      $\theta_i \leftarrow \emptyset$
37      for all $v \in V_M$ s.t. $v \in$ range($\phi_i$)
38        $\theta_i(\phi_i^{-1}(v)) \leftarrow \theta(v)$
39      AS$(Q_i, \mathcal{S}) \leftarrow$ AS$(Q_i, \mathcal{S}) \cup \{\theta_i\}$
```

Figure 7.8: MultiView algorithm

Suppose we pick the edge-vertex pair (e, w) in line 9, where e is the edge to be processed and w is a vertex of e and the perspective from which we will process e – that is, retrieve all indicent edges that match e. If $\lambda(e) \neq A$ we are not processing all of the remaining active EA-views, so we have to split the query into the active EA-views associated with $\lambda(e)$, A_1, and all other ones in A_2 (lines 10–13). Splitting the query means that we branch our search for view updates which expands the search tree. The cost model we employ therefore favors those edges that do not require query splitting, i.e. $\lambda(e) \supset A \Rightarrow A_2 = \emptyset$.

After the potential query splitting, we process the edge by retrieving all neighbors for all substitution candidates (or actual substution) m of query vertex w that are connected by an edge with the same edge label as e. We copy the query's data structures and update the substitution candidates or substitution of the other vertex of e with the retrieved vertices from the index (lines 20–30). If an update leads to an empty set of substitution candidates or is incompatible with an existing substitution, we backtrack the search. Otherwise, we call the algorithm recursively with the updated data structures and substitution θ' (lines 31–32).

With each invocation, the selectEdge algorithm processes an active edge. When no active edges remain, θ is an answer to all active EA-views in A and we collect these answers (lines 34–38).

Proposition 7.3. *The MultiView algorithm terminates and correctly updates the answer sets of the views in* $\mathbf{Q}$*. Its worst-case time complexity is* $O(2^n)$ *where* n *is the number of variables in the merged view of* $\mathbf{Q}^l$*.*

Sketch. This proof is in large parts analogous to Proof 4.2 with the only difference being the existance of the set of active EA-views A. Hence, we abbreviate the analogous parts here. To proof Proposition 7.3 we first show that the following invariant holds: The substitution candidate sets R_z are supersets for the set of actual answer substitutions $\{\theta'(z) | \theta'(x) = \theta(x) \forall x \in \mathsf{domain}(\theta) \wedge \theta \text{ is an update to some EA-view in } A\}$ for all active EA-views and query vertices z where θ is the current substitution. If R_z =null, then we consider $R_z = V_{\mathcal{S}}$ in this invariant. It is trival to see that this invariant holds initially, because all substitution candidate sets R_z are initialized to null or to the constant vertex of their associated query vertex and $\theta = \emptyset$. Hence, the R_z cover all answer substitutions for the active views. As we process an edge e in the query, the invariant continues to hold because we restrict candidate sets only by the condition that there must exist some edge with the label of the currently processed query edge e between substitution candidates and the current substitution θ. Any substitution θ' that extends θ to an answer substitution must satisfy this condition and hence the invariant holds. However, this only applies for the EA-views which are included in $\lambda(e)$. Therefore, if e does not cover all active views, we have to split the query to ensure that the invariant holds.

Now, since the algorithm searches over all substitution candidates for possible answer substitutions, it must therefore hold that the algorithm will find all answer substitutions for the EA-views. To see why the algorithm produces *only* feasible answer substitutions and terminates, note, that the algorithm processes one edge at a time in a depth-first search manner an never revisists an edge. This means, it must ultimately process all edges unless the current substitution cannot be extended to an answer substitution. If it does process all edges, any remaining substitutions must therefore be an answer substitutions because it was ensured that the required relationships exist between the vertex substitutions. □

Cost Model. Line 9 employs a cost function cost. The cost model we employ primarily aims to reduce the cost of query splitting. Additionally, since processing an edge $e = (u, l, v)$ requires that we either retrieve the l-neighborhood of u or v from the social network database index, the cost model also chooses the vertex that requires less costly neighborhood retrieval. We define cost as follows: $\mathsf{cost}(e, z) = (|A| - |\lambda(e) \cap A|) \times 10^8 + \mathsf{cost}(z) \times \sqrt[\alpha]{\mathsf{cost}(v)}$, where z, v are the vertices of edge e, $\alpha > 1$, and $z \in \mathsf{domain}(\theta)$ or $R_z \neq \emptyset$. If z has not been initialized, then $\mathsf{cost}(e, z) = \infty$. The first term of the summation takes into account the number of active EA-views that are contained in the edge's λ-annotation. We multiply its cost by a large factor of 10^8 to ensure that we are always choosing the edge with the highest EA-view coverage and thus processing the common query subgraphs first. The second term is an estimate of the substitution cost for vertex z with respect to edge e. We define the cost of a vertex u as $\mathsf{cost}(u) = 1$ if $u \in \mathsf{domain}(\theta)$, $|R_u|$ if $R_z \neq \emptyset$, and β otherwise. The constants α and β allow tuning of the model – in our experiments, we chose $\alpha = 5$ and $\beta = 100$.

7.5 Experimental Evaluation

We implemented the MultiView and ViewBasic algorithms in Java, on top of the COSI graph DB presented in Chapter 6. The view merging algorithm AddView was also implemented in Java. All experiments were conducted on machines with 8-core Intel Xeon CPUs clocked at 2.33 GHz and allotted 2GB of RAM. Only the view maintenance experiments for the large datasets were allowed 4GB of RAM.

We used 6 real-world social network datasets in our experimental evaluation. We included one dataset for each size category (small, medium, large) and density (sparse, dense)[8] to cover a wide range of social network databases with up to 540 million edges. All datasets were crawled from real world social networks. Detailed statistics for each dataset are shown in Table 7.5. The datasets were converted to RDF by assigning a random label (i.e. predicate) from a fixed set of labels L to each edge in the network. We withheld a set of 10,000 edges for each dataset to be added during view maintenance (in insertion order) and loaded the rest into the graph database. For the two large datasets we used a Cassandra cluster consisting of 8 machines as the storage backend. The other datasets were persisted in a local BerkeleyDB database.

Next, we automatically generated sets of queries to be maintained as views for each dataset. For each dataset, we generated queries of increasing complexity. Query complexity is measured by the number of edges and vertices in the query. To evaluate the hypothesis of this paper, that exploiting query overlap improves

[8]We consider a network to be dense if the average vertex degree is larger than 30.

Dataset	*Type*	*# Edges*	*# Vertices*	$\|L\|$	*Size (MB)*	*Source*
Physics	small, dense	230,000	11,939	10	4.2	[55]
Enron	small, sparse	360,000	34,070	10	6.5	[83]
Youtube	medium, sparse	5,000,000	1,104,807	17	99.1	[103]
Flickr	medium, dense	30,000,000	1,792,727	17	600	[103]
LiveJournal	large, sparse	180,000,000	11,981,930	17	3,900	[103]
Orkut	large, dense	540,000,000	11,749,292	17	12,000	[103]

Figure 7.9: Datasets used in the experiments

view maintenance performance, we also generated queries with different degrees of overlap. We generated 12,000 queries in total, with 4 to 11 vertices and 4 to 16 edges.

7.5.1 Query Generation Algorithm

Our query generation algorithm is shown in Figure 7.10. To generate a query with $\#e$ edges and $\#v$ vertices, the algorithm is invoked with the empty query and substitution (GenerateQuery$(\emptyset, \emptyset, \#e, \#v)$). After initialization, the algorithm expands the query one edge/vertex at a time by randomly exploring the incident edges of substituted vertices and calls itself recursively until the desired number of edges and vertices has been added. The algorithm maintains a valid answer substitution θ while expanding the generated query Q to ensure that Q has a nonempty answer set – such queries are less likely to be of interest to many users.

During initialization (lines 2–8), Q is initialized to a single variable vertex if the algorithm has been invoked with $Q = \emptyset$ (line 2). Otherwise, the initial query is expanded by $\#e$ edges and $\#v$ vertices. This option is useful to generate queries that share a common "base" query. Next, we find the answer set A to Q on the social network database $\mathcal{S}$. We select a random substitution from A and invoke the algorithm recursively to expand Q for the chosen substitution. We repeat this process up to τ times, where τ is a constant, in case the chosen substitution cannot be expanded by $\#e$ edges and $\#v$ vertices. For our experiments, we set $\tau = 10$.

When GenerateQuery is invoked with an initialized substitution θ (lines 10–30), we start by randomly picking a query vertex x from V_Q. Next, we decide if we should try adding an edge that connects to a new query vertex (lines 14–17) or adding an edge that connects existing query vertices (lines 19–22). Intuitively, if $\#e \approx \#v$ we do the former and if $\#e >> \#v$ we do the latter.

To add a new vertex, we retrieve all edges E incident on $\theta(x)$ whose other vertex is not yet part of the substitution θ by a database index call. We choose the edge e_{xy} randomly from the set E if it is not empty, initialize y to be a new query vertex and extend the substitution θ to map y to the edge's other vertex.

```
Global variables
$\mathcal{S}$: social network database (index)

Algorithm GenerateQuery
Input: Partial query $Q = (V_Q, E_Q)$, substitution $\theta$,
       remaining edges $\#e$, remaining vertices $\#v$

if $\theta = \emptyset$ then /* initialize */
  if $V_Q = \emptyset$ then $Q \leftarrow (\{?v_1\}, \emptyset)$
  $A \leftarrow$ answer($Q$,$\mathcal{S}$)
  $\tau$-times do
    $\theta \leftarrow$ choose randomly from $A$
    $Q' \leftarrow$ GenerateQuery($Q, \theta$)
    if $Q' \neq$ null then return $Q'$
  return "Could not generate query"
else if $\#e > 0$ then
  $\tau$-times do
    $x \leftarrow$ choose randomly from $V_Q$
    $y, e_{xy} \leftarrow$ null
    if $random[0,1) < \frac{\#v}{\#e}$ then
      $E \leftarrow$ retrieveEdges($\mathcal{S}$,$\theta(x)$) $\cap\{e = (u,l,v) \mid \theta(x) = u \wedge v \notin$ range($\theta$)
                        $\vee\, \theta(x) = v \wedge u \notin$ range($\theta$)$\}$
      if $E \neq \emptyset$ then
        $e_{xy} = (u,l,v) \leftarrow$ choose randomly from $E$
        $y \leftarrow ?v_{|V_Q|+1}$
    else
      $E \leftarrow$ retrieveEdges($\mathcal{S}$,$\theta(x)$) $\cap\{e = (u,l,v) \mid \theta(x) = u \wedge v \in$ range($\theta$)
                        $\vee\, \theta(x) = v \wedge u \in$ range($\theta$)$\}$
      if $E \neq \emptyset$ then
        $e_{xy} = (u,l,v) \leftarrow$ choose randomly from $E$
        if $\theta(x) = u$ then $y \leftarrow \theta^{-1}(v)$ else $y \leftarrow \theta^{-1}(u)$
    if $e_{xy} \neq$ null then
      $Q' = (V'_Q, E'_Q) \leftarrow Q$
      if $y \notin V'_Q$ then $V'_Q \leftarrow V'_Q \cup \{y\}$, $\Delta\#v \leftarrow 1$
      if $\theta(x) = u$ then $E'_Q \leftarrow E'_Q \cup \{(x,l,y)\}$, $\theta' \leftarrow \theta \cup \{y \mapsto v\}$
      else $E'_Q \leftarrow E'_Q \cup \{(y,l,x)\}$, $\theta' \leftarrow \theta \cup \{y \mapsto u\}$
      $\tilde{Q} \leftarrow$ GenerateQuery($Q'$,$\theta'$,$\#e - 1$,$\#v - \Delta\#v$)
      if $\tilde{Q} \neq \emptyset$ then return $\tilde{Q}$
  return null /* could not expand query */
else
  while |answer($Q, \mathcal{S}$)| $> \delta$ do
    if $V_Q \cap$ domain($\theta$) $= \emptyset$ then return null
    $x \leftarrow$ choose randomly from $V_Q \cap$ domain($\theta$)
    replace $x$ by $\theta(x)$ in $Q$ /* add constant */
  return $Q$
```

Figure 7.10: Query generation algorithm

To add an edge that connects existing vertices, we retrieve all edges E incident on $\theta(x)$ whose other vertex is in the range of the substitution θ. As before, we choose e_{xy} randomly from E, but this time y is set to be the query vertex that is mapped to

e_{xy}'s other vertex under θ. Hence, in both cases we retrieve appropriate edges from the database and then extend the query by a corresponding query edge to ensure that the substitution θ remains a valid answer substitution. The query extension is executed on lines 24–27. Finally, we invoke the algorithm recursively with the updated query and substitution and reduced edge/vertex numbers. If that call returns a nonempty query, we return it. If we cannot find a suitable incident edge e_{xy} or if the recursive call returns an empty query because subsequent expansions were unsuccessful, we repeat this expansion process up to τ times. If all of these trails fail we return "null".

Finally, when the algorithm is invoked with $\#e = 0$, Q contains the desired number of edges (and vertices). Furthermore, substitution θ is a valid answer substitution for Q by construction. Consequently, Q has a nonempty answer set. However, Q might be under-constrained and thus have an unduely large answer set. Hence, we post-process generated queries (lines 32–36) by replacing variable vertices v in Q by constant vertices, where the constant is set to be the variable vertex substitution $\theta(v)$, for as long as the answer set is larger than some fixed constant δ. In our experiments, we use $\delta = 1000$.

7.5.2 View Merging Results

To evaluate the performance of the merging algorithm AddView we took subsets of 30 queries from our generated query sets. We consecutively registered these 30 queries as views by invoking the AddView algorithm and measured the time taken to merge the view. Figure 7.11 shows the time taken to merge each of the 30 views averaged over sets of queries with 12 edges and 10 vertices each as a representative example. The solid black line represents the AddView algorithm as shown in Figure 7.7. *Top k* denotes a greedy version of the AddView algorithm where we terminate the for-loop on line 22 after k iterations, that is, we only consider the k highest scoring vertex pairs. As could be expected, the time to merge an additional view increases with the number of registered views. It can also be seen that the heuristic versions of the algorithm are much faster. While view merging is only a one time cost incurred at registration time, the optimal algorithm can become prohibitively expensive when more or more complex views are registered.

Figure 7.12(top) displays the total time taken to merge all 30 views averaged across all generated query sets for each version of the algorithm. These results confirm that the heuristic versions are orders of magnitude faster than the optimal algorithm (the vertical axis is in logarithmic scale). In addition, Figure 7.12(bottom) shows the quality of the resulting merged view, measured as a percentage of optimal merged view score, averaged across all generated query sets. It can be observed that the quality of the heuristic algorithms is surprisingly good. For instance, the Top 2 algorithm achieves 99.6% of the optimal score on average, yet it is two orders of magnitude faster. In the following view maintenance experiments, we used the Top 4 view maintenance algorithm to merge registered views.

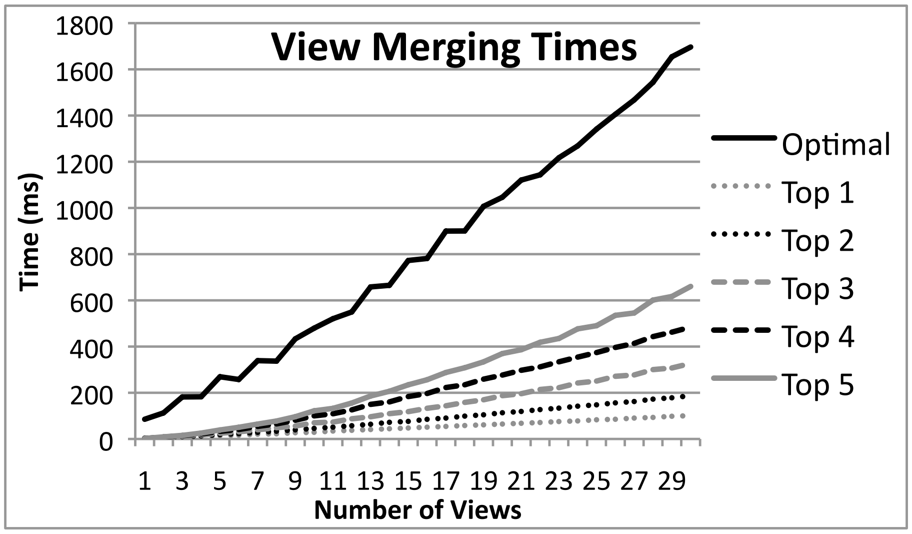


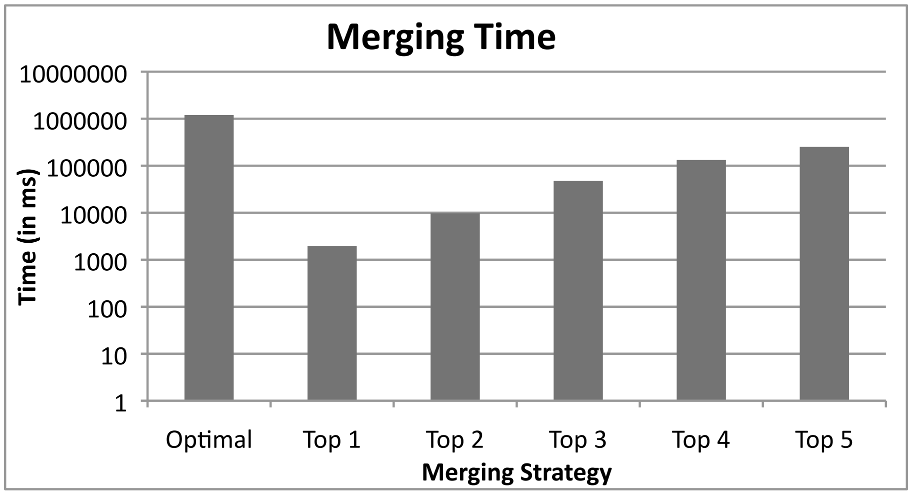
Figure 7.11: Average merging times for views with 12 edges and 6 vertices

7.5.3 View Maintenance Results

To compare the MultiView algorithm against the baseline ViewBasic algorithm, we took each set of generated queries and registered 5, 10, 20, 40 and 80 of these queries as views with the database holding the respective social network dataset in separate experiments. That is, we had one trial for 5 views, one trial for 10 views, etc. for each query set. Each trial consisted of separate runs adding 100 batches of 10 edges each and 30 batches of 100 edges each to the database and updating the views with both the MultiView and the ViewBasic using warm and cold caches. Hence, each trial consisted of 8 runs and we flushed and reloaded the database between runs. We measured the execution time of each algorithm when updating the views for a single batch of edges averaged over all batches.

In total, we conducted 750 individual trials for all 6 datasets. In 94.4% of all trials, MultiView outperformed the baseline, by 477% on average. Figure 7.13 shows those statistics for each of the 6 social network datasets. The grey dot indicates the percentage of trails on which the MultiView algorithm outperformed ViewBasic (right vertical axis). The bar shows the average performance improvement of MultiView comapared against ViewBasic (left vertical axis). We observe that MultiView performed exceptionally well on the Flickr dataset with an average 9-fold improvement on almost 100% of all trails. Interestingly, the MultiView algorithm performed worst on the smallest dataset where it outperformed the baseline only 85% of all trails.

Next, we looked in greater detail at views of medium complexity (as a representative example) by considering those query sets that were generated by extending a base query consisting of 9 edges and 5 vertices. Figure 7.14 shows one chart for each dataset. Each chart has 5 entries on the horizontal axis for each of the 5 query extensions of the base query (the abbreviation “Xe, Yv” stands for “X edges, Y vertices”). For each entry on the horizontal axis, we show 5 bars representing the performance improvement achieved by the MultiView algorithm when maintaining



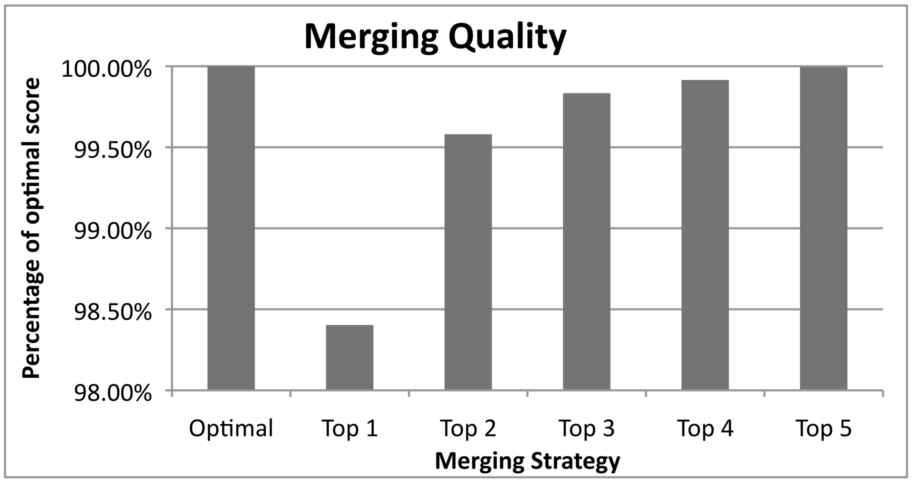


Figure 7.12: Merging time and quality for 30 views averaged over all query sets

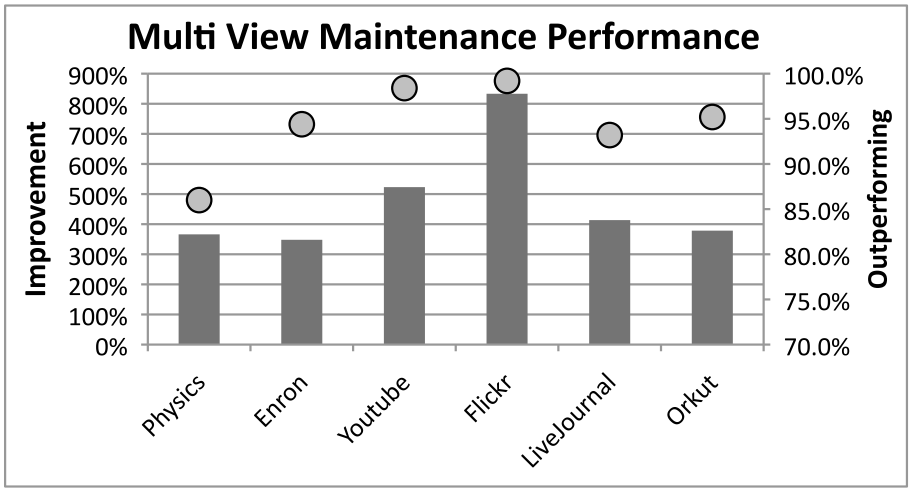


Figure 7.13: Performance of MultiView compared to baseline for each dataset

5, 10, 20, 40, and 80 views respectively (from left to right). The results are averaged over 30 batches of 100 added edges each.

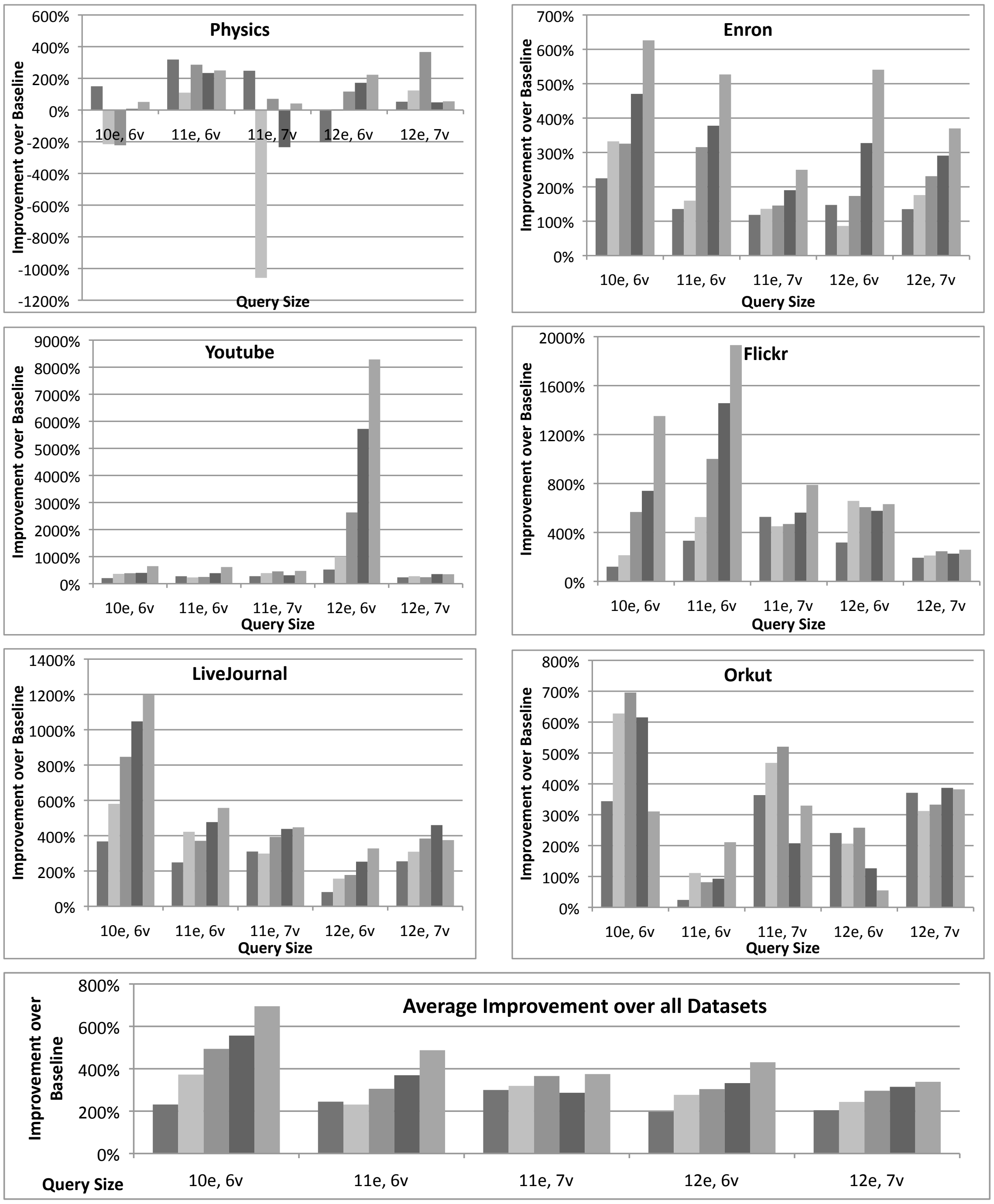


Figure 7.14: MultiView performance improvement compared to ViewBasic for each dataset on sets of medium-sized views. The horizontal axis represents query size where “Xe, Yv” stands for “X edges, Y vertices”. The vertical bars show performance improvements for 5, 10, 20, 40, and 80 registered views (from left to right).

The results vary considerably across datasets and query size. On the Youtube dataset, MultiView was more than 80 times faster than the baseline. In contrast, it performed only marginally better on the Physics dataset and – for some trails – even much worse than the simple ViewBasic algorithm. However, when we average these results across all datasets (also removing the top and bottom outliers) a clearer picture emerges, shown in the bottom most chart of Figure 7.14. The performance improvement increased with the number of views maintained, and decreased with view complexity compared to the base query used to generate the views. These results confirmed our hypothesis: as the number of registered views increases, the number of views that are being simultaneously maintained by MultiView increases as well and so does its performance advantage. Additionally, the smaller the view size, the closer it is to the base query size, hence the larger the relative overlap between views, which means larger common substructures that MultiView can exploit resulting in better performance.

We did not observe proportional performance improvements in the number of registered views. This can be explained by observing that merged views can get considerably larger than each individual view and therefore MultiView has to spend significantly more time on data structure maintenance and manipulation. To illustrate this point, Figure 7.15 shows a fragement of an example merged view for 30 queries where edge thickness indicates the degree of overlap. The merged view contains 177 vertices and 241 edges. Moreover, the cost model we used in this work always prefers high-overlap edges over potentially less costly alternatives. The ViewBasic algorithm, in constrast, directly relies on a more sophisticated cost model for subgraph matching queries [23]. This is also the reason why ViewBasic outperforms MultiView significantly on some trails. Developing more sophisticated cost models for massive merged views will be the subject of future work.

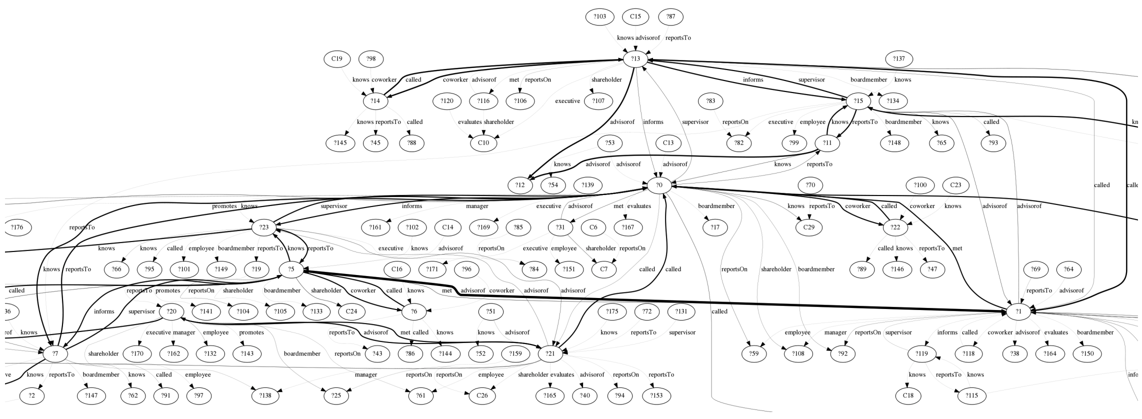


Figure 7.15: Fragment of a merged view for 30 queries consisting of 16 edges and 11 vertices each.

Wrapping up, the experimental results verify that exploiting the common substructures between view queries during view maintenance leads to significant performance improvements.

7.6 Related Work

View maintenance was first studied in the context of relational databases motivated by the need to incrementally materialize query results for faster access [64]. Since then, view maintenance has been applied for different data models (e.g., tree-structured [140, 128, 116, 134, 46, 39, 90, 3]) and to particular classes of views (e.g., aggregate [71] and time-oriented views [97]).

There is less work on view maintenance for very large graph datasets. The work in [168] targets graphs of OEM objects and gives an incremental algorithm whose efficiency on very large graphs is not experimentally validated. Aggregate view maintenance for RDF data is addressed in [71]. Both lines of work introduce an incremental view maintenance algorithm similar to ViewBasic. [152] proposes incremental maintenance of materialized ontologies in the context of rule-enabled Semantic Web, based on the exploitation of the implicit entailments provided by the ontologies. The proposed approach extends a known approach for the incremental maintenance of views in deductive relational databases. The approach in [97] is aimed at maintenance of temporal views through a cache-based mechanism that enables faster access to time-varying moderately-sized datasets.

None of these approaches to view maintenance address the problem of incrementally maintaining multiple views at once. We have shown that exploiting common substructures between views is an appropriate approach for better performance in incremental view maintenance.

Chapter 8
Probabilistic Match Queries in Social Networks

In the preceding chapters, we described index structures and algorithms to answer subgraph matching queries. Subgraph matching queries are *exact*, meaning, any answer to a subgraph matching query must match the query pattern exactly. However, users asking complex queries against massive social networks are often not 100% certain about what they are looking for. For instance, consider the SM-query in Figure 1.8 from Chapter 1. Is the user sure that Peter organized the party? And was that recommended movie really a drama?

In this chapter, we introduce the concept of a "probabilistic subgraph matching" (PS) query over a social network database in which users can specify "approximately" what they are looking for. As before, we assume that the social network database is *certain*.

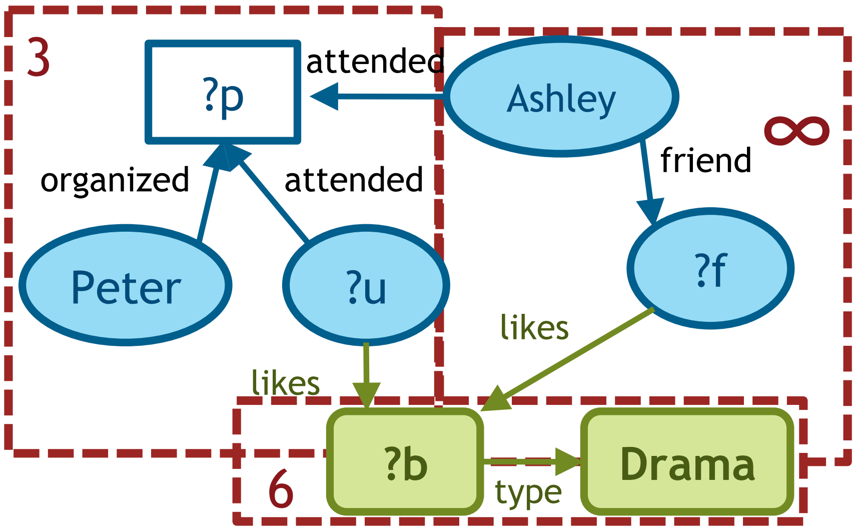


Figure 8.1: Example probabilistic query

There are many situations where such queries could arise. Recall the online social network example in Figure 1.7 in Chapter 1 with the exact SM-query shown in Figure 1.8. We now consider a probabilistic or approximate version of that query. A user (Ashley) is trying to find a book recommended by a friend (in the social network). She believes this book was a drama and that a person she met at Peter's party also liked the book, but is not entirely sure. Figure 8.1 shows this query graphically. The query is divided up into three "boxes" representing the part she is certain about (the fact that the book was recommended by a friend who liked it) and two parts she is not sure about — (is the book a drama? and was it recommended by someone who attended the party organized by Peter?). Each of these query components represents a different box and annotated with a weight, which represents its relative "importance".

There are of course many other applications of probabilistic querying. A security organization with only a vague idea of what it is looking for might want to search for patterns that are inexactly specified. A company hiring new employees might search through a social network for people who satisfy some (or all) of various requirements.

Clearly, a system to answer such queries must need to identify a probability that a given substitution satisfies the query and find all such substitutions efficiently. This chapter proposes a formal syntax and semantics for probabilistic subgraph queries (PS-queries) in Section 8.1 that defines an *answer* to a PS-query. Informally, a PS-query asks for all subgraphs in the graph database that "match" a query pattern (e.g. Figure 8.1) with a probability over some threshold. One important aspect of our approach is that the user can specify his own definition of a match probability in his PS-query, which is not legislated *a priori*. Future work includes the development of an efficient algorithm to answer PS-queries that is provably correct and an experimental evaluation on large social network datasets to demonstrate the efficiency of the algorithm.

8.1 Probabilistic Subgraph (PS) Queries

Throughout this chapter, we assume that the social network database $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is fixed. We now define a probabilistic subgraph query w.r.t. $\mathcal{S}$.

Definition 8.1 (Probabilistic Subgraph Query)**.** A *probabilistic subgraph (PS) query* w.r.t. a social network database $\mathcal{S} = (V, E, start, end, \mathcal{P})$ is an 8-tuple $Q = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q, H_Q, p_Q, \tau_Q)$ where:

- $\tilde{Q} = (V_Q, E_Q, start_Q, end_Q, \mathcal{P}_Q)$ is an SM-query over $\mathcal{S}$ as defined in Chapter 2;
- $H_Q = \langle H_0, H_1, \ldots, H_n \rangle$ is a sequence of "boxes" $H_i \subseteq E_Q$ such that for all $1 \leq i \leq n$, $H_0 \cap H_i = \emptyset$ and $E_Q = \bigcup_{i=0}^{n} H_i$.
- $p_Q : \{0,1\}^{n+1} \to [0,1]$ is a monotonic probability function, i.e. if $x_0 \leq y_0, x_1 \leq y_1, \ldots, x_n \leq y_n$, then $p_Q(\langle x_0, \ldots, x_n \rangle) \leq p_Q(\langle y_0, y_1, \ldots, y_n \rangle)$;
- $\tau_Q \in (0,1]$ is called a *threshold*.

Intuitively, the H_i's represent the "boxes" of Figure 8.1 — in that PS-query, $V_Q = \{Peter, Ashley, Drama, ?p, ?u, ?b, ?f\}$, $E_Q = \{\, e_1 = (Ashley, ?f), e_2 = (?f, ?b),$ $e_3 = (Ashley, ?p), e_4 = (Peter, ?p), e_5 = (?u, ?p), e_6 = (?u, ?b), e_7 = (?b, Drama)\}$, $\mathcal{P}_Q = \{type\}$ with $type = \{(e_1, friend), (e_2, likes), (e_3, attended), (e_4, organized),$ $(e_5, attended), (e_6, likes), (e_7, type)\}$, $H_Q = \{H_0, H_1, H_2\}$ where $H_0 = \{e_1, e_2\}$, $H_1 = \{e_3, e_4, e_5, e_6\}$ and $H_2 = \{e_7\}$. All edges are variables and to simplify notation we use the short-hand notation $e = (u, t, v)$ in the following to denote an edge from u to v labeled with type t.

We will explain the p_Q and τ_Q parts of the query shortly — in order to do so, we first need to define an ordering on substitutions.

Definition 8.2 (Substitution, Ordering, Application)**.** The concept of a substitution defined in Chapter 2 readily applies to PS-queries since they are an extension of SM-queries. If θ_1, θ_2 are both substitutions for query Q, we say $\theta_1 \preceq \theta_2$ iff for all variables in $dom(\theta_2)$, it is the case that $dom(\theta_1) = dom(\theta_2)$. Intuitively, when $\theta \preceq \theta'$, it means that θ' has not instantiated any more variables than θ and agrees with θ on the ones it does.

Suppose $\theta_1 = \{(?b, \textit{Gone with the wind}), (?f, Mark)\}$ and $\theta_2 = \{(?b, \textit{Gone with the wind})\}$. Then it holds that $\theta_1 \preceq \theta_2$.

Definition 8.3 (Pre-answer)**.** A *pre-answer* w.r.t. substitution θ to PS-query Q w.r.t. social network database $\mathcal{S}$ is the maximal subgraph $PA\theta$ of $\mathcal{S}$ such that for every edge $e \in PA$, there is an edge $e' \in E_Q$ such that $e'\theta = e$. We say $PA\theta$ is the *pre-answer* for θ and write PA when θ is unambiguous in the context.

Intuitively, a pre-answer PA is a set of instantiated edges from Q which match the database $\mathcal{S}$.

Example 8.1. There are several possible substitutions and associated pre-answers to the query of Figure 8.1 w.r.t. the database in Figure 1.7. For instance, the substitution $\theta_1 = \{?f = \textit{Jennifer}, b = \textit{Gone with the wind}\}$ has the pre-answer consisting of the edges $PA_1 =$ *(Ashley, friend,Jennifer), (Jennifer, likes, Gone with the wind)* and substitution $\theta_2 = \{?f = \textit{Jennifer}, b = \textit{Titanic}\}$ has pre-answer $PA_2 =$ *(Ashley, friend, Jennifer), (Jennifer, likes, Titanic).*

Now that we have the definition of a pre-answer, we define the probability of a substitution.

Definition 8.4 (Match vector, Probability)**.** Suppose θ is a substitution for PS-query Q w.r.t. graph database $\mathcal{S}$.

The *match vector* for θ w.r.t. Q and $\mathcal{S}$ is an $(n+1)$-bit vector $\langle b_0, \ldots, b_n \rangle$ where $b_i = 1$ iff $H_i\theta$ is a subgraph[1] of PA (0 otherwise). The *probability that* θ *and its pre-answer* PA *matches* Q *w.r.t.* $\mathcal{S}$, denoted $\mathbb{P}(\theta, Q, \mathcal{S})$, is given by $p_Q(\vec{m})$ where $\vec{m}$ is the match vector for θ w.r.t. Q and $\mathcal{S}$.

The match vector tells us which "boxes" of Figure 8.1 exactly match a subgraph of $\mathcal{S}$. This definition of the probability of match explains the monotonicity condition on p_Q: if some "boxes" in the PS-query exactly match PA (after the substitution), and PA' matches all boxes that match PA for a different substitution θ', then θ' is a better answer.

Example 8.2. Suppose we consider the substitutions θ_1 and θ_2 of Example 8.1. Suppose $\theta_3 = \{?f = \textit{Jennifer}, b = \textit{Gone with the wind}, p = \textit{Peter's Bday party}\}$ with pre-answer $PA_3 = \{$*(Ashley, friend, Jennifer), (Ashley, attended, Peter's Bday Party), (Jennifer, likes, Gone with the wind), (Gone with the wind, type, drama), (Jennifer, attended, Peter' Bday Party), (Peter, organized, Peter's Bday Party)*$\}$ The match vector for θ_1 and θ_2 is $\langle 1, 0, 0 \rangle$ and the match vector for $PA_3 = \langle 1, 1, 1 \rangle$.

[1]We are abusing notation a bit here as $H_i\theta$ is a set of labeled edges. The vertices (and labels) of the graph associated with H_i are the vertices (resp. labels) associated with its edges.

In the above example, θ_3 is a complete match for the query in Figure 8.1 since all boxes are satisfied. However, the query of Figure 8.2 (where H_0 represents the edges in the left-most box, H_1 represents the edges in the middle box, and H_2 represents the edges in the rightmost box) has no exact match in our graph DB: there is no movie of type Drama that both an attendee and an organizer of a party attended by Bob have liked. We see that $PA_5 = \{$*(Bob, attended, Spring Break Trip), (Jon, organized, Spring Break Trip), (Alice, attended, Spring Break Trip), (Alice, likes, Toy Story), (Jon, likes, Toy Story)*$\}$ is a pre-answer for a corresponding substitution θ_5 (which does not satisfy the edge requiring that Toy Story is a drama). In this case, the match vector is $\langle 1, 1, 0 \rangle$. Another pre-answer is $PA_6 = \{$*(Bob, attended, Spring Break Trip), (Jon, organized, Spring Break Trip), (Alice, attended, Spring Break Trip), (Jon, likes, Harry Potter)*$\}$.

This pre-answer violates two requirements in the query, namely that the movie be a drama and that Alice likes it. The match vector here is $\langle 1, 0, 0 \rangle$. The set of boxes in the query satisfied by θ_5 is a strict superset of that satisfied by θ_6 — this explains the monotonicity requirement on p_Q in the definition of a PS-query — we do want $\mathbb{P}(\theta_5, Q, \mathcal{S}) \geq \mathbb{P}(\theta_6, Q, \mathcal{S})$.

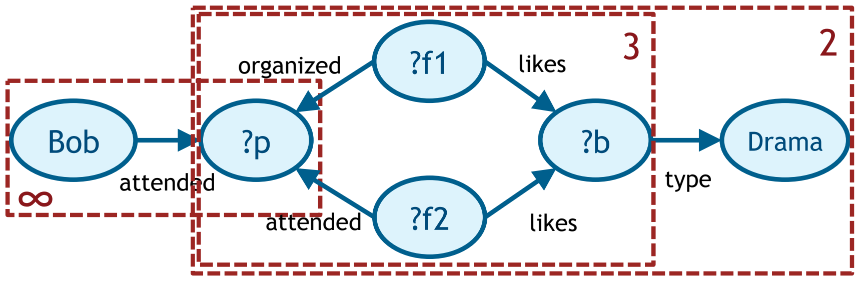


Figure 8.2: Probabilistic query with no exact matches

Definition 8.5 (Answer set)**.** Suppose Q is a PS-query to graph database $\mathcal{S}$. The *answer set* A to Q w.r.t. $\mathcal{S}$, denoted $A(Q, \mathcal{S})$, is the set $\{\theta \mid \mathbb{P}(\theta, Q, \mathcal{S}) \geq \tau_Q\}$.

This definition might include redundancy, as shown in the following example.

Example 8.3. Consider again the query in Figure 8.1. The substitution $\theta_7 = \{?f =$ *Jennifer*, $b =$ *Gone with the wind*, $u =$ *Alice Goodbye*$\}$ and substitution θ_1 both have the same probability $p_Q(\langle 1, 0, 0 \rangle)$. However, θ_7 is redundant given θ_1 because $\theta_7 \preceq \theta_1$.

Now consider the query shown in Figure 8.2 except that the edge *(?b, type, drama)* is deleted (so the third box is deleted from Figure 8.2). As there is no book that two distinct attendees of a party attended by Bob have read, it is clear that there is no way to satisfy the second box. So *any set* $H_0' \supseteq H_0$ *of edges* is a pre-answer to the query. If the query threshold is sufficiently low that H_0 is an answer to the query, then any such H_0' will also be an answer to the query even though these $H_0's$ do not satisfy any boxes over and above H_0.

The next definition eliminates this redundancy.

Definition 8.6 (Compact answer set)**.** Suppose Q is a PS-query to social network database $\mathcal{S}$. The *compact answer set* to Q w.r.t. $\mathcal{S}$, denoted $\mathsf{CAS}(Q,\mathcal{S})$, is a minimal subset of $A(Q,\mathcal{S})$ such that for every substitution $\theta \in ANS(Q,\mathcal{S})$ there is a substitution $\theta' \in CAS(Q,\mathcal{S})$ such that (i) $PA\theta$ is a subgraph of $PA(\theta')$ and $\mathbb{P}(\theta,Q,\mathcal{S}) < \mathbb{P}(\theta',Q,\mathcal{S})$, or (ii) $PA(\theta')$ is a subgraph of $PA\theta$.

For instance, an element of the compact answer set for the query of Figure 8.2 is θ_5. One problem with the definition of a PS-query Q is that p_Q can be exponential in length if one has to specify the value of $p(\langle b_0,\ldots,b_n\rangle)$ for every bit-vector of length $(n+1)$. In order to force users to succinctly specify p_Q, we will assume that p_Q is a generalized linear probability function commonly used in logistic regression analysis.

Definition 8.7 (Generalized linear prob. function)**.** A *generalized linear probability function* p_Q of a probabilistic query Q has the form,
$p_Q(b_0,b_1,\ldots,b_n) =$

$$\begin{cases} \frac{1}{1+e^{-(\alpha_0+\sum_{i=1}^{n}\alpha_i b_i)}} & \text{if } b_0 = 1 \\ 0 & \text{if } b_0 = 0 \end{cases}$$

This definition says that when b_0 is 0, (i.e. H_0 does not exactly match a subgraph of $\mathcal{S}$), then the probability returned is 0 — otherwise, we associate a constant α_i with each H_i and compute the sum s of the $\alpha_i \cdot b_i$'s. We then compute $1+e^{-s}$ — let this quantity be s'. We return $1/s'$ as the probability. For instance, in the case of the query in Figure 8.1, suppose $\alpha_0 = 1, \alpha_1 = 6, \alpha_2 = 3$. Then $p_Q(\langle 1,1,0\rangle)$ is given by $\frac{1}{1+e^{-(1+6)}} = \frac{1}{1+e^{-7}}$. It is easy to see that all generalized linear probability functions satisfy the monotonicity requirement.

8.2 Query Answering

A *naive algorithm* to answer a query Q is to first generate the set all exact subgraph matching queries by considering all sets E of edges such that $H_0 \subseteq E \subseteq E_Q$. One could then compute the answers for each and every one of these, find the associated match vectors and compute those that exceed the threshold τ_Q. This method would be exponential in $|E_Q|$ — as we consider real world social network data in this paper with edge sets of 778M and over 1B, this approach is not feasible.

A *seminaive algorithm* would take advantage of the compactness property and only look at all subsets $\mathcal{H}$ of $\{H_0,\ldots,H_n\}$ such that the corresponding match vector yields a probability above threshold. This would still be exponential in n.

Fig. 8.3 shows our proposed PMATCH algorithm. Before we describe the detailed workings of the algorithm, a few notational definitions are in order:

- For each vertex $z \in V_Q$ where Q is a query, we associate two sets:
  - R_z consists of a set of pairs of the form $\langle x, \overline{H}(x)\rangle$ where $x \in V$ is a vertex in the graph database and $\overline{H}(x)$ is a subset of H_Q specifying the subset of all initialized edge sets for which x is a potential match. For each $x \in V$, there is at most one pair of the form $\langle x, -\rangle$ in R_z. Intuitively, R_z

consists of a set of potential candidates vertices in the graph database that could instantiate a particular vertex in the query graph. R_z is built incrementally by the PMATCH algorithm.

  – S_z stores the identifiers i of the edge sets H_i in the query for which the set R_z has been initialized by the algorithm.

- A special function $h_{select}(z, R_z, S_z, Q)$ looks at a query vertex z, the current value of the sets R_z and S_z (which intuitively show how much processing has been done with R_z) and a query Q and chooses the next query vertex to process.

- Finally, H_Q^B denotes the set of 4-tuples $\{(u, \ell, I, v) \mid I = \{j \mid (u, \ell, v) \in H_j\}\}$. In other words, if (u, ℓ, v) is an edge in a query Q, I is the set of all edge set indices j such that $(u, \ell, v) \in H_j$. For instance, in the query of Fig. 8.2, we can associate the set $I = \{1, 2\}$ with the edge *(?f2,attended,?p)* because this edge lies in both H_1 and H_2.

PMATCH processes and modifies the query Q, the substitution θ, as well as the sets R_z and S_z associated with each vertex during each iteration and calls itself recursively.

During initialization, the algorithm looks at all vertices in the query Q – each vertex z in V_Q that is a variable has R_z and S_z set to $\emptyset$ – setting S_z to the empty set means that the variable z has not been processed by the algorithm yet. Each *constant* vertex z in V_Q has R_z set to $\langle\{z, \{0\}\rangle$ denoting that there is only one potential match for that vertex, namely z itself, and $S_z = \{0\}$ denoting that z has been instantiated. After this, in Step 4, the algorithm checks if there is any chance at all of the current substitution becoming a match that exceeds the specified match probability threshold τ_Q. To do this, we see what match probability is assigned when all those edge sets $H_0, \ldots, H_n$ which are still included in Q are satisfied. If this match probability does not exceed the threshold, then we can prune. This is implemented by checking if $\frac{1}{1+e^{-(\alpha_0+\sum_{H_i \in H_Q} \alpha_i)}} < \tau_Q$. If this is the case, it means that even if all the remaining edge sets match a subgraph of G exactly, we still do not have a match probability exceeding the threshold and the search can be immediately terminated – this is a pruning condition.

In Step 5, we check if there is at least one vertex with $R_z \neq \emptyset$. Note that $R_z = \emptyset$ iff vertex z has not been processed yet or if it has already been substituted and therefore processing of z is finished. Our algorithm requires that queries be anchored,[2] and all constant nodes z are initialized in Step 3 to have $R_z \neq \emptyset$. If there

[2]Specifically, we assume that the given query is *anchored*. In exact subgraph matching, a subgraph query is said to be anchored iff every connected component of the query graph contains a constant vertex, *i.e.*, an element from V. We say that a PS-query Q is anchored iff $\forall I \subseteq \{1, \ldots, n\}$ such that $\wp_Q(1, x_1^I, \ldots, x_n^I) \geq \tau_Q$, where $x_i^I = 1$ if $i \in I$ and 0 otherwise, $H_0 \bigcup_{i \in I} H_i$ is anchored in the standard sense. If a query is not anchored, it cannot be answered by retrieving adjacent vertices alone. In such cases, the graph database needs to support index structures which allow the retrieval of all edges for a given label. The algorithm can be easily extended to accommodate such additional index structures.

```
    **Algorithm** PMATCH
    **Input:** PS-query $Q$, Graph database index $G$, partial substitution $\theta$,
        candidate sets $\{R_z\}$ with associated sets of initialized edge sets $\{S_z\}$
    **Output:** Answer set $A$ containing the canonical substitutions for CAS$(Q, G)$

1   if $\theta = \emptyset$ then do for all $z \in V_Q$ /* query initialization */
2     if $z \in VAR$ then $R_z \leftarrow \emptyset$, $S_z \leftarrow \emptyset$
3     else if $z \in V$ then $R_z \leftarrow \{ \langle z, \{0\} \rangle \}$, $S_z \leftarrow \{0\}$
4   if $\frac{1}{1+e^{-(\alpha_0 + \sum_{H_i \in H_Q} \alpha_i)}} < \tau_Q$ then return "NO"
5   if $\exists z : R_z \neq \emptyset$
6     $w \leftarrow \text{argmax}_{z \text{ s.t. } R_z \neq \emptyset} h_{select}(z, R_z, S_z, Q)$
7     for all $\langle m, \overline{H}(m) \rangle \in R_w$ /* substitute candidates */
8       retrieveVertex$(G, m)$
9       if $w \in VAR$ then $\theta' \leftarrow \theta \cup \{w \to m\}$
10      $Q', \{R'_z\}, \{S'_z\} \leftarrow Q, \{R_z\}, \{S_z\}$ /* copy */
11      delete $R'_w$,$S'_w$
12      remove $H_i$ from $Q'$ for all $i \in S_w \setminus \overline{H}(m)$
13      for all edges $e = (w, l, I, v) \in H^B_{Q'}$ incident on $w$ and some $v \in V'_Q$
14        if $w = v$ /* treat self-loops as a special case */
15          if $m \notin$ retrieveNeighbors$(G, m, l)$
16            if $I = \{0\}$ then return "NO"
17            else remove $H_i$ from $Q'$ for all $i \in I$
18        else
19          for all $i \in I$
20            if $i \notin S_v{}'$ /* initialize candidate set */
21              if $i = 0$ then $R'_v \leftarrow \{\langle x, X \cup \{0\}\rangle \mid x \in$ retrieveNeighbors$(G, m, l)$,
                  $X = \overline{H}(x)$ if $\langle x, \overline{H}(x) \rangle \in R'_v$ else $X = \emptyset\}$
22              else for $x \in$ retrieveNeighbors$(G, m, l)$
23                if $\langle x, \overline{H}(x) \rangle \in R'_v$ then $\overline{H}(x) \leftarrow \overline{H}(x) \cup \{i\}$
24                else if $0 \notin S_v{}'$ then $R'_v \leftarrow R'_v \cup \{ \langle x, \{i\} \rangle \}$
25              $S_v{}' \leftarrow S_v{}' \cup \{i\}$
26            else if $i \in S_v{}'$ /* restrict candidate set */
27              if $i = 0$ then
                  $R'_v \leftarrow \{\langle x, \overline{H}(x) \rangle \mid x \in$ retrieveNeighbors$(G, m, l) \wedge \langle x, \overline{H}(x) \rangle \in R'_v\}$
28              else for $\langle x, \overline{H}(x) \rangle \in R'_v$ s.t. $i \in \overline{H}(x)$
29                if $x \notin$ retrieveNeighbors$(G, m, l)$
30                  $\overline{H}(x) \leftarrow \overline{H}(x) \setminus \{i\}$
31                  if $\overline{H}(x) = \emptyset$ then $R'_v \setminus \{\langle x, \overline{H}(x) \rangle\}$
32          for all $i \in S_v{}'$ s.t. $\nexists \langle x, \overline{H}(x) \rangle \in R'_v$ with $i \in \overline{H}(x)$ /* check for emptiness */
33            if $i = 0$ then return "NO"
34            else remove $H_i$ from $Q'$
35        for all $i \in I$ do remove $(w, l, v)$ from $H_i$ in $Q'$
36      PMATCH$(Q', \tau_Q, G, \theta', \{R'_z\}, \{S'_z\})$
37    if $0 \notin S_w \wedge \exists\, e = (w, l, I, v) \in H^B_Q$ s.t. $I \not\subset S_w$ /* split query if necessary */
38      $Q', \{R'_z\}, \{S'_z\} \leftarrow Q, \{R_z\}, \{S_z\}$ /* copy */
39      $R'_w \leftarrow \emptyset$ , $S_w{}' \leftarrow \emptyset$
40      remove $H_i$ from $Q'$ for all $i \in S_w$
41      PMATCH$(Q', G, \theta, \{R'_z\}, \{S'_z\})$
42  else
43    if $\nexists \theta' \in A$ s.t. $\theta' \preceq \theta$
44      $A \leftarrow A \cup \{\theta\}$
45    return /* done */
```

Figure 8.3: PMATCH algorithm

exists a non-empty set R_z, we use the selection cost-benefit analysis function h_{select} to determine the next vertex w from the query to substitute.

On Line 13, we iterate over all potential matches m with associated edge sets in $\overline{H}(x)$ from the set R_w. The vertex m is retrieved from disk and assigned to w in the extended substitution θ' if w is a variable vertex in the query. Before we process the vertex m any further, we make copies of the query and the data structures R_z, S_z associated with all vertices which will be modified in the following. The set $\overline{H}(x)$ contains the ids i of all edge sets H_i for which the vertex m is a potential match. This also means that it cannot be a match for any additional edge set for which S_w has been initialized, that is, for all edge sets H_i with $i \in S_w \setminus \overline{H}(x)$. Consequently, we remove all those edge sets from the query Q' (removing H_i from the query Q' means removing it from $H_{Q'}$ and adjusting the query accordingly). We now process all incident edges $e = (w, l, I, v)$ on w where $I = \{j \mid (w, l, v) \in H_j\}$. To ease the presentation of the algorithm, we assume edges to be undirected. Our implementation, however, does not make this assumption and treats incoming and outgoing edges separately.

Lines 14-17 address the special case of e being a self-loop. The function $\mathsf{retrieveNeighbors}(G, m, l)$ uses the index structures provided by the graph database G to retrieve all vertices connected to m by an edge labeled with l. If m is not a neighbor of itself in the graph G, then the edge e cannot be matched. If that edge is part of the exact H_0 we can prune the search, otherwise, we remove the edge sets which are associated with e.

Lines 19-34 process general edges where v is distinct from w. We iterate over all edge set indices i from the edge annotation and distinguish two cases: (1) the set of potential matches R'_v for which the adjacent vertex v has not yet been initialized wrt i, *i.e.*, $i \notin {S_v}'$ (Lines 21-25) and (2) it has been (Lines 26-31). In the first case, we use the set of neighboring vertices retrieved by $\mathsf{retrieveNeighbors}(G, m, l)$ to initialize the potential match set R_z and then mark the set as initialized w.r.t. i by adding i to S'_v. The function $\mathsf{retrieveNeighbors}(G, m, l)$ uses the index structures provided by the graph database G to retrieve all vertices connected to m by an edge labeled with l. In the second case, we restrict the set R_z by the retrieved adjacent vertices either by adjusting the edge set indices $\overline{H}(x)$ for all potential matches x contained in R_z or discarding the potential match x entirely. In both cases, $i = 0$ has to be treated differently from the general case $i > 0$ since it refers to H_0 which has to be matched exactly by any answer substitution and therefore all potential matches which do not can be directly discarded.

After processing the edge we check whether the updates to the set R'_v have eliminated all potential matches for some of the edge sets in which case we either remove the edge set from the query Q' if $i > 0$, otherwise we prune the search.

Finally, we remove the edge e from all edge sets H_i in Q' with $i \in I$ to establish that it has been processed (Line 35) before calling the algorithm recursively on the updated substitution θ', query Q', and sets $\{R'_z\}, \{S'_z\}$. Additionally, we have to consider whether the choice of w requires the query to be split (lines 37-41). When the set of potential matches R_w has not yet been initialized for some H_i such that there exists an edge in H_i incident on w, we need to split the query in order to guarantee that we do not inadvertently ignore viable answer substitutions. In other words, if $\exists e = (w, l, I, v) \in H_Q^B$ s.t. $I \not\subset S_w$, we have to split the query unless R_w has been initialized for H_0. $0 \in S_w$ guarantees that we do not ignore answer substitutions because any valid substitution must match the exact part of the query. Splitting the query means that we process a copy of the current query Q with S'_w, R'_w reset to be uninitialized (Line 39) and all edge sets removed for which the set R_w had been initialized (Line 40) before continuing.

Finally, if all vertices and edges have been processed, *i.e.*, all sets R_z are empty, we have a answer substitution θ. If the answer set does not contain a more specific answer substitution, we add θ to the result set A. When the algorithm terminates, the compact answer set $CAS(Q, G)$ has been added to the global variable A (initialized to be the empty set) which is the output.

Note that every iteration of the PMATCH algorithm potentially adds one substitution to the set A of answers in line 44. The following result specifies an important invariant of the algorithm.

Proposition 8.1. *Let Q be a PS-query, G a graph database, θ the current partial substitution, θ' any canonical substitution for a pre-answer $PA \in \mathsf{CAS}(Q, G)$ such that $\theta' \preceq \theta$, and $I = \{j | H_j\theta$ is a subgraph of $G\}$. During execution of PMATCH$(Q, G, \emptyset, \emptyset)$, $\forall z \in domain(\theta') \setminus domain(\theta)$ it is the case that $\theta'(z) \in \{x \mid \langle x, \overline{H}(x)\rangle \in R_z$ and $S_z \cap I \subset \overline{H}(x)\}$, if $S_z \cap I \neq \emptyset$, or $\theta'(z) \in V$ otherwise (in which case R_z is* uninitialized *w.r.t. I).*[3]

This result says that the sets R_z are always supersets, unless uninitialized, of the possible extensions to the current substitution θ. As a consequence, PMATCH's strategy amounts to (i) initializing candidate sets R_z and (ii) iteratively restricting candidate sets by retrieving adjacent vertices from disk until we are only left with those substitutions in that capture an answer. PMATCH iteratively constructs such answer substitutions (eliminating answers that are not compact in Step 43). It is easy to see that the invariant holds when the PMATCH algorithm is called initially since all sets R_z are uninitialized. In subsequent iterations, we update the sets of potential matches to ensure that the invariant continues to hold. Splitting the query is crucial to guaranteeing the invariant for cases where our update strategy itself is not sufficient alone.

The following results ensures that the PMATCH algorithm is correct.

Proposition 8.2. *Given a PS-query Q against a graph database G, PMATCH$(Q, G, \emptyset, \emptyset)$ terminates and correctly computes a compact answer set for Q.*

To see why the proposition is true, note that the invariant ensures that we are considering all potential specialization of the current substitution θ which could be part of the answer set at any iteration of the algorithm. By considering all potential substitutions at each iteration, we therefore ensure that the entire search space of answer substitutions is explored. The termination conditions and edge set removals

[3]Note, that for any z in the domain of definition of θ it must be the case that $\theta'(z) = \theta(z)$ by definition of $\preceq$, so we need not consider this case.

throughout the algorithm ensure that we prune away all non-answer substitutions. Edge set removals together with the verification on Line 43, which checks that no substitution more specialized than the current one has already been added to A, ensure that the result set is compact.

As for exact subgraph matching algorithms, the worst case time complexity of the PMATCH algorithm is exponential in the size of the database G, when G is a clique. However, our experiments show that PMATCH is very efficient in practice.

8.2.1 Query Vertex Selection

The order in which we select the vertices for substitution in line 6 of the PMATCH algorithm greatly impacts its performance, because it determines the branching factor and depth of the search tree. We propose to use a cost-benefit function h_{select} which assigns a value to unprocessed but initialized vertices w based on a cost-benefit analysis of the current status of query processing. In the following, we will discuss how to devise good cost-benefit functions and compare their performance experimentally in Section 8.3.

The cost-benefit function has to put the progress in query answering that can be made by choosing a vertex w in relation to the cost of that choice. Let $progress(w, S_w, Q)$ be some measure of progress made in answering Q when selecting w, and let $cost(w, R_w, S_w, Q)$ denote some measure of cost incurred by the choice. We can then define

$$h_{select}(w, R_w, S_w, Q) = \frac{progress(w, S_w, Q)}{cost(w, R_w, S_w, Q)}.$$

Measuring Progress

At each iteration of PMATCH, all incident edges on the selected vertex w are processed and then removed. The algorithm discontinues the current branch of the search tree when there are no more edges left or the maximum potential probability of the substitution drops below the threshold. Hence, the edges to be processed are a good indicator for the expected progress. However, rather than just counting the number of incident edges, we also need to take the weights of their associated edge sets into account, for two reasons: (i) answers are the highest probability substitutions, so we should weigh edges according to the weight of their edge sets to bias the search, and (ii) processing edges from sets with high weight can lead to the removal of edge sets from the query, which in turn might lead to the search being discontinued earlier because the maximum potential probability drops below the threshold. We therefore propose the progress measure

$$progress(w, S_w, Q) = \sum_{(w,l,I,v)\in H_Q^B} \sum_{i\in(I|S_w)} weight(e = (w, l, v), i)$$

for some weight function $weight$ and $(I|S_w)$ defined as I if $0 \in S_w$ and $I \cap S_w$ otherwise. This definition of a progress measure sums the weights of all adjacent edges as discussed above but with one important modification: it excludes those edges that would cause the query to be split after choosing w. Excluding these edges is justified by the fact that they are still contained in the split query Q'. $(I|S_w)$ is defined according to the splitting condition of Line 37 in algorithm PMATCH.

The following weight functions uses the weight of the associated edge set as the weight of the edge, unless the edge set corresponds to the exact component H_0 in which case it returns a constant γ multiple of the maximum weight.

$$weight_1(e = (w, l, v), i) = \begin{cases} \alpha_i & \text{if } i > 0, \\ \gamma \times \max(\alpha_1, \ldots, \alpha_n) & \text{if } i = 0. \end{cases}$$

We can also refine the above formulation by taking the size of the edge sets into account and evenly distribute their weight across the contained edges:

$$weight_2(e = (w, l, v), i) = \begin{cases} \frac{\alpha_i}{|H_i|} & \text{if } i > 0, \\ \gamma \times \max(\alpha_1, \ldots, \alpha_n) & \text{if } i = 0. \end{cases}$$

We denote the versions of the PMATCH algorithm using the above weight functions by PMATCH-1 and PMATCH-2, respectively.

Measuring Cost

Effective cost functions have been extensively studied for exact subgraph matching. Rather than developing a comprehensive framework of cost functions in the setting of probabilistic subgraph matching from scratch, we show how two previously proposed cost measures can be adapted. The size of set $|R_w|$ equals the branching factor of the search tree after choosing the query vertex w and is, therefore, a good cost indicator. A similar observation with more extensive discussion was first made in the work on the DOGMA subgraph matching system [17]. In addition, we can use selectivity statistics for edge labels to estimate the cost of retrieval from disk, as proposed in [143]. Combining both metrics and adapting them to our setting, we propose the cost measure:

$$cost(w, R_w, S_w, Q) = \sum_{\langle x, \overline{H}(x) \rangle \in R_w} \big(\rho + \sum_{(w,l,I,v) \in H_Q^B} avg(l)\big)$$

where ρ is the fixed vertex lookup cost, and $avg(l)$ is the average number of edges labeled l per vertex.

For each candidate $\langle x, \overline{H}(x) \rangle$ we estimate the cost of retrieval from disk as the sum of the cost of vertex lookup and the sum of edge retrieval costs. The edge retrieval cost is approximately proportional to the number of edges being retrieved from disk, which we estimate through $avg(l)$.[4]

[4]We assume that the statistic $avg(l)$ is made available by the graph database.

8.3 Experimental Evaluation

We implemented PMATCH in 5500 lines of Java code on top of the Neo4j (https://neo4j.org) graph database framework. We used Neo4j for persistence – PMATCH calls the Neo4J API to retrieve vertices and neighborhoods from disk. We maintain the sets R_z as hash tables. In order to reduce memory allocation overhead, we only copy those R_z's that actually get modified during a given PMATCH iteration. As the query is updated and edge sets removed, we maintain an aggregate *potential* probability so we can quickly evaluate the termination condition. Recall that the R_z's are sets of pairs $\langle x, \overline{H}(x)\rangle$ where $\overline{H}(x)$ is a set of query edge sets. As the R_z's can become very large, we use use bit-vectors to store the $\overline{H}(x)$'s. Each position in the vector represents one edge set and the corresponding bit indicates whether that set has been initialized for the vertex in the tuple. While the number of edge sets can potentially be very large, the number of edge sets incident[5] on a single vertex is usually relatively small. By mapping edge sets onto bit vector positions for each vertex independently, we can keep the size of the bit vector small.

8.3.1 Setup

We compared PMATCH against two baselines, both of which are implementations of the semi-naive algorithm that computes all subsets $H'_Q \subseteq H_Q$ such that any substitution θ with $(\bigcup_{H \in H'_Q} H)\theta$ being a subgraph of G is assigned a probability above threshold, *i.e.*, $\mathbb{P}(\theta, Q, G) \geq \tau_Q$. For each subset, it merges the edge sets in H'_Q and uses an exact subgraph matching algorithm to find all answers. Both implementations of the semi-naive algorithm are built on top of the Neo4j database library. The first implementation, which we denote SN-1, uses the exact subgraph matching component provided with Neo4j, while the second implementation, denoted SN-2, runs the faster DOGMA [17] algorithm on top of the Neo4j library.

We evaluated PMATCH on two large social media datasets. The first dataset is a crawl of four social networking sites – Orkut, Flickr, Youtube, and LiveJournal – containing approximately 778 million edges and 28 million vertices [104]. The edges denote relationships between individuals and between individuals and groups. In the original dataset, the edges were unlabeled, so we randomly assigned one of four labels to each edge. The second dataset was extracted from the delicious social bookmarking service capturing user posts, various attributes and tag assignments [156]. The dataset consists of approximately 1122 million edges. Both datasets were batch loaded into the Neo4j database using a customized loading framework, with an average loading rate of approximately 23000 edges per second.

The query evaluation benchmark consisted of 18 PS-queries – 9 for each dataset – each having four to six edge sets; the total number of edges in the queries varied from 6 to 17. We designed queries by first fixing an initial structure and designating *constant* vertices. We then initialized those query vertices with vertex identifiers retrieved randomly from the respective dataset. The α values defining the weights

[5] We say an edge set H_i is incident on a vertex v iff H_i contains an edge which is incident on v.

and the probability threshold were set randomly so that the values differ among the queries. During this process, we excluded queries with an empty result set. As vertices were randomly selected, the selectivity of the queries varies with the degree and connectivity characteristics of the randomly chosen vertices. Hence, it is not the complexity of the query (as measured by the number of edge sets or total number of edges) that is expected to primarily determine the query evaluation time, but rather the constant vertices.

All experiments were executed on the same machine powered by a six-core AMD Opteron processor, a 15K RPM 300 GB SAS hard drive and 256 GB of RAM. However, we only allocated a maximum of 2GB of RAM to the Java virtual machine for all experimental trails. We ran the queries with cold and warm caches. For the cold cache mode, we started the database, used a dummy query to initialize it and then ran the benchmark. To warm the caches, we ran the benchmark once and then immediately ran it again in the same order (we report the second query evaluation time). We also tried to run each query twice in a row; however, we obtained very small differences with respect to the warm cache mode. All reported times are in milliseconds and averaged over 5 independent runs.

8.3.2 Results

The query times for the delicious and social networks datasets are reported in Figs. 8.4 and 8.5, respectively. The queries in the delicious benchmark are labeled TA through TI, those for the social networks dataset are labeled SA through SI. Queries are labeled in order of expected query evaluation time. For each query, we also indicated the number of edge sets B and total number of edges E. For instance, query TA has 5 edge sets and 13 edges in total. Both charts compare the query times for PMATCH algorithm equipped with functions $weight_1$ and $weight_2$ of Section 8.2.1 (denoted PMATCH-1 and PMATCH-2, respectively) with the baseline SN-2. For each algorithm and query, the chart reports the query evaluation time with cold (left bars) and warm (right bars) caches.

For both datasets, we observe that PMATCH consistently and significantly outperforms the SN-2 baseline by a factor of 5.2 and 7.6 averaged over all queries for the social networks and delicious datasets respectively. For some queries, our proposed algorithm is up to ten times faster. As expected, this difference is smaller for cold caches (factor of 3.1 and 4.9 respectively).

Fig. 8.6 compares PMATCH-1 with PMATCH-2 and both semi-naive baselines on the delicious dataset. We report the ratio between the query evaluation time of the respective algorithm and the evaluation time of PMATCH-1. Fig. 8.7 provides the same data for the social networks dataset, except the SN-1 baseline which did not converge on any of the queries in the social networks benchmark within one hour or ran out of memory.

On most queries, the differences between PMATCH-1 and PMATCH-2 are small, and averages over all queries do not appear significant. On query TD, PMATCH-1 is consistently faster by a factor of 3. Ignoring this query, it also seems that PMATCH-2 performs better on cold caches for most of the queries in the deli-

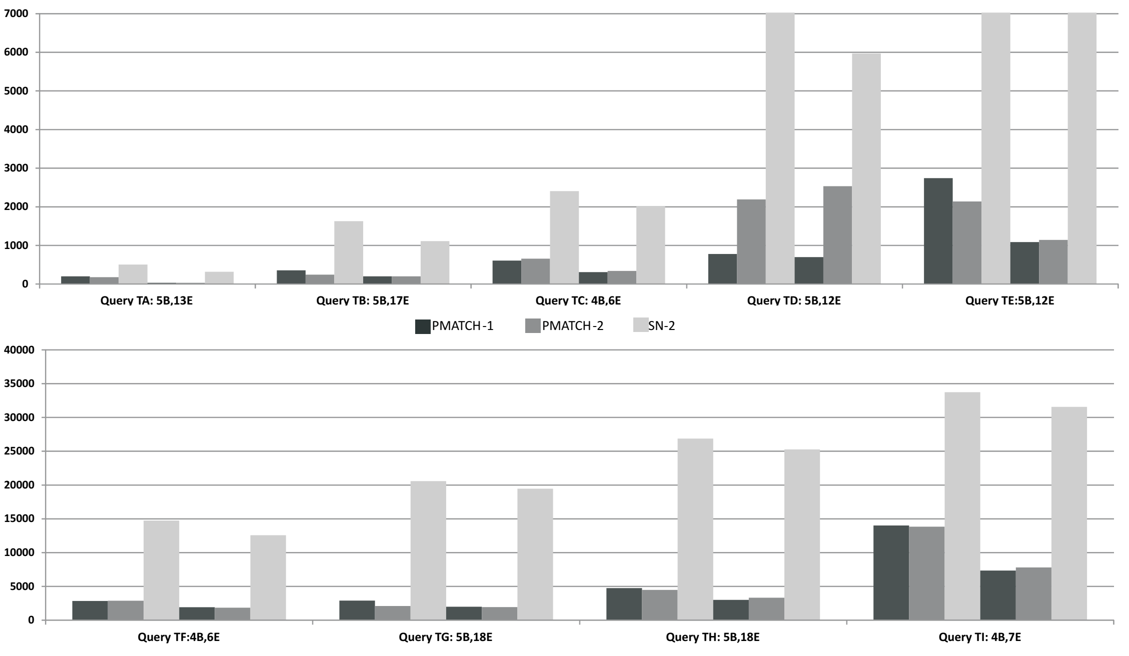


Figure 8.4: Query execution times (in milliseconds) on the delicious dataset of 1.122B edges. B denotes number of edge sets, E denotes number of edges (*e.g.*, the leftmost bar labeled 5B,13E represents 5 edge sets and 13 edges). For each query, the left (resp. right) three bars represent performance with cold (resp. warm) caches

cious benchmark. In general, our results show that the effectiveness of the weight function depends on the query, and no function is significantly better than the other. The tables also show that PMATCH outperforms the SN-2 baseline by a factor of 3 to 7.5 on average. As expected, this difference is larger for warm caches. The SN-1 baseline is up to 3000 times slower and about 500 to 600 times slower on average. Therefore, it is not included in Figures 8.4 and 8.5.

8.4 Related Work

Nilsson [114] proposed a basic approximate subgraph matching algorithm that is still widely used today. Nillson's algorithm is based on A^* search in which each node in the search tree represents a vertex pair matching (a special *null* vertex is used to denote that a vertex is left out from the match). Each vertex pair matching has an associated cost that is inversely proportional to the goodness of the match between the vertices. As in A^* the algorithm then expands the search tree nodes in the order of their cost. Once a path in the search tree corresponds to a full matching it is returned.

This basic notion of *cost* or *edit distance* [133] is used in many approximate subgraph matching systems proposed since Nillson's early work. Authors have proposed ad-hoc measures of distance between two graphs that arguably reflect the

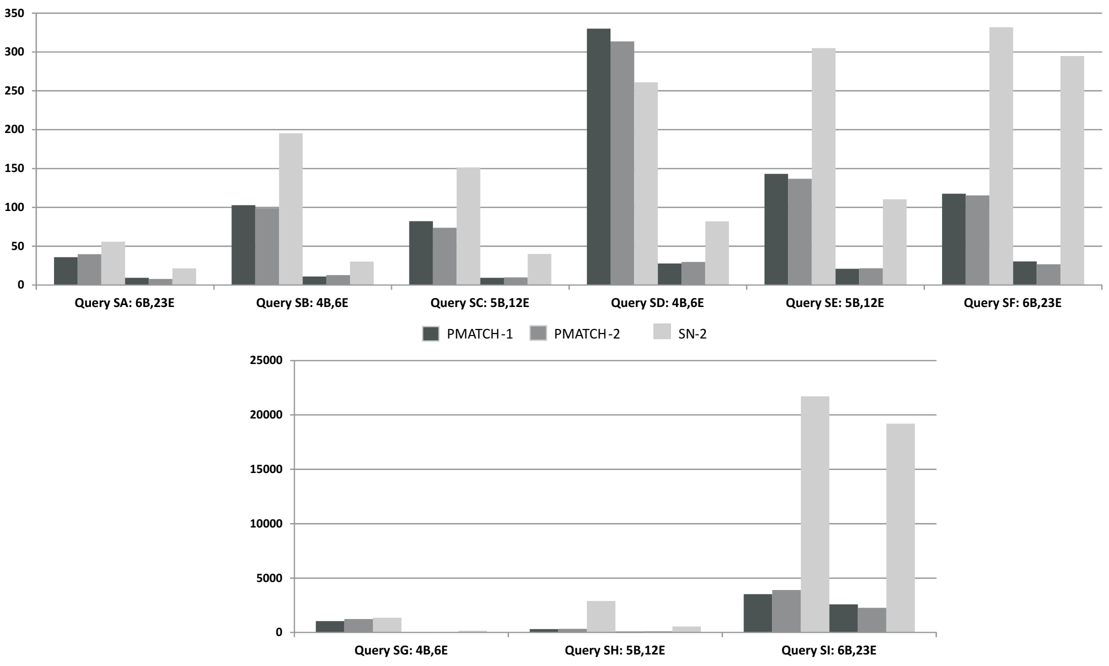


Figure 8.5: Query execution times (in milliseconds) on the social networks dataset of 778M edges. B denotes number of edge sets, E denotes number of edges (*e.g.*, the leftmost bar labeled 6B,23E represents 6 edge sets and 23 edges). For each query, the left (resp. right) three bars represent performance with cold (resp. warm) caches

similarity between them. Grafil [162] tries to identify frequently occurring but still informative (i.e. selective) subgraphs in D, which the authors call *features*. Given a set of features F, they construct a feature matrix with the features as rows and the graphs in D as columns. A cell in the feature matrix counts the number of times a feature occurs in a graph. Features used by Grafil might be the frequent subgraphs identified by gIndex [161]. Given a query graph Q, all features in Q are determined and looked up in the features matrix. For each graph, this allows the computation of a feature similarity score (based on the number of missing features) and return approximate matches in the order of their similarity score.

SAGA [149] measures the distance between two graphs as a weighted linear combination of the structural difference, vertex mismatches, and vertex gaps. Figure 8.8 shows an example graph G1 from D and a query G2 that is matched against G1. The bottom vertex L4 in G2 does not have any correspondence in G1 and is what the authors call a *vertex gap*. Structural differences account for different path length between vertices and vertex mismatches refer to differences in vertex labels. The authors propose functions to measure each of these individual distances. The authors propose an index to efficiently match queries against D in an approximate fashion. The index contains sequences up to a maximum length l of vertex labels such that each of the vertices in the sequences is at most a certain maximum distance

	Cold caches			*Warm caches*		
Query	PMATCH-2	SN-2	SN-1	PMATCH-2	SN-2	SN-1
TA	0.88	2.51	151.28	1.02	9.79	623.37
TB	0.68	4.56	n/a	1.00	5.58	n/a
TC	1.08	3.95	43.91	1.09	6.47	46.47
TD	2.81	9.26	2818.15	3.63	8.55	2931.21
TE	0.78	3.87	6.45	1.05	8.82	9.84
TF	1.01	5.18	8.77	0.96	6.50	0.90
TG	0.73	7.08	n/a	0.97	9.77	n/a
TH	0.94	5.64	n/a	1.11	8.41	n/a
TI	0.99	2.41	81.13	1.06	4.29	61.56
Average	1.10	4.94	518.28	1.32	7.57	612.23

Figure 8.6: Comparison to PMATCH-1 on the delicious dataset

k apart (for the experiments, the authors choose $k = 3$). Note that these sequences generalize the notion of paths in that vertices must no longer be adjacent. As in previous work, the authors use this index to select the set of graphs from D that lie within a certain maximum distance from the query graph Q.

The indexing used in SAGA only works for small graphs as enumerating all possible sequences gets prohibitively expensive, even for moderate l and k. In their later work, the authors propose TALE [150] which allows for more scalable approximate subgraph matching on larger graphs. The idea behind TALE is to create an index based on "interesting" vertices, i.e. those vertices which have selective properties and occur frequently in subgraphs. TALE characterizes a vertex by its label, the list of its neighbors, and the interconnections between the neighbor vertices. The idea is to determine the subset V of interesting vertices in the query graph Q and then to look up all graphs in D which contain the vertices in V. The authors propose a sophisticated index structure to make these lookups efficient. The index-label[6] of each vertex is given by the tuple (label,degree,neighborhood) where the neighborhood is a bitmap index over all neighbor vertices's labels and connections.

Although the authors explicitly design TALE for larger graphs, their understanding of *large* is quite different from ours. They consider graphs with thousands of vertices to be large. Unfortunately, this is inconsistent with reality - real world social networks such as Facebook and Twitter can include billions of vertices. In contrast, we would like to scale probabilistic subgraph matching to millions of vertices. Both TALE and SAGA allow the user to modify the distance measures by configuring weight parameters and label distributions. Nevertheless, the overall similarity measure is system specific and idiosyncratic.

The approaches for approximate subgraph matching reviewed above address the problem of matching a query approximately against a database of many (small)

[6]Here is meaning of "label" is that of GiST.

	Cold caches		*Warm caches*	
Query	PMATCH-2	SN-2	PMATCH-2	SN-2
SA	1.1	1.6	0.8	2.3
SB	1.0	1.9	1.2	2.7
SC	0.9	1.8	1.0	4.3
SD	1.0	0.8	1.1	2.9
SE	1.0	2.1	1.0	5.3
SF	1.0	2.8	0.9	9.7
SG	1.2	1.3	0.9	6.2
SH	1.1	9.4	1.2	6.0
SI	1.1	6.2	0.9	7.4
Average	1.03	3.10	1.00	5.19

Figure 8.7: Comparison to PMATCH-1 on the social networks dataset

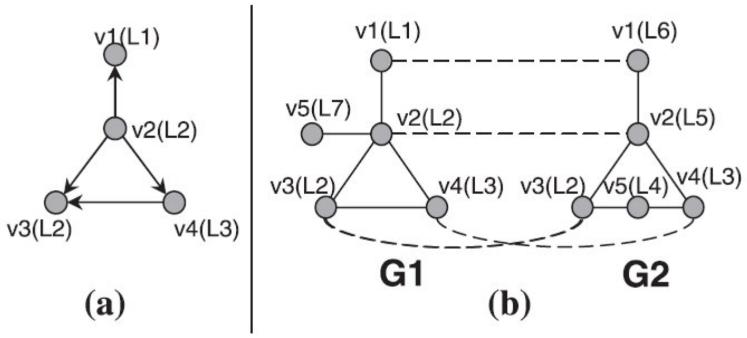


Figure 8.8: Subgraph query matching in SAGA [149]

graphs. The measures proposed for determining the degree of similarity between graphs are ad-hoc and guided by the authors' intuition about the problem domain (such as the importance of vertex labels in biological networks) rather than a sound syntax and semantics. In using any of these systems, the user has to rely on the particular distance measure and is only able to tweak certain system parameters. Furthermore, it is difficult to assess what the expressiveness of existing approximate subgraph matching systems is as they lack a syntax and semantics. Grafil, for instance, cannot handle vertex gaps. In contrast, we allow the author to express what he means by an inexact match in the query, allowing the user to control the semantics of probabilistic query matching.

Another line of work [106], [158] proposes a probabilistic model for the alignment between a query graph Q and a host graph H. The focus of this work is to describe a probability distribution over the set of all possible alignments using features from either graph (such as labels or structure) and then to find the most likely assignment, or MAP-state.
The authors show how edit distances can be modeled using their approach and pro-

vide a rigorous formalism based on probability theory. The usage of a theoretically sound framework to describe subgraph matches is also the focus of our work and can draw inspirations from the work of Hancock et. al. It should be pointed out that Hancock et al. do not provide a query language that could be exposed to the user (in fact their formalism is highly complicated) nor any index methods to speed up finding the MAP-assignment, which makes their work not applicable to large graphs. Furthermore, we believe that modeling a probability distribution over assignments is not the right approach to the problem of probabilistic subgraph querying over large graph databases. The uncertainty of the problem arises from uncertainties about the actual query to be asked rather than how this query matches against the host graph. This is why our approach tries to assign probabilities to the latter which leads to a different problem than the one considered by the authors.

All of the approaches reviewed above have two major shortcomings:

1. They have not demonstrated scalability to social networks with billions of edges. In fact, most prior work focuses on approximate subgraph matching for many small graphs. We, in contrast, are interested in probabilistic subgraph matching against one massive social network.

2. They do not present an expressive but simple query language through which a user can define probabilistic subgraph matching queries that express what they are looking for. Most prior work on approximate subgraph matching, like [149], [150] and others, makes strong assumptions about what constitutes an "approximate" match and thus forcing the user to adopt that assumption. Other work, [106], [158], is mostly theoretical and very general, making it difficult for the user to apply in real world scenarios.

8.5 Query Examples

To give an idea of the benchmark queries we used for our experiments, we present one small query out of the nine queries from each benchmark.

Listing 8.1: Query SB with $\tau_Q = 0.88$, $\alpha_1 = \alpha_2 = \alpha_3 = 2$.

```
Query SB with $\tau_Q=0.88$, $\alpha_1=\alpha_2=\alpha_3 = 2$.
H0
(?A  http://dog.ma/family  http://dogma.umd.edu/orkut/user71852)

H1
(?A  http://dog.ma/coworker  ?B)
(?B  http://dog.ma/boss  http://dogma.umd.edu/orkut/user82456)

H2
(?A  http://dog.ma/family  ?D)
(?D  http://dog.ma/leader  http://dogma.umd.edu/orkut/group13785)

H3
(?A  http://dog.ma/member  ?X)
```

Listing 8.2: Query TC with $\tau_Q = 0.95$, $\alpha_1 = 5$, $\alpha_2 = \alpha_3 = 3$ – the domain name *DN* is *tagora.ecs.soton.ac.uk*.

```
H0
(?T  http://DN/schemas/tagging#hasDomainTag  http://DN/anonymous/
    delicious/tag/sos)
(?P  http://DN/schemas/tagging#tagAssigned  ?T)

H1
(?P  http://DN/schemas/tagging#taggedResource  http://DN/anonymous/
    resource/delicious/698826)

H2
(?U  http://DN/schemas/tagging#hasDomainTag  http://DN/anonymous/
    delicious/tag/debugging)
(?P  http://DN/schemas/tagging#tagAssigned  ?U)

H3
(?P  http://DN/schemas/tagging#taggedOn  $D:2006-09-13T19:08:47Z)
```

Part II

Probabilistic Reasoning on Social Networks

Chapter 9
Probabilistic Soft Logic
9.1 Introduction

In the previous chapters, we have developed several algorithms and index structures for querying social network data and retrieving existing information. In this chapter we introduce a novel formalism to learn from and reason about noisy, or uncertain, multi-relational data, such as social networks. Our goal is to infer information that is not present in the given data based on models that we can learn from other, complete datasets.

For example, consider a Wikipedia-like environment in which a set of hyperlinked documents are being edited by a set of interacting users as shown in Figure 9.1. In this example. Bob has edited the document "Algebra" which links to the document "Relativity". Given such a social network about users and documents, we may be interested in identifying similar documents in order to provide user recommendations. Since the dataset does not indicate which documents may be similar we have to infer document similarity from the available data. Inference, in this context, refers to "probabilistic" inference since there is no way for us to deterministically deduce this information. In other words, we have to predict similarity. In addition to text analysis, we can exploit the network structure to improve probabilistic inference. For instance, it is likely that documents link to similar documents, or that users edit similar documents. Infering a document's category for classification purposes is another motivating example. Some documents are assigned a category, while others are not. We can infer the categories of those documents by reasoning about the similarity to documents that do have a category.

Each of these problems can be approached in isolation by comparing pairs of entities based on their attributes alone. However, due to the relational structure of this problem, document category and document similarity are closely entangled and we can improve inference performance by exploiting the network and reasoning about these properties collectively. In this chapter we present probabilistic soft logic (PSL), a general-purpose framework for expressing, reasoning about, and learning structural dependencies. PSL provides a declarative, logic-based language tailored to relational domains that require reasoning about similarity and/or probability. In PSL, a probabilistic inference problem is described by a set of rules. For instance, the simple rule $link(D_1, D_2) \Rightarrow D_1 \cong D_2$ captures that two documents, D_1 and D_2, are considered similar, if there is a link between them. The similarity is represented by values in $[0, 1]$ (0 being most dissimilar and 1 being most similar) and we denote the similarity predicate by $\cong$. PSL's set constructs allow for significant modeling flexibility in this domain. For example, let $\{D.editedBy\}$ represent the set of all users who edited D, then, one can write $\{D_1.edited\} \cong \{D_2.edited\} \Rightarrow D_1 \cong D_2$ to state that if the sets of users who edited D_1 and D_2 are similar we conclude that D_1 is similar to D_2.

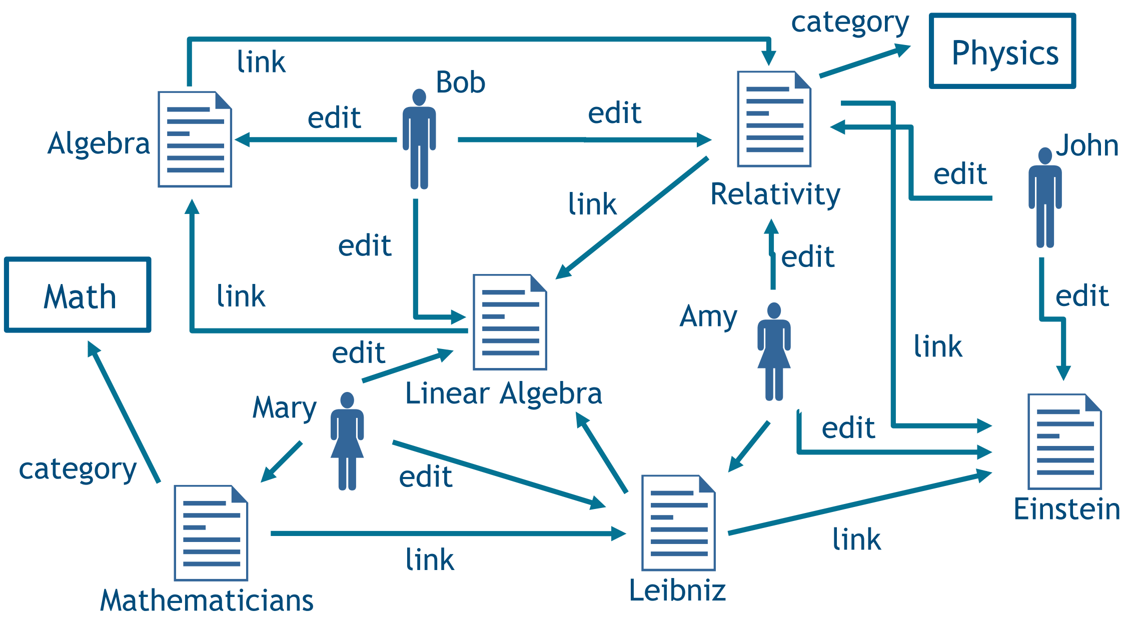


Figure 9.1: Wikipedia Social Network

A variety of artificial intelligence applications require the ability to learn from and reason about noisy, or uncertain, multi-relational data, such as the Wikipedia example discussed above. This has motivated the fields of statistical relational learning (SRL) and multi-relational data mining, which have made significant progress on developing unified frameworks that provide 1) a language for expressing structural regularities in a domain, and 2) principled support for probabilistic inference (see e.g. [57, 37]). In addition, many applications of interest also involve a third aspect: the need to reason about soft truth values such as similarities or probabilities, which has not been directly supported in existing SRL frameworks.

PSL is similar to existing SRL models, e.g., RMNs [147], BLPs [78], MLNs [125], in that it defines a probabilistic graphical model over the properties and relations of the entities in a domain as a grounding of a set of rules that have attached parameters. In contrast to existing SRL models, PSL embodies the following novel characteristics. First, PSL provides a unified framework in which probabilistic reasoning about relational structure is seamlessly incorporated with reasoning about continuous random variables. A direct consequence of this is that PSL can incorporate any existing similarity or probability measures, thereby extending their applicability to a relational context. This is particularly useful for probabilistic reasoning in domains with semi-structured data, such as computer vision, natural language processing, personalized medicine, information integration and others, where PSL can integrate existing analysis techniques for unstructured data into its structural inference engine. Second, as an extension to its treatment of continuous random variables, PSL supports reasoning about *sets* of entities and *aggregates*, defined by a given relation. Such aggregates are treated as first order citizens in PSL and enrich its modeling capabilities. Third, PSL features provably efficient and practically fast inference algorithms.

There are several applications that could benefit from the ability to incorporate similarities into a relational framework. For example, in computer vision, the similarity of two images can be based both on domain-specific similarity measures and on relational structure within the image; in bioinformatics, one may predict the function of a protein based on its similarity to other proteins, inferred from its properties, and protein-protein interactions; and in information integration tasks one can determine the similarity between entities based on entity attributes as well as the relations between entities.

PSL combines aspects from many distinct fields of computer science into a coherent framework. The declarative syntax and semantics of PSL, which allows users to easily define and understand complex, joint probabilistic models, are inspired by previous work in artificial intelligence and logic programming (e.g. [79, 113]). The probabilistic model underlying PSL is an extension of Markov random fields frequently used in machine learning and physics [80]. Inferring the most likely outcome of a given PSL model (called *MAP inference*) is cast as a convex optimization problem and the inference engine of PSL relies on efficient interior point methods developed in the numerical optimization community during the last decade to quickly determine its solution [91]. Inferring the probability distribution over truth values (called *marginal inference*) is related to volume computation in high dimensional polytopes, so PSL adapts sampling algorithms from computational geometry to compute marginal distributions [92]. The initial data and inferred values are stored in a relational database backend so that PSL can take advantage of efficient querying techniques developed in the database community [53]. Inspired by best practices in software engineering, the architecture of PSL is event driven and highly modular which leads to efficient probabilistic model construction and allows the integration of external similarity and probability measures.

A more detailed review of prior work related to PSL can be found in Section 9.9. In the next section, we will introduce the formal syntax and semantics of PSL, before we discuss inference and learning.

9.2 Syntax and Semantics

9.2.1 Syntax

PSL specifies how similarities and probabilities propagate through the relational structure using annotated rules. PSL represents a family of languages, defined by particular user choices. In the following general description, we discuss the choice points and explain the specific choices made in this chapter.

PSL rules are the basic building blocks of PSL programs. PSL rules follow the syntax of generalized annotated logic programs (GAPs) [79] for the special case of real-valued interpretations. Hence, PSL rules are similar to those in Prolog, but with continuous truth values, that is, interpretations assign real values – typically in $[0, 1]$ – to atoms, clauses, and rules. In contrast to GAPs, PSL rules are annotated with weights which denote the relative probability or strength of a rule. We will

$w_1 : A.text \overset{s_n}{=} B.text \tilde{\Rightarrow} A \cong B$
$w_2 : \{A.editor\} \overset{s_{\{\}}}{=} \{B.editor\} \tilde{\Rightarrow} A \cong B$
$w_3 : \{A.linksTo\} \cup \{A.linksTo.linksTo\}$ $\overset{s_{\{\}}}{=} \{B.linksTo\} \cup \{B.linksTo.linksTo\} \tilde{\Rightarrow} A \cong B$
Hard : $A \cong B \tilde{\wedge} B \cong C \tilde{\Rightarrow} A \cong C$

Table 9.1: Example PSL program for the Wikipedia domain.

formalize rule weights and how they factor into the probabilistic model after we discuss the syntax and semantics of the rule itself.

PSL rules can be written in first-order logic (FOL) syntax using predicates, logical connectives, constants and variables; in addition, we support an object-oriented (OO) short-hand that is more succinct in many cases. In particular, let X and Y be variables representing entities in the domain. Entities are typed, and T_X is the type of X, e.g., person, document, etc. Each entity X has a set of attributes $\mathcal{A}(T_X)$ and may participate in a set of relations $\mathcal{R}(T_X)$. Using FOL, $a(X, V)$ asserts that X has attribute $a \in \mathcal{A}(T_X)$ with value V. Using OO, $X.a$ returns the value of attribute a. Analogously, for a relation $r \in \mathcal{R}(T_X)$, in FOL $r(X, Y)$ indicates that X and Y are related via r, and in OO, $X.r$ refers to the entities related to X via r which allows us to succinctly express set terms as defined below.

In addition to variables and constants, terms in PSL rules can also refer to sets of entities. To define set terms, we use an object-oriented rather than first-order logic syntax for simplicity. If $X.edited$ refers to a document X edited, then $\{X.edited\}$ refers to the set of *all* documents edited by X. Using set terms, one can reason about the aggregate similarity or probability of sets of entities. For instance, $\{X.edited\} \overset{s_3}{=} \{Y.edited\}$ means that *the entire set* of documents edited by X is similar to the set of Y's edits according to the user defined aggregate similarity function s_3. The ability to reason about set aggregates extends the modelling capabilities of PSL beyond existing SRL frameworks and yields better predictive performance on some inference tasks.

PSL program is a set of PSL rules and constraints. A PSL program may contain three types of rules: soft rules, each of which has a weight that determines the relative importance of the rule, as discussed below; hard constraints, which are always required to hold; and exclusivity constraints. Hard constraints can be viewed as rules with infinite weight but are maintained separately in PSL to enforce them throughout inference. An exclusivity constraint on a relation r and entity X states that X can be related to at most one entity via r.

The semantics of soft truth values depends on the domain of reasoning. In our Wikipedia example, truth values denote document and user similarities. Table 9.1 shows an example PSL program for our Wikipedia domain. The first rule states that if two documents have similar text, then they are similar; the second rule states that two documents are similar if the sets of their editors are similar; the third rule states that two documents are similar if the sets of their first- and second-order neighbors in the hyperlink graph are similar; the fourth rule encodes transitivity of similarity and is a hard constraint. The similarity function s_n is based on attributes and $s_{\{\}}$ is defined below.

However, unlike in binary logic, here the conjunction and implication operators need to combine similarities or probabilities, which are real numbers in $[0, 1]$. To emphasize this distinction, we have placed a tilde over these operators in Table 9.1. Moreover, we would also like to have "soft" versions of disjunction and negation that behave as their counterparts from binary logic so that we are able to manipulate PSL rules as in logic. For example, the fourth rule could be rewritten as a disjunction as follows: $\tilde{\neg}(A \cong B) \tilde{\vee} \tilde{\neg}(B \cong C) \tilde{\vee} A \cong C$. One set of such truth-combining operators that generalize their Boolean counterparts is provided by t-norms and their corresponding t-conorms [84]. In PSL, any t-norm/t-conorm pair may be used. We used the Lukasiewicz t-(co)norm, defined as follows:

$$a \tilde{\wedge} b = \max\{0, a + b - 1\} \tag{9.1}$$

$$a \tilde{\vee} b = \min\{a + b, 1\} \tag{9.2}$$

$$\tilde{\neg} a = 1 - a \tag{9.3}$$

Above, $a, b \in [0, 1]$ can be similarities or Boolean truth values. The Lukasiewicz t-norm is appealing because it is linear in the values being combined, and because unlike, for example, the product t-norm, which defines $a \tilde{\wedge} b = ab$, the Lukasiewicz t-norm leads to sparser grounded PSL programs because the $\tilde{\wedge}$ operator evaluates to 0 on all a, b for which $a + b < 1$. On the other hand, the product t-norm may be more appropriate in domains in which it is important to model longer-range dependencies.

For computational efficiency, we currently require sets to be fully observed, e.g., all groundings of the *edited* relation in $\{A.edited\}$ must be provided as evidence. However, as an essential feature, PSL supports set similarity functions that combine the results of reasoning over similarities between individual members of the sets. For example, in $\{A.edited\} \overset{s_{\{\}}}{=} \{B.edited\}$, $s_{\{\}}$ can be defined as

$$s_{\{\}} = \frac{\sum_{i \in \{A.edited\}} \sum_{j \in \{B.edited\}} \cong (i, j)}{|\{A.edited\}| + |\{B.edited\}|},$$

where $\cong$ is a soft predicate subject to an exclusivity constraint denoting the similarity of two individuals (written here in prefix notation). The values of $\cong$ for any two individuals can be inferred from the data and are *not* required to be part of the evidence.

9.2.2 Semantics

A PSL program defines a probability distribution over similarities or probabilities between entities or sets of entities in a domain. Based on the weight annotated rules, PSL constructs the probabilistic model which we refer to as a *constrained continuous Markov random fields* (CCMRF).

9.2.2.1 Constraint Continuous Markov Random Field

Continuous Markov random fields are a general and expressive formalism to model complex probability distributions over multiple continuous random variables. Potential functions, which map the values of sets (cliques) of random variables to real numbers, capture the dependencies between variables and induce a exponential family density function as follows: Given a finite set of n random variables $\mathbf{X} = \{X_1, \dots, X_n\}$ with an associated bounded interval domain $D_i \subset \mathbb{R}$, let $\phi = \{\phi_1, \dots, \phi_m\}$ be a finite set of m *continuous* potential functions defined over the interval domains, i.e., $\phi_j : \mathbf{D} \to [0, M]$, for some bound $M \in \mathbb{R}^+$, where $\mathbf{D} = D_1 \times D_2 \dots \times D_n$. For a set of free parameters $\Lambda = \{\lambda_1, \dots, \lambda_m\}$, we then define the probability measure $\mathbb{P}$ over $\mathbf{X}$ with respect to ϕ through its density function f as:

$$f(\mathbf{x}) = \frac{1}{Z(\Lambda)} \exp[-\sum_{j=1}^{m} \lambda_j \phi_j(\mathbf{x})] \quad ; \quad Z(\Lambda) = \int_{\mathbf{D}} \exp\left(-\sum_{j=1}^{m} \lambda_j \phi_j(\mathbf{x})\right) d\mathbf{x} \qquad (9.4)$$

where Z is the normalization constant. The definition is analogous to the popular discrete Markov random fields (MRF) but using integration over the bounded domain rather than summation for the partition function Z.
In addition, we assume the existence of a set of k_A equality and k_B inequality constraints on the random variables, that is, $A(\mathbf{x}) = a$, where $A : \mathbf{D} \to \mathbb{R}^{k_A}, a \in \mathbb{R}^{k_A}$ and $B(\mathbf{x}) \leq b$, where $B : \mathbf{D} \to \mathbb{R}^{k_B}, b \in \mathbb{R}^{k_B}$. Both equality and inequality constraints restrict the possible combinations of values the random variables $\mathbf{X}$ can assume. That is, we set $f(\mathbf{x}) = 0$ whenever any of the constraints are violated and constrain the domain of integration, denoted $\tilde{\mathbf{D}}$, for the normalization constant correspondingly. Constraints are useful in probabilistic modeling to exclude inconsistent outcomes based on prior knowledge about the distribution. We call this class of MRFs *constrained continuous Markov random fields* (CCMRF).

9.2.2.2 Probabilistic Model of PSL

Given a PSL program $\mathcal{P}$ and data domain $\mathcal{D}$, we now define how $\mathcal{P}$ gets grounded against $\mathcal{D}$ to induce a CCMRF. In the following discussion, we require that all weights be positive. This requirement does not detract from the generality of PSL because any negative weight can be made positive by negating the corresponding rule.

For the given domain $\mathcal{D}$, each **grounding** of each PSL rule R represents an instantiation of all variables in R by replacing them with entities from $\mathcal{D}$. For each rule, all possible groundings are generated. For example, let R be:

$$\{A.editor\} \overset{s_1}{=} \{B.editor\} \tilde{\wedge} A.text \overset{s_2}{=} B.text \tilde{\Rightarrow} A \overset{s_3}{=} B$$

Suppose $\mathcal{D}$ contains the entities *doc*1 and *doc*2. Then, there are 4 unique **groundings** of R, corresponding to the possible ways of replacing A and B with *doc*1 and *doc*2, and in each grounding we expand the *editor* relation and the *text* attribute of each participating entity. A statement of the form $a \overset{s_i}{=} b$, where a and b are entities or sets of entities, is called a **ground proposition**. Each ground proposition represents a statement about particular entities and can be assigned a truth value by the function s_i. Let $\mathcal{G}$ be the set of all ground propositions with the entities in $\mathcal{D}$. Let I be an **interpretation,** i.e., a particular truth assignment to the elements in $\mathcal{G}$, such that for each $g \in \mathcal{G}$, its truth value is a real number between 0 and 1, i.e., $I(g) \in [0, 1]$.

Definition 9.1. The distance from satisfaction of a single ground rule r, according to a particular interpretation I, is $d(r, I) = 1 - I(r)$.

Intuitively, the closer the value of a particular ground rule is to 1, the closer it is to being satisfied, and the smaller its distance from satisfaction.

We now induce a CCMRF as follows:

Definition 9.2. Given a PSL program $\mathcal{P}$, data domain $\mathcal{D}$ and fixed Minkowski distance metric δ, we define the induced CCMRF, denoted $\mathbb{P}_{\mathcal{P},\mathcal{D}}$ with random variables $\mathbf{X}_{\mathcal{P},\mathcal{D}}$, potential functions $\phi_{\mathcal{P},\mathcal{D}}$, free parameters $\Lambda_{\mathcal{P},\mathcal{D}}$, equality constraints $A_{\mathcal{P},\mathcal{D}}$, and inequality constraints $B_{\mathcal{P},\mathcal{D}}$ by the following procedure:

1. For each ground atom $a_i \in \mathcal{G}$ we introduce a continuous random variable X_i to $\mathbf{X}_{\mathcal{P},\mathcal{D}}$ with associated domain $D_i = [0, 1]$. If a_i denotes a set aggregate atom defined via some aggregate similarity or probability function s over a fixed set of atoms $\{a_{i_1}, \ldots, a_{i_k}\}$ we introduce an equality constraint $X_i = s(X_{i_1}, \ldots, X_{i_k})$ in $A_{\mathcal{P},\mathcal{D}}$.

2. For each ground soft rule r_j in $\mathcal{G}$ with weight annotation w_j we add a potential function $\phi_j = d(r_j, \tilde{I})^\delta$ to $\phi_{\mathcal{P},\mathcal{D}}$ with corresponding parameter $\lambda_j = w_j$ and range $[0, 1]$. $\tilde{I}$ is defined such that it evaluates rules and propositions according to the chosen real-valued semantics for PSL and evaluates $\tilde{I}(a_i) = X_i$ for all ground atoms $a_i \in \mathcal{G}$.

3. For each hard rule r_j in $\mathcal{G}$, we add an equality constraint of the form $d(r_j, \tilde{I}) = 0$ to the equality constraint matrix $A_{\mathcal{P},\mathcal{D}}$.

4. For each domain integrity constraint on the binary relationship $R \in \mathcal{P}$, we add one linear inequality constraint to $B_{\mathcal{P},\mathcal{D}}$ for each entity $e \in \mathcal{D}$ in our domain as follows: Let $\{d_1, \ldots, d_n\}$ be the set of entities in $\mathcal{D}$ then we let $a_k = R(e, d_k)$ denote the ground atom where the entity d_k appears as the second argument (i.e. range). Let X_k denote the associated random variable in $\mathbf{X}_{\mathcal{P},\mathcal{D}}$. We then add the inequality constraint $\sum_{k=1}^{n} X_k \leq 1$. Similarly, we add constraints for range integrity constraints and equality integrity constraints.

Hence, each PSL program $\mathcal{P}$ together with a data domain $\mathcal{D}$ is associated with a unique CCMRF $\mathbb{P}_{\mathcal{P},\mathcal{D}}$. The continuous random variables $\mathbf{X}_{\mathcal{P},\mathcal{D}}$ map uniquely onto the ground atoms of the grounded PSL program $\mathcal{P}$ with respect to $\mathcal{D}$. Therefore, any interpretation I on the ground atoms in $\mathcal{G}$ is uniquely associated with a value assignment $\mathbf{x}$ to the random variables $\mathbf{X}_{\mathcal{P},\mathcal{D}}$. Vice versa, any value assignment $\mathbf{x}$ to the random variables directly translates to an interpretation I. Thus, we abuse notation and denote the probability distribution over truth value assignments by $\mathbb{P}_{\mathcal{P},\mathcal{D}}(I)$. When the PSL program $\mathcal{P}$ and data domain $\mathcal{D}$ are clear from the context, we omit the index on their associated CCMRF and therefore simply refer to the random variables by $\mathbf{X}$ and to the probability distribution by $\mathbb{P}$.

The Minkowski distance metric δ (i.e. $\delta \in (0, \infty)$) presents another choice point by which members in the PSL family of languages are identified. For example, if we set $\delta = 1$, reflecting the L_1-norm (or Manhatten distance) $\delta_{L_1}(x, y) = \|x - y\|_1$, we obtain the log-linear representation, commonly used in SRL and graphical models. Alternatively, we could set $\delta = 2$, reflecting the L_2-norm (or Euclidean distance) $\delta_{L_2}(x, y) = \|x, y\|_2^2$, thus making the penalty for not satisfying a rule a faster-growing function of the distance from satisfaction.

9.3 The Importance of Similarity

The ability to reason about similarities in a relational framework is a central novel property of PSL. Here we motivate its importance. The most immediate advantage of reasoning about similarity is that, in PSL, the numerous well-understood domain-specific similarity measures that exist in the literature can be easily brought to bear in a relational context. A further advantage results from the interplay of relational structure and similarity; namely, PSL supports reasoning about similarity not only between the attributes of two entities X and Y, but also between the respective sets of entities related to X and Y, e.g., the sets of entities related to X and Y via the *editor* relation.

Because support for set similarity is such an important aspect of PSL, we consider it further by contrasting setFree-PSL, in which set similarity is not allowed, to the complete PSL. Suppose we would like to reason about the similarity between two documents based on the editors they have in common.

This is expressed in setFree-PSL as:

$$A.editor \overset{s_s}{=} B.editor \tilde{\Rightarrow} A \cong B \tag{9.5}$$

The main issue with this rule is that the number of its groundings that are active during inference depends on the *absolute* number of editors that A and B have in common. Consider what happens as a result. Let a_1 and b_1 be two documents, each having n editors with perfect overlap between their editor sets; and let a_2 and b_2 be two documents each having m editors, $m \gg n$, such that they have n editors in common. Then, all else being equal, the penalty for not inferring that a_1 and b_1 are similar is equal to the penalty for not inferring that a_2 and b_2 are similar, although in the former case we have much stronger evidence of the similarity of the two documents. A related issue is that to maintain the relative importance of rules constant across domains, when rules such as the above are present in the model, their weight needs to depend on the sizes of the relations. For example, if a weight for the above rule is learned in one data set and used for prediction in another one in which documents have larger numbers of editors, all else being equal, the relative importance of that rule will increase simply because it will have more active groundings during inference.

These issues are completely resolved by the introduction of sets. For example, in PSL we can write:

$$\{A.editor\} \overset{s_{\{\}}}{=} \{B.editor\} \tilde{\Rightarrow} A \cong B \tag{9.6}$$

This rule now has a single grounding, and the strength of the evidence on the left equals the amount of overlap between the two sets. Sets are also beneficial when they appear in the consequent of a rule. Consider the difference between the following two rules that relate the similarity of two concepts in a taxonomy tree to the similarity of their sub-concepts[1]:

$$C_1 \overset{s_1}{=} C_2 \tilde{\Rightarrow} C_1.subconcept \overset{s_1}{=} C_2.subconcept \tag{9.7}$$

$$C_1 \overset{s_1}{=} C_2 \tilde{\Rightarrow} \{C_1.subconcept\} \overset{s_{\{\}}}{=} \{C_2.subconcept\} \tag{9.8}$$

For two similar concepts that each have n sub-concepts, the first rule will have at most n true groundings out of n^2 possible groundings (even if their sub-concepts align perfectly), while the second rule correctly captures the intended meaning.

As a further benefit of sets, using rules such as (9.6) and (9.8) instead of (9.5) and (9.7) leads to fewer groundings per rule. Specifically, if rule (9.8) is used, then there will be a single grounding for each pair of concepts C_1, C_2; on the other hand, if rule (9.7) is used, there will be k^2 groundings for each pair C_1, C_2, where k is the maximum relation size.

9.4 MAP Inference

Given a PSL model $\mathcal{P}$ and data domain $\mathcal{D}$, we are interested finding that assignment of truth values I to the ground atoms in $\mathcal{G}$ which is most likely, i.e. maximizes the probability $\mathbb{P}(I)$. For example, in the Wikipedia document similarity task, we would like to predict the similarity between documents which is most likely.

Infering the most likely values for a set of propositions, given observed values for the remaining (evidence) propositions, is called maximum *a posteriori* (MAP) inference (also called MPE inference). We split the set of propositions into two subsets: let $\mathbf{Y}$ be the set of propositions with unknown values and let $\mathbf{X}$ be the set of evidence (i.e. known) propositions with values in $I(\mathbf{X})$[2]. Then the task is to find a truth assignment $I_{MAP}(\mathbf{Y})$ that is most likely according to the PSL program $\mathcal{P}$,

[1]We use taxonomies as an example alignment problem here for illustration purposes only. For a more elaborate discussion of taxonomies and ontologies, please refer to Section 9.8.2

[2]Note, that we will use the same notation to refer to ground atoms and their uniquely associated random variables in the CCMRF

given the evidence:

$$
\begin{aligned}
I_{MAP}(\mathbf{Y}) &= \arg\max_{I(\mathbf{Y})} \mathbb{P}(I(\mathbf{Y})|I(\mathbf{X})) &\quad (9.9)\\
&= \arg\max_{\mathbf{y}} \mathbb{P}(\mathbf{Y}=\mathbf{y}|\mathbf{X}) &\quad (9.10)\\
&= \arg\max_{\mathbf{y}\text{ s.t. }\langle\mathbf{y},\mathbf{x}\rangle\in\tilde{\mathbf{D}}} \frac{1}{Z(\Lambda)}\exp[-\sum_{j=1}^{m}\lambda_j\phi_j(\mathbf{y},\mathbf{x})] &\quad (9.11)\\
&= \arg\max_{\mathbf{y}\text{ s.t. }\langle\mathbf{y},\mathbf{x}\rangle\in\tilde{\mathbf{D}}} -\sum_{j=1}^{m}\lambda_j\phi_j(\mathbf{y},\mathbf{x}) &\quad (9.12)
\end{aligned}
$$

where $\tilde{D}$ is the constraint domain of the CCMRF as defined in 9.2.2.1. That is, we require that any assignment of truth values $\mathbf{y}$ considered in Equation 9.12 satisfies the equality constraints A and inequality constraints B. Hence, finding the most likely interpretation, or, equivalently, the most likely values of the random variables, is a constraint numeric optimization problem with the objective function in Equation 9.12 and the constraints given by A and B.

In the full generality of PSL, there is little we can say about the complexity of the MAP inference problem other than it is a general constraint optimization problem which are intractable to solve. Hence, we now introduce a special class of PSL programs restricted to certain t-norms and distance metrics.

Definition 9.3. We call a PSL program $\mathcal{P}$ a *conePSL* program if and only if

1. The semantics of PSL rules is defined by the Lukasiewicz t-norm. That is, $\tilde{\vee}, \tilde{\wedge}$ are implemented by the Lukasiewicz t-(co)norms $\otimes(x,y) = \max(x+y-1,0)$ and $\oplus(x,y) = \min(x+y,1)$, respectively.

2. All combining functions for both similarities and probabilities (e.g. set equality) are linear.

3. The L_1- or L_2 distance norm is used in the potential functions. That is, $\delta = 1$ or $\delta = 2$.

conePSL is a particular subset of the PSL language family restricted to linear t-norms and similarity functions as well as absolute or squared distance functions. Despite these restrictions, conePSL is still a very expressive language. In fact, all of our experimental evaluations in Section 9.8 were implemented in conePSL. This subset of PSL has some nice theoretical properties for both MAP and marginal inference[3].

Theorem 9.1. *MAP inference for a given conePSL program $\mathcal{P}$ and data domain $\mathcal{D}$ can be cast as a convex optimization problem which is solvable in time polynomial, $O(|\mathcal{G}|^{3.5})$, in the number of ground rules, atoms and constraints $\mathcal{G}$.*

[3]We will discuss marginal inference in Section 9.5.

Complexity of conePSL. The proof is organized as follows. We show how to generate a convex optimization problem with a set of linear and conic constraints for a given conePSL program $\mathcal{P}$ applied to data domain $\mathcal{D}$ such that the optimal solution to this problem corresponds to a MAP state of the associated CCMRF. The class of convex optimization problems with linear and conic constraints is called *Second Order Cone Programs* (SOCP). SOCPs can be solved in polynomial time using interior point methods [109] which gives the desired result. Moreover, it has been shown that SOCPs can be solved almost as efficiently as Linear Programs [7]. Various commercial and open source optimization toolboxes can solve SOCPs[4].

SOCP can be understood as a generalization of Linear Programming with additional support for conic constraints. Given a vector $o \in \mathbb{R}^n$, a matrix $C \in \mathbb{R}^{m \times n}$, and a vector $c \in \mathbb{R}^m$, a linear program in o, C, c asks for the vector $x \in \mathbb{R}^n$ which satisfies:

$$\max o^T x$$

subject to

$$Cx \leq c$$

Here, o denotes the linear objective function and C, b denote the set of linear constraints.
In addition, a second order cone program allows constraints of the form

$$x_i x_j \geq \left\| c^T x \right\|_2^2 \ , \ c \in \mathbb{R}^n, \ x_i, x_j \geq 0$$

Note, that SOCP does allow more general constraints, but the one given is sufficiently general for conePSL.

We will now describe the transformation process of the optimization problem in Equation 9.12 for the CCMRF associated with $\mathcal{P}, \mathcal{D}$ into a SOCP in detail.

1. For each random variable $X \in \mathbf{X}$ we declare a corresponding variable in the optimization problem. We denote the index of this atom variable v_X. If the value of X is fixed for the MAP inference (i.e. its considered evidence) then we introduce an equality constraint for its variable v_X and the fixed value.

2. Add upper and lower bound constraints in C for each introduced variable v as $0 \leq v \leq 1$.

3. Let ϕ_j be any potential function in ϕ. There there exists a corresponding ground soft rule in $\mathcal{G}$ by Definition 9.2. Let $r = H \Leftarrow B_1 \tilde{\wedge} B_2 \tilde{\wedge} \ldots \tilde{\wedge} B_n$ be that rule. We declare a variable v_j corresponding to ϕ_j (and therefore r) in the optimization problem. According to the semantics of a PSL-rule we have

[4]The Wikipedia article on Second Order Cone Programming (`http://en.wikipedia.org/wiki/Secondorderconeprogramming`) lists 11 commercial and open source optimization toolboxes with support for SOCP

$I(r) = (\oplus_i I(\neg B_i)) \oplus I(H)$, hence $d(r, I) = \max(1 - \sum_i (1 - I(B_i)) - I(H), 0)$ under the Lukasiewicz t-conorm. Therefore, we introduce the constraints $v_j \geq 1 - \sum_i (1 - b_i) - h$ and $v_j \geq 0$, where b_i, h denote the variables that were introduced for the random variables associated with the ground atoms B_i, H (see above).

4. Finally, replacing all potential functions ϕ_j with their associated variable v_j, the objective function of Equation 9.12 can be rewritten as $-\sum_{j=1}^{m} \lambda_j v_j^{\delta}$. If $\delta = 1$, then this equation is linear and we can use it as the objective function o of the SOCP optimization problem. If $\delta = 2$, then we introduce the additional variable $\tilde{v}$,the conic constraint $\tilde{v}^2 \geq \sum_{j=1}^{m} (\sqrt{\lambda_j} v_j)^2$, and set $o = -\tilde{v}$

5. By our assumptions about aggregate similarity and probability function and by Definition 9.2 all (in)equality constraints in A and B must be linear and hence they can be copied to the linear constraint matrix C after suitable replacement of random variables X_i by v_i.

The vector $x \in \mathbb{R}^n$ which maximizes $o^T x$ with respect to the introduced constraints yields the MAP state of the CCMRF and therefore the interpretation I_{MAP}.

Clearly, all the constraints we have introduced above are either linear or conic. Therefore, the resulting optimization problem with the linear objective function o is a second order cone program and can be solved efficiently in polynomial according to the standard complexity result for SOCP [91], namely $O(n^{3.5})$ where n refers to the number of variables in the optimization problem.

Note, that the size of the optimization problem produced by the above procedure is linear in the number of ground rules and ground atoms. Hence, MAP inference in conePSL has polynomial time complexity in the number of ground rules and ground atoms in $\mathcal{G}$ of a given program $\mathcal{P}$ by extension of the SOCP complexity result. The number of ground rules and atoms depends on the particular rule set and the size of the domain $\mathcal{D}$ of a PSL program $\mathcal{P}$ and is of the order $O(|\mathcal{D}|^v)$ in the worst case, where $|\mathcal{D}|$ denotes the number of entities in the domain base and v is the maximum number of distinct variables appearing in any one rule in $\mathcal{P}$.

□

Our discussion thus far assumed that we have to consider all ground rules and atoms. This can be prohibitively expensive depending the number and type of rules and the size of the data domain. To limit the number of grounded rules that are active during inference, we take advantage of the fact that only grounded rules that evaluate to strictly less than 1 need to be considered. Thus, grounded rules that have value 1 given the evidence $I(\mathbf{x})$ can be excluded from consideration because their value does not depend on assignments to the propositions in $\mathbf{y}$ and thus don't matter in the optimization problem. Furthermore, rather than performing inference over all remaining grounded rules at once, we employ a lazy grounding technique, whereby only grounded rules whose value becomes smaller than 1 at some point during inference are included in the inference problem. Such rules are called **activated** in Alg. 9.2, which describes the MAP inference algorithm in PSL. This

```
Algorithm MAP Inference
Input: Evidence and inference random variables $\mathbf{x}, \mathbf{y}$

1  $I_0(\mathbf{y}) \leftarrow$ all zeros assignment
2  $R \leftarrow$ all grounded rules activated by $I(\mathbf{x}) \cup I_0(\mathbf{y})$
3  while $R$ has been updated do
4     $i \leftarrow$ current iteration
5     $O \leftarrow$ generateSOCP($R$)
6     $I_i(\mathbf{y}) \leftarrow$ optimize(O)
7     for each Proposition $y \in \mathbf{y}$ do
8        if $I_i(y) > \theta(\theta = 0.01)$ do
9           $R_y \leftarrow$ activated rules containing $y$
10          $R \leftarrow R \cup R_y$
11       end
12    end
13 end
```

Figure 9.2: MAP-Inference in PSL

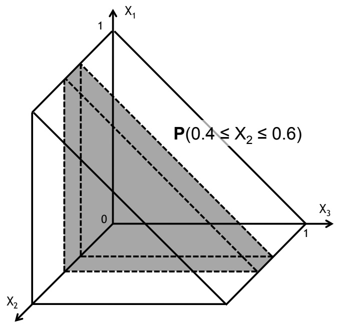
algorithm works by transforming the grounded conePSL program into its CCMRF and generating a SOCP for the optimization problem in 9.12 (line 4) whose solution gives an assignment to the propositions in y (line 5). Any rules that have not achieved their maximal possible value of 1 in the current solution, are added to the set of active ground rules (lines 6-10) and the process repeats for as long as new rules are activated. Generating the SOCP (line 5) follows the procedure outlined above but restricted to the set of active rules and atoms. To solve the SOCP, one can use any available technique [7].

Alg. 9.2 is in essence equivalent to cutting plane inference (CPI) [126], except that, unlike CPI, PSL programs are continuous constrained numeric optimization programs.

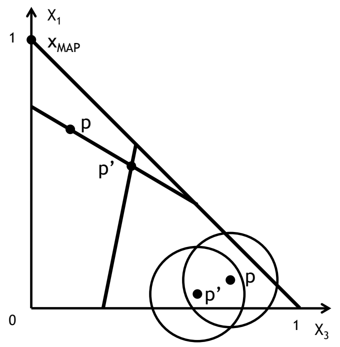
9.5 Marginal Inference

MAP inference computes the *single* most likely truth value assignment to *all* ground atoms. Marginal inference, in constrast, determines the entire probability distribution for a subset of the ground atoms. For instance, we might be particularly interested in the similarity between two documents, a and b. We may then ask for the marginal distribution over the similarity values for $a \cong b$ which will tell us the likelihood of each potential similarity value in $[0, 1]$.

Since the probability distribution for a PSL program is defined by its associated CCMRF, marginal inference in PSL requires marginal inference in CCMRFs which is the subject of this section. We start our study of marginal computation for CCMRFs by proving that computing the exact density function is #P hard (9.5.1). In Section 9.5.2, we discuss how to approximate the marginal distribution using a



a) Example of geometric marginal computation



b) Hit-and-Run and random ball walk illustration

MCMC sampling scheme which produces a guaranteed ϵ-approximation in polynomial time under suitable conditions. We show how to improve the sampling scheme by detecting phases of slow convergence and present a technique to counteract them (9.5.3). Finally, we describe an algorithm based on the sampling scheme and its improvements (9.5.4). In addition, we discuss how to relax the linearity conditions in Section 9.5.5.
Throughout this discussion we use the following simple example for illustration:

Example 9.1. Let $\mathbf{X} = \{X_1, X_2, X_3\}$ be subject to the inequality constraint $x_1 + x_3 \leq 1$. Let $\phi_1(\mathbf{x}) = x_1$, $\phi_2(\mathbf{x}) = max(0, x_1 - x_2)$, $\phi_3(\mathbf{x}) = max(0, x_2 - x_3)$ where $\lambda = (1, 2, 1)$ are the associated free parameters.

9.5.1 Exact marginal computation

Theorem 9.2. *Computing the marginal probability density function* $f_{\mathbf{X}'}(\mathbf{x}') = \int_{\mathbf{y} \in \times \tilde{D}_i, s.t. X_i \notin \mathbf{X}'} f(\mathbf{x}', \mathbf{y}) d\mathbf{y}$ *for a subset* $\mathbf{X}' \subset \mathbf{X}$ *under a probability measure* $\mathbb{P}$ *defined by a CCMRF is #P hard in the worst case.*

We prove this statement by a simple reduction from the problem of computing the volume of a n-dimensional polytope defined by linear inequality constraints. To see the relationship to computational geometry, note that the domain $\mathbf{D}$ is a n-dimensional unit hypercube[5]. Each linear inequality constraint B_i from the system B can be represented by a hyperplane which "cuts off" part of the hypercube $\mathbf{D}$. Finally, the potential functions induce a probability distribution over the resulting convex polytope. Figure 9.5a) visualizes the domain for our running example in the 3-dimensional Euclidean space. The constraint domain is shown as a wedge. The highlighted area marks the region of probability mass that is equal to the probability $\mathbb{P}(0.4 \leq X_2 \leq 0.6)$.

Sketch. For any random variable $X \in \mathbf{X}$, the marginal probability $\mathbb{P}(l \leq X \leq u)$ under the uniform probability distribution defined by a single potential function

[5]We ignore equality constraints for now until the discussion of the algorithm in Section 9.5.4

$\phi = 0$ corresponds to the volume of the "slice" defined by the bounds $u < l \in [0, 1]$ relative to the volume of the entire polytope. In [20] it was shown that computing the volume of such slices is at least as hard as computing the volume of the entire polytope which is known to be #P-hard [40]. □

9.5.2 Approximate marginal computation and sampling scheme

Despite this hardness result, efficient approximation algorithms for convex volume computation based on MCMC techniques have been devised and yield polynomial-time approximation guarantees. We will review the techniques and then relate them to our problem of marginal computation.

The first provably polynomial-time approximation algorithm for volume computation was based on "random ball-walks". Starting from some initial point p inside the polytope, one samples from the local density function of f restricted to the inside of a ball of radius r around the point p. If the newly sampled point p' lies inside the polytope, we move to p', otherwise we stay at p and repeat the sampling. If $\mathbb{P}$ is the uniform distribution (as typically chosen for volume computation), the resulting Markov chain converges to $\mathbb{P}$ over the polytope in $O^*(n^3)$ steps assuming that the starting distribution is not "too far" from $\mathbb{P}$ [74].[6]

More recently, the hit and run sampling scheme [139] was rediscovered which has the advantage that no strong assumptions about the initial distribution needs to be made. As in the random ball walk, we start at some interior point p. Next, we generate a direction d (i.e., n dimensional vector of length 1) uniformly at random and compute the line segment l of the line $p + \alpha d$ that resides inside the polytope. We then compute the distribution of $\mathbb{P}$ over the segment l, sample from it uniformly at random and move to the new sample point p' to repeat the process. For $\mathbb{P}$ being the uniform distribution, the Markov chain also converges after $O^*(n^3)$ steps but for hit-and-run we only need to assume that the starting point p does not lie on the boundary of the polytope [92]. In [20], the authors show that hit-and-run significantly outperforms random ball walk sampling in practice, because it (1) does not get easily stuck in corners since each sample is guaranteed to be drawn from inside the polytope, (2) does not require parameter setting like the radius r which greatly influences the performance of random ball walk. Figure 9.5 b) shows an iteration of the random ball walk and the hit-and-run sampling schemes for our running example restricted to just two dimensions to simplify the presentation. We can see that, depending on the radius of the ball, a significant portion may not intersect with the feasible region.

Lovász and Vempala[92] have proven a stronger result which shows that hit-and-run sampling converges for general log-linear distributions. Based on their result, we get a polynomial-time approximation guarantee for distributions induced by CCMRFs as defined above.

[6]The O^* notation ignores logarithmic and factors and dependence on other parameters like error bounds.

Theorem 9.3. *The complexity of computing an approximate distribution σ^* using the hit-and-run sampling scheme such that the total variation distance of σ^* and $\mathbb{P}$ is less than ϵ is $O^*\left(\tilde{n}^3(k_B + \tilde{n} + m)\right)$, where $\tilde{n} = n - k_A$, under the assumptions that we start from an initial distribution σ such that the density function $d\sigma/d\mathbb{P}$ is bounded by M except on a set S with $\sigma(S) \leq \epsilon/2$.*

Sketch. Since A, B are linear, $\tilde{\mathbf{D}}$ is an $\tilde{n} = n - k_A$ dimensional convex polytope after dimensionality reduction through A. By definition, f is from the exponential family and since all factors are linear or maximums of linear functions, f is a log concave function (maximums and sums of convex functions are convex). More specifically, f is a log concave and log piecewise linear function. Let σ^s be the distribution of the current point after s steps of hit-and-run have been applied to f starting from σ. Now, according to Theorem 1.3 from [92], for $s > 10^{30}\frac{n^2R^2}{r^2}\ln^5\frac{MnR}{r\epsilon}$ the total variation distance of σ^s and $\mathbb{P}$ is less than ϵ, where r is such that the level set of f of probability $\frac{1}{8}$ contains a ball of radius r and $R^2 \geq \mathbf{E}_f(|x - z_f|^2)$, where z_f is the centroid of f.
Now, each hit-and-run step requires us to iterate over the random variable domain boundaries, $O(\tilde{n})$, compute intersections with the inequality constraints, $O(\tilde{n}k_B)$, and integrate over the line segment involving all factors, $O(\tilde{n}m)$. □

9.5.3 Improved sampling scheme

Our proposed sampling algorithm is an implementation of the hit-and-run MCMC scheme. However, the theoretical treatment presented above leaves two questions unaddressed: 1) How do we get the initial distribution σ? 2) The hit-and-run algorithm assumes that all sample points are strictly inside the polytope and bounded away from its boundary. How can we get out of corners if we do get stuck?

The theorem above assumes a suitable initial distribution σ, however, in practice, no such distribution is given. Lovász and Vempala also show that the hit-and-run scheme converges from a *single* starting point on uniform distributions under the condition that it does not lie on the boundary and at the expense of an additional factor of n in the number of steps to be taken (compare Theorem 1.1 and Corollary 1.2 in [92]). We follow this approach and use a MAP state $\mathbf{x}_{MAP}$ of the distribution $\mathbb{P}$ as the single starting point for the sampling algorithm. Choosing a MAP state as the starting point has two advantages: 1) we are guaranteed that $\mathbf{x}_{MAP}$ is an interior point and 2) it is the point with the highest probability density and therefore highest probability mass in a small local neighborhood.

However, starting from a MAP state elevates the importance of the second question, since the MAP state often lies exactly on the boundary of the polytope and therefore we are likely to start the sampling algorithm from a vertex of the polytope. The problem with corner points p is that most of the directions sampled uniformly at random will lead to line segments of zero length and hence we do not move between iterations. Let W be the subset of inequality constraints B that are "active" at the corner point p and b the corresponding entries in b, i.e. $Wp = b$ (since all constraints are linear, we abuse notation and consider B, W to be matrices). In other words, the hyperplanes corresponding to the constraints in W intersect in p. Now, for all directions $d \in \mathbb{R}^n$ such that there exist active constraints W_i, W_j with $W_i d < 0$ and $W_j d > 0$, the line segment through p induced by d must necessarily be 0. It also follows that more active constraints increase the likelihood of getting stuck in a corner.

For example, in Figure 9.5 b) the point $\mathbf{x}_{MAP}$ in the upper left hand corner denotes the MAP state of the distribution defined in our running example. If we generate a direction uniformly at random, only 1/4 of those will be feasible, that is, for all others we won't be able to move away from $\mathbf{x}_{MAP}$.

To avoid the problem of repeatedly sampling infeasible directions at corner points, we propose to restrict the sampling of directions to feasible directions only when we determine that a corner point has been reached. We define a corner point p as a point inside the polytope where the number of active constraints is above some threshold θ.[7] A direction d is feasible, if $Wd < 0$. Assuming that there are a active constraints at corner point p (i.e., W has a rows) we sample each entry of the a-dimensional vector z from $-|N(0,1)|$ where $N(0,1)$ is the standard Gaussian distribution with zero mean and unit variance. Now, we try to find directions d such that $Wd \leq z$.
A number of algorithms have been proposed to solve such systems of linear inequalities for feasible points d. In our sampling algorithm we implement the relaxation method introduced by Agmon [5] and Motzkin and Schoenberg [105] due to its simplicity. The relaxation method proceeds as follows: We start with $d_0 = \mathbf{0}$. At each iteration we check if $Wd_i \leq z$; if so, we have found a solution and terminate. If not, we choose the most "violated" inequality constraint W_k from W, i.e., the row vector W_k from W which maximizes $\frac{W_k d_i - z_k}{\|W_k\|}$, and update the direction,

$$d_{i+1} = d_i + 2\frac{z_k - W_k d_i}{\|W_k\|^2} W_k^T$$

The relaxation method is guaranteed to terminate, since a feasible direction d always exists [105].

9.5.4 Sampling algorithm

Putting the pieces together, we present the marginal distribution sampling algorithm in Figure 9.3. The inputs to the algorithm were discussed in Section 9.1. In addition, we assume that the domain restrictions $D_i = [l, u]$ for the random variables X_i are encoded as pairs of linear inequality constraints $l \leq \mathbf{x}_i \leq u$ in B, b. The algorithm first analyzes the equality constraints A to determine the number of "free" random variables and reduce the dimensionality accordingly. The singular-value decomposition of A is used to determine the $n \times n'$ projection matrix P which maps from the null-space of A to the original space $\mathbf{D}$, where $n' = n - rank(A)$ is the dimensionality of the null-space. If no equality constraints have been specified, P is the n-dimensional unit matrix. Next, the algorithm determines a MAP state $\mathbf{x}^0$

[7] We used $\theta = 2$ in our experiments.

```
Algorithm CCMRF Sampling
Input: CCMRF specified by RVs $\mathbf{X}$ with domains $\mathbf{D} = [0,1]^n$, equality constraints
    $A(\mathbf{x}) = a$ inequality constraints $B(\mathbf{x}) \leq b$, potential functions $\phi$, parameters $\Lambda$
Output: Marginal probability density histograms $H[\mathbf{X}_i] : [0,1] \to \mathbb{R}^+, \forall X_i \in \mathbf{X}$

1   if $A = \emptyset$
2     $P \leftarrow \mathbf{1}_{|\mathbf{X}|}$
3     $n' \leftarrow n$
4   else
5     $r \leftarrow \mathrm{rank}(A)$
6     $[U, \Sigma, V] \leftarrow \mathrm{svd}(A)$
7     $P \leftarrow V|_{\text{columns: } [r+1,n]}$
8     $n' \leftarrow n - r$
9   $\mathbf{x}^0 \leftarrow \mathrm{MAP}(A(\mathbf{x}) = a, B(\mathbf{x}) \leq b, \phi)$

10  $cornered \leftarrow$ FALSE
11  for $j = 0$ to $\rho$
12    if $cornered$
13      $\mathbf{d} \leftarrow \vec{0}$
14      $W \leftarrow B|_{rows:active} \times P$
15      $\mathbf{z} \leftarrow z_i = \sim -|N(0,1)| \; \forall i = 1 \ldots n'$
16      while $\exists i : W_k\mathbf{d} - z_k > 0$
17        $v \leftarrow \mathrm{argmax}_k \frac{W_k\mathbf{d} - z_k}{\|W_k\|}$
18        $\mathbf{d} = \mathbf{d} + 2\frac{z_v - W_v\mathbf{d}}{\|W_v\|^2} W_v^T$
19      $cornered \leftarrow$ FALSE
20    else
21      $\mathbf{d} \leftarrow d_i = \sim N(0,1) \; \forall i = 1 \ldots n'$
22    $\mathbf{d} \leftarrow \frac{1}{\|\mathbf{d}\|}\mathbf{d}$
23    $\mathbf{d} \leftarrow P \times \mathbf{d}$
24    $active \leftarrow \emptyset$
25    $\alpha_{low} \leftarrow -\infty, \alpha_{high} \leftarrow \infty$
26    $\mathbf{cd} \leftarrow B \times \mathbf{d}$ ; $\mathbf{cx} \leftarrow B \times \mathbf{x}^j$
27    for $i = 1 \ldots |\mathrm{rows}(B)|$
28      if $cd_i \neq 0$
29        $a = \frac{b_i - cx_i}{cd_i}$
30        if $cd_i > 0$ then $\alpha_{high} \leftarrow \min(\alpha_{high}, a)$
31        if $cd_i < 0$ then $\alpha_{low} \leftarrow \max(\alpha_{low}, a)$
32        if $a = 0$ then $active \leftarrow active \cup \{i\}$
33    if $\alpha_{high} - \alpha_{low} = 0 \wedge |active| > \theta$
34      $cornered \leftarrow$ TRUE
35      continue
36    $M \leftarrow \mathrm{map} : [0,1] \to \mathbb{R} \times \mathbb{R}$
37    for $\phi_i = \max(0, o_i \cdot \mathbf{x} + q_i) \in \phi$
38      $r \leftarrow \lambda_i(o_i \cdot \mathbf{d})$
39      $c \leftarrow \lambda_i(o_i \cdot \mathbf{x_j} + q_i)$
40      $a \leftarrow -c/r$
41      if $r > 0 \wedge a < \alpha_{high}$
42        $M(\max(a, \alpha_{low})) \leftarrow M(\max(a, \alpha_{low})) + [r, c]$
43      else if $r < 0 \wedge a > \alpha_{low}$
44        $M(\alpha_{low}) \leftarrow M(\alpha_{low}) + [r, c]$
45        if $a < \alpha_{high}$ then $M(a) \leftarrow M(a) + [-r, -c]$
46      else $M(\alpha_{low}) \leftarrow M(\alpha_{low}) + [0, c]$
47    $[r_\alpha, c_\alpha] \leftarrow \sum_{a \leq \alpha} M(a)$
48    $\Sigma_\alpha \leftarrow \sum_{a<b<\alpha \wedge \nexists c: a<c<b} \frac{1}{r_a} e^{-c_a}(e^{-r_a a} - e^{-r_a b})$
49    $s \leftarrow \sim [0, \Sigma_{\alpha_{high}}]$
50    $a \leftarrow \max\{\alpha \in M \mid \Sigma_\alpha \leq s\}$
51    $\alpha \leftarrow \frac{-1}{r_a}\left(\log\left(-sr_a + r_a\Sigma_a + e^{-c_a - r_a a}\right) + c_a\right)$
52    $\mathbf{x}^{j+1} \leftarrow \mathbf{x}^j + \alpha\mathbf{d}$
53    if $j > \frac{\rho}{100} n'^3$
54      $H[i][\mathbf{x}_i^{j+1}] \leftarrow H[i][\mathbf{x}_i^{j+1}] + 1 \; \forall i = 1 \ldots n$
```

Figure 9.3: Constrained continuous MRF sampling algorithm

of the density function defined by the CCMRF, which is the point with the highest probability mass, that is, $\mathbf{x}^0 = \mathrm{argmax}_{x \in \tilde{\mathbf{D}}} f(x)$. Computing the MAP state is cast as a convex optimization problem as described in Section 9.4.

After determining the null-space and starting point, we begin collecting ρ samples. If we detected being stuck in a corner during the previous iteration, we sample a direction $\mathbf{d}$ from the feasible subspace of all possible directions in the reduced null-space using the adapted relaxation method described above (lines 13-19). Otherwise, we sample a direction uniformly at random from the null-space of A. We then normalize the direction and project it back into our original domain $\mathbf{D}$ by matrix multiplication with P. The projection ensures that all equality constraints remain satisfied as we move along the direction $\mathbf{d}$. Next, we compute the segment of the line $l : x^j + \alpha\mathbf{d}$ inside the polytope defined by the inequality constraints B (lines 25-32). Iterating over all inequality constraints, we determine the value of α where l intersects the constraint i. We keep track of the largest negative and smallest positive values to define the bounds $[\alpha_{low}, \alpha_{high}]$ such that the line segment is defined exactly by those values of α inside this interval. In addition, we determine all active constraints, i.e. those constraints where the current sample point x^j is the point of intersection and hence $\alpha = 0$. If the interval $[\alpha_{low}, \alpha_{high}]$ is 0, then we are currently sitting in a corner. If, in addition, the number of active constraints exceed some threshold θ we are stuck in a corner and abort the current iteration to start over with restricted direction sampling.

In lines 36-48 we compute the cumulative density function of the probability $\mathbb{P}$ over the line segment l with $\alpha \in [\alpha_{low}, \alpha_{high}]$. For conePSL, the sum of potential functions $S = \sum_{i=1}^{m} \lambda_i \phi_i$ restricted to the line l is a continuous piece-wise linear function. In order to integrate the density function, we need to segment S into its differentiable parts, so we start by determining the subintervals of $[\alpha_{low}, \alpha_{high}]$ where S is linear and differentiable and can therefore be described by $S = rx + c$. We compute the slope r and y-intercept c for each potential function individually as well as the point of undifferentiability a where the line crosses 0. We use a map M to store the line description $[r, c]$ with the point of intersection a (lines 36-46). Then, we compute the aggregate slope r_a and y-intercept c_a for the sum of all potentials for each point of undifferentiability a (line 47) and use this information to compute the unnormalized cumulative density function by integrating over each subinterval and summing those up in Σ_α (line 48). Now, $\Sigma_a / \Sigma_{\alpha_{high}}$ gives the cumulative probability mass for all points of undifferentiability a which define the subintervals. Next, we sample a number s from the interval $[0, \Sigma_{\alpha_{high}}]$ uniformly at random (line 49) and compute α such that $\Sigma_\alpha = s$ (line 50-51). Finally, we move to the new sample point $\mathbf{x}^{j+1} = \mathbf{x}^j + \alpha \mathbf{d}$ and add it to the histogram which approximates the marginal densities if the number of steps taken so far exceeds the burn-in period which we configured to be 1% of the total number of steps.

9.5.5 Generalizing to conePSL

In our treatment so far, we has assumed that all constraints and potential functions are linear. However, Theorem 9.3 also holds when the inequality constraints as well as the potential functions are convex. A system of inequality constraints is convex if the set of all points that satisfy the constraints is convex, that is, any line connecting two points in the set is completely contained in the set.
Hence, we can extend our treatment of marginal distributions in general and the proposed sampling algorithm specifically to conePSL which allows conic constraints in addition to linear ones. Conic constraints are a particular type of convex constraint.

Our algorithm needs to be modified where we currently assume linearity. Firstly, computing a MAP state requires SOCP optimization as discussed in Section 9.4. Secondly, our method for finding feasible directions when being caught in a corner of the polytope needs to be adapted to the case of arbitrary conic constraints. One simple approach is to use the tangent hyperplane at the point $\mathbf{x}_j$ as an approximation to the actual constraint and proceed as is.

Similarly, we need to modify the computation of intersection points between the line and the conic constraints as well as how we determine the points of undifferentiability. Lastly, the computation of integrals over subintervals for the potential functions requires an extension to analytic integration of conic functions.

9.6 Weight learning

The weights are an important component of the probabiliy distribution for a PSL program $\mathcal{P}$ defined on a data domain $\mathcal{D}$ as defined in 9.2. Adjusting the weights tailors a PSL program to a particular domain for probabilistic inference and determines its accuracy. These weights can be configured by an expert user, but in most cases this is difficult to do or at least very cumbersome. This section describes how weights can be learned from available ground truth data. Suppose we are given an interpretation I with the *correct* truth value assignments to all ground rules and atoms in $\mathcal{G}$ for $\mathcal{P}, \mathcal{D}$. That is, the interpretation also assigns values to those ground atoms for which we would typically infer its values. Learning the weights then means finding that set of weights which maximizes the probablity of the ground truth interpretation I. Weight learning in PSL and CCMRFs is virtually identical to weight learning in standard Markov networks. Here, we briefly review standard weight learning algorithms applied to PSL and used in our experimental evaluation.

We try to find the weight vector Λ^* that maximizes the likelihood of I[8]:

$$\begin{aligned} \Lambda^* &= \arg\max\nolimits_\Lambda \mathbb{P}(I(\mathcal{G})) \\ &= \arg\max\nolimits_\Lambda \log \mathbb{P}(I(\mathcal{G})) \\ &= \arg\max\nolimits_\Lambda - \textstyle\sum_{j=1}^m \lambda_j \phi_j(\mathbf{x}) - \log Z(\Lambda) \end{aligned}$$

Recall that all ground rules derived from the same PSL rule share the same weight. Since each potential function is associated with a ground rule, most of the free parameters λ_i are coupled, i.e. refer back to the same PSL rule. For each of these rule weights, we can compute the gradient with respect to the above optimization equation. Let λ_k be the weight associated with the k-th PSL rule and $\delta = 1$. Then:

$$-\frac{\partial}{\partial \lambda_k} \log \mathbb{P}(I(\mathcal{G})) = \sum_{l=1}^{k_n} \phi_l(\mathbf{x}) - \mathbb{E}\left(\sum_{l=1}^{k_n} \phi_l(\mathbf{z})\right)$$

where we only consider the potentials ϕ_l associated with the k-th rule and $\mathbb{E}(\sum_{l=1}^{k_n} \phi_l(\mathbf{z}))$ is the expected value of $\sum_{l=1}^{k_n} \phi_l(\mathbf{z})$ with respect to the currently learned weights averaged over all possible truth value assignments $\mathbf{z}$.

We experimented with two ways of optimizing the above gradient: we used BFGS, a popular quasi-Newton method [115], and the Perceptron algorithm [35], where in both cases the expectation was approximated with the value of in the MAP state, which is a frequently used approximation since computing the expectation is intractable[9].

[8]In our experiments, we extend the objective function by a weight regularizer term.

[9]Computing the expectation exactly is intractable since it requires averaging over all truth value assignments of which there are exponentially many.

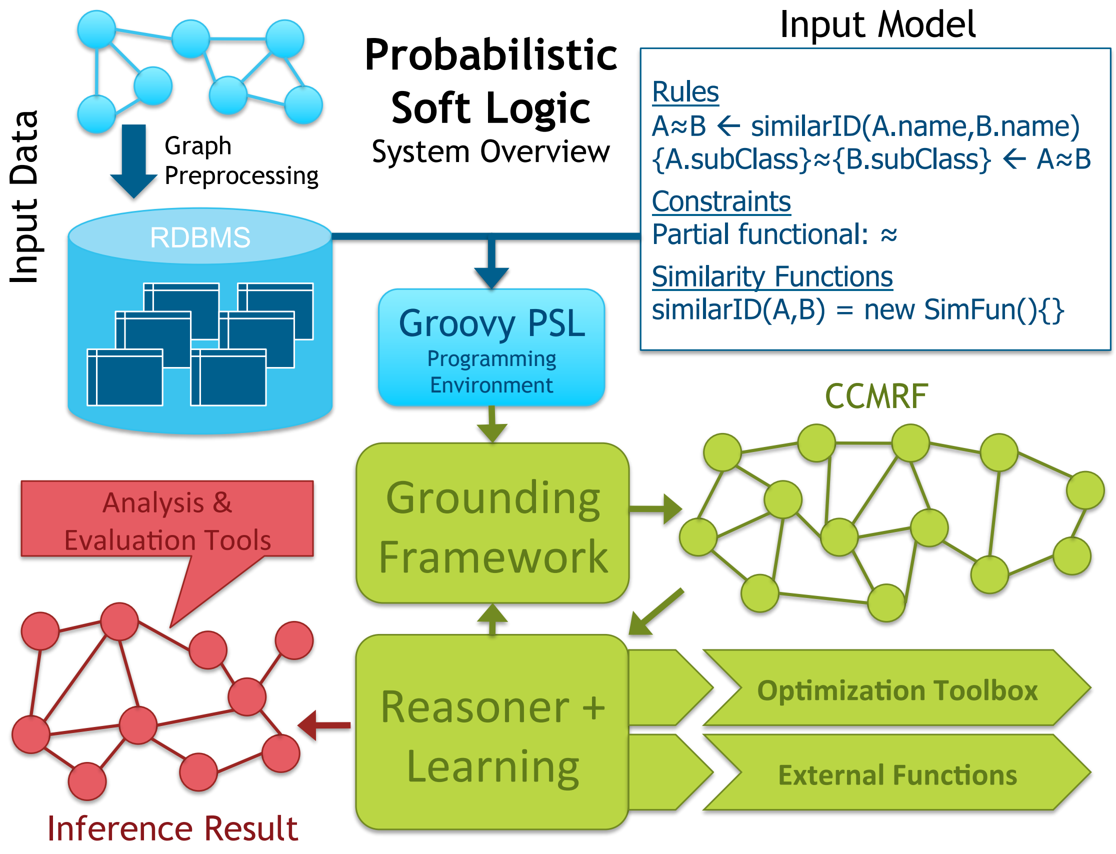


Figure 9.4: PSL System Architecture

Figure 9.4 shows the PSL system architecture. The PSL framework is implemented in Java. The input data can be any semi-structured data which is read in, converted according to the PSL model definition and loaded into a relational database system. PSL supports any relational database that is accessible via the common JDBC interface. For performance reasons, the default configuration uses an in-process database system like H2[10]. On top of the core PSL framework, a Groovy interface provides convenient, programmatic access to all functionality. Groovy is a scripting language for the Java virtual machine. We developed a PSL domain specific language in Groovy which allows the user to define the model, load data, execute weight learning, run inference and evaluate results. The Appendix 9.10 shows some example PSL programs written in the Groovy syntax. The Groovy interface calls the core PSL framework which has 4 major components: database access, grounding framework, reasoner and weight learning proxies, and evaluation. As described above, the grounding framework intelligently constructs the CCMRF. The reasoner and weight learning component convert the CCMRF into the respective numeric

[10]http://www.h2database.com

optimization problem which is fed into an industrial-strength optimization toolbox. If the optimal solution determined by the numeric solver changes the truth value of an atom, the system automatically determines all affected rules and grounds out any rules that might become unsatisfied as a consequence. A number of data structures are maintained to efficiently determine such changes. Changes to the ground rules are reflected in the numeric model which is maintained throughout the reasoning process and updated within the solver. This allows the solver to exploit knowledge about the previous optimal solution to quickly converge on the new solution. We used the MOSEK optimization toolbox (`http://www.mosek.com`) in our experiments reported in Section 9.8. This component also integrates external similarity functions. Lastly, the results of the inference process can be analyzed using the evalution component and the tools provided therein.

9.8 Experimental Evaluation

This section presents an empirical evaluation of PSL that addresses three questions:

1. Is PSL effective at modeling relational inference tasks?
2. How useful are the novel features provided by PSL?
3. How efficient are the proposed algorithms on real-world prediction and learning tasks?

We study these questions on two distinct problems, namely (a) category prediction and similarity propagation for Wikipedia documents and (b) ontology alignment on a standard corpus of bibliographic ontologies. After describing the data and the experimental methodology, we present results that demonstrate PSL's effectiveness on relational inference. We then investigate in more detail the importance of sets and the benefit of reasoning about similarity.

9.8.1 Wikipedia Category Prediction

We collected all Wikipedia articles that appeared in the featured list[11] in the period Oct. 7-21, 2009, thus obtaining 2460 documents. We used featured articles because they are richly connected, both by their hyperlinks and by their network of human editors [16]. After stemming and stop-word removal, we represented the text of each document as a tf/idf-weighted feature vector. Each document belongs to one of 19 distinct categories, which were obtained by using the category under which each featured article was listed. Some of the original categories that were similar were merged to ensure that each category contains sufficiently many documents. The data contains the relations Link(fromDoc, toDoc), which establishes a hyperlink between two documents; Talk(document, user), which states that the user edited the "Talk"

[11] `http://en.wikipedia.org/wiki/Wikipedia:Featured_lists`

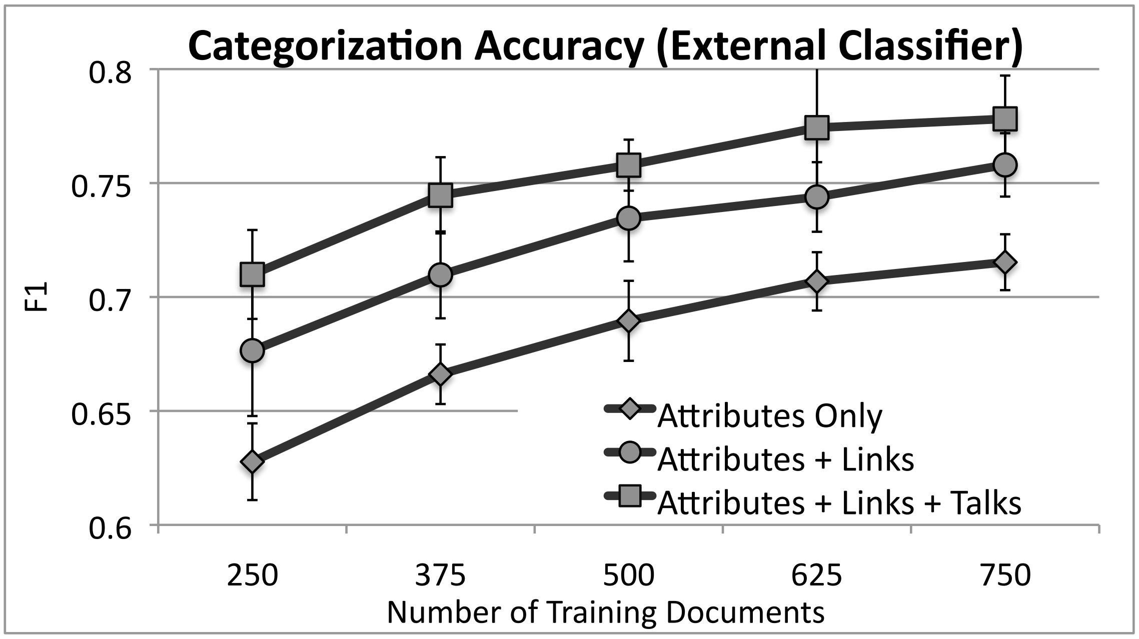


Figure 9.5: F1 score on classification against number of training documents.

page of the given document;[12] and HasCat(document, category), which states that the document has a particular category. We used the last two years of edits to the talk pages. To reduce noise, we discarded talks that were marked as "minor" by the users themselves or were authored by users with no user names, which typically correspond to automated bots or instances of vandalism. The dataset is available at http://psl.umiacs.umd.edu.

We applied PSL to two distinct tasks in this data set. First, to verify that PSL can handle common relational problems, we experimented with a collective classification setting, where relational information is used in an effort to improve over a classifier trained on the tf/idf-weighted word features on a holdout document set. The goal is to predict HasCat for each test document. The second task tests PSL's ability to reason about and propagate similarities. In this task, the text features of the documents are used only to compute a measure of similarity between any given pair of documents and are not used directly as features in a classifier. At test time, the categories of a small subset of the documents, called "seed documents," are observed, and the goal is to *propagate* these assignments to unlabeled documents based on the similarity between them and the relationships in which they participate. The accuracy of the inferred similarities is evaluated through the correctness of the category assignments of the unobserved documents, inferred through the HasCat relation, and thus, on the surface this task may seem almost identical to collective classification. However, we emphasize that in the similarity propagation task the text of documents is not used directly but only to measure similarities between the documents; no model is trained on the textual features of the documents. In

[12]In Wikipedia, each page has an accompanying Talk page where editors discuss potential changes to the content.

other words, there is no assumption that the documents in the test set are from the same domain, or even the same language, as those in the training set. Thus, to perform well, our model needs to effectively propagate these similarities through the relational structure.

The methodology in both tasks is as follows. We randomly split the entire corpus of documents into two equal-sized sets A and B and remove all relations between documents in different sets. We use the data in A for training and then test on set B. The results we report in Figures 9.5 and 9.6 are averages over 16 independent runs, and the vertical error bars show the standard deviations.

For the collective classification task, we trained a Naive Bayes classifier over the text features of a randomly selected subset of X documents from A (Figure 9.5 will show results for varying X). The predictions of this classifier were provided as evidence through the ClassifyCat(wordFeatures, category) similarity function. We used the remaining documents in A for training the rule weights with the BFGS algorithm. We experimented with three sets of rules. The first set, **Attributes-Only**, is a baseline that uses only the following rule, which simply copies the predictions of the Naive Bayes classifier:

$$ClassifyCat(A, N) \tilde{\Rightarrow} HasCat(A, N)$$

The second set, **Attributes+Links:**, contains an additional rule stating that hyperlinked documents tend to have the same category:

$$hasCat(B, C) \tilde{\wedge} link(A, B) \tilde{\wedge} A \neq B \tilde{\Rightarrow} HasCat(A, C)$$

The third set **Attributes+Links+Talks** contains an additional rule stating that two documents talked about by the same user have the same category:

$$\begin{aligned} talk(D, A) \tilde{\wedge} talk(E, A) \tilde{\wedge} hasCat(E, C) \tilde{\wedge} \\ E \neq D \tilde{\Rightarrow} HasCat(D, C) \end{aligned}$$

Each set of rules includes an additional constraint which ensures that each document can have at most one category. To compute the precision, for each document we select the category to which it is most related, according to the propagated similarities, and compare that category to the ground truth. We note that no document categories were provided during testing, but that Naive Bayes training requires a significant number of labeled documents.

The F1 scores (computed as the harmonic mean between precision and recall) on collective classification are shown in Figure 9.5. All differences are statistically significant at $p = 0.01$. We observe that considering link relationships yields an average 6.5% improvement over the baseline, whereas link and talk relationships combined improve the baseline F1 score by an average of 10.6%. As expected, improvements are larger for smaller training corpus size where the base classifier is less accurate.

On the similarity propagation task, we randomly designate $X\%$ of the docu-

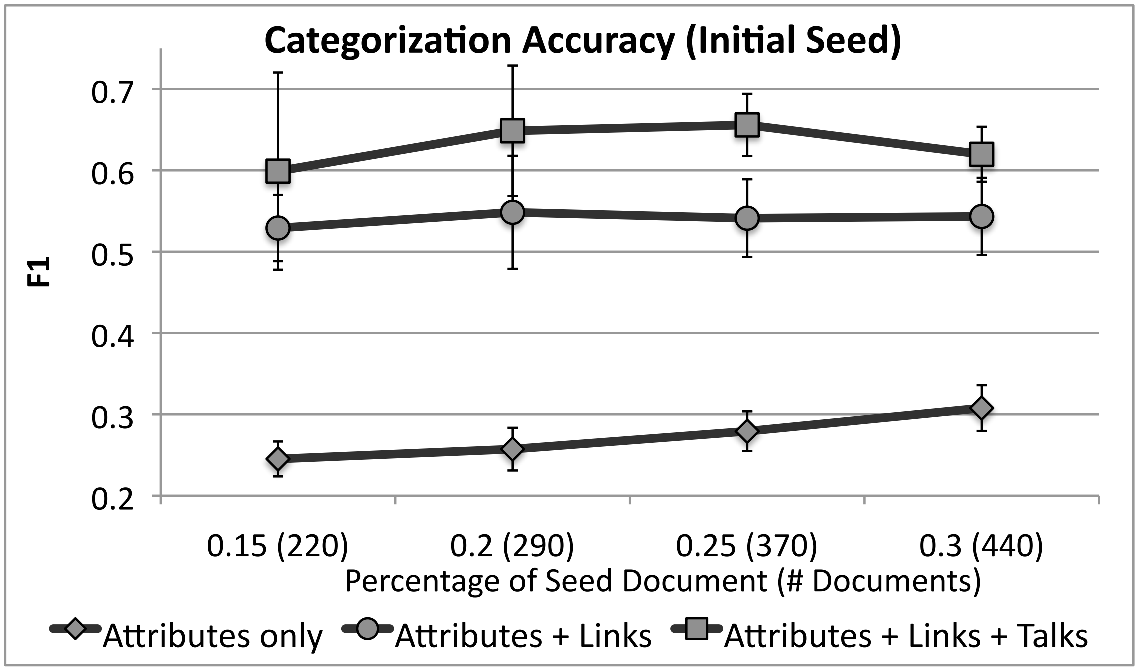


Figure 9.6: F1 score on category prediction against percentage of seed documents

ments in the train (A) and test (B) sets as seed documents and reveal their category during inference. As a baseline (**Attributes-Only**), we use a rule stating that documents with similar word vectors, as measured by cosine similarity, have the same category. As before, we used two additional rule sets by extending the baseline with rules concerning link and talk relationships (**Attributes+Links** and **Attributes+Links+Talks** respectively). We emphasize that in contrast to the first task, here we do not use the words as features in the model but only to compute similarities between documents. Figure 9.6 shows the average F1 scores for varying percentage of seed documents. All observed differences are significant at $p = 0.01$, except for the two left-most points, where the significance is at $p = 0.02$. We observe that propagating category assignments via link and talk relationships yields a huge improvement over the attribute similarity baseline.

These results demonstrate that the relational structure in the Wikipedia data set is helpful and that PSL can effectively exploit it to model both tasks.

9.8.2 Ontology Alignment

Ontology alignment, and information integration tasks in general, are promising application areas for PSL. An ontology is a formal specification of a set of concepts and the different relationships that exist among them, usually forming a concept hierarchy. The goal of ontology alignment is, given two ontologies O_1 and O_2 that may use different vocabularies to describe the same, or similar, concepts, to find a matching between the concepts and relationships in O_1 and O_2, e.g., [33, 45].

Because frequently no exact match exists, one needs to reason about degrees

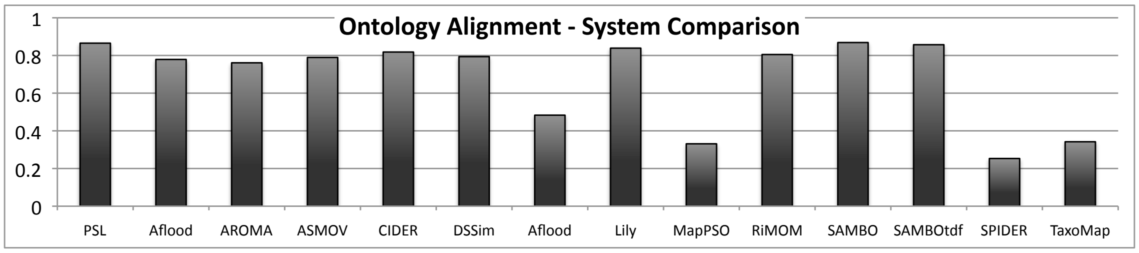


Figure 9.7: F1 Measure comparison of different ontology alignment systems on real bibliographic ontologies

of similarity between concepts and relations, while at the same time incorporating this reasoning into a relational framework. For example, one can exploit regularities such as that two concepts are similar if their sub-concepts or parent concepts are similar.

Ontology alignment has received growing attention in recent years, in part due to the explosion of interest in web services, information exchange over the web in general and the semantic web in particular. A large number of approaches have been proposed (see [33] and [45] for surveys). Ontology alignment is a particularly challenging problem due to the complexities of ontologies themselves. Ontologies define concepts, relations, and objects and a host of possible relationships between those basic entities. In addition, ontologies have an associated semantics which constrains feasible alignments to ensure consistency.

Using the general PSL framework, we designed a set of 21 rules and constraints expressing our understanding for how similarity propagates within ontologies. Some of these rules are hard rules, like a rule stating that one concept from ontology O_1 can be equivalent to at most one concept in ontology O_2, that ensure the consistency of a computed alignment. The majority of the rules are soft-weighted rules, like rules stating that concepts are equivalent if their names or their parents are similar. For example:

$$\begin{aligned} type(A, concepts) \quad & \tilde{\wedge}\; type(B, concepts) \;\tilde{\wedge}\; name(A, X) \\ & \tilde{\wedge}\; name(B, Y) \;\tilde{\wedge}\; similarID(X, Y) \\ & \tilde{\wedge}\; A.source \neq B.source \\ & \tilde{\Rightarrow}\; similar(A, B) \end{aligned}$$

states that two concepts A, B with similar names defined in different source ontologies are likely to be similar. $similarID$ is a similarity function implemented using a modified Levenshtein metric that detects camel-case notation for word separation. If two concepts align, then it is likely that their respective sets of sub-concepts align as well, which we capture in the following rule using the set equivalence operator $s_{\{\}}$ defined in Section 9.2.

$$\begin{gathered} type(A, concepts) \;\tilde{\wedge}\; type(B, concepts) \;\tilde{\wedge}\; similar(A, B) \\ \tilde{\wedge}\; A! = B \;\;\tilde{\Rightarrow}\; \{A.subclassOf\} \overset{s_{\{\}}}{=} \{B.subclassOf\} \end{gathered}$$

Several rules consider attribute similarity as source of evidence while the remaining rules focus on equivalences of related entities, such as sub-concepts, superconcepts, incident relations, and others. The full set of rules is included in the Appendix 9.10. We extended standard string similarity measures, such as Levenshtein and Dice similarity, to measure the similarity between attributes. Given a pair of ontologies, we convert each ontology into a knowledge base of ground atoms and load it into the database.

We do not claim that 21 rules suffice to capture all intricacies of ontologies, but catering toward any of them is as easy as adding more rules or similarity measures. The ease with which rules can be modified and similarity functions integrated into PSL allows model designers to quickly evaluate their intuitions by testing different rules and similarity functions with little implementation effort.

To evaluate the performance of our set of PSL rules, we conducted an experimental study using the OAEI benchmark [26]. The Ontology Alignment Evaluation Initiative (OAEI) invites researchers to compare their ontology alignment systems on a fixed set of benchmark ontology pairs for which reference alignments are provided[13]. We used the same set of rules for all ontology pairs and compared the reference alignment to the alignment inferred by PSL. Since the reference alignments provided by OAEI declare equivalences to be either true or false, we used a threshold of 0.5 on the inferred similarities.

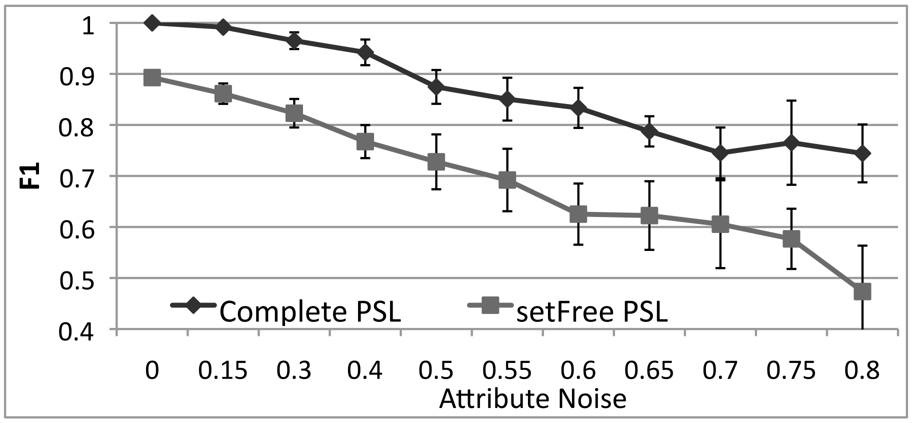
Figure 9.7 compares the F1 score of PSL against the reported scores of other systems that participated in the evaluation initiative [26] on the real-world ontologies included in the benchmark (300 level). The ontologies used in our evaluation contained approximately 100 entities (concepts, properties) each. We use one ontology pair for training rule weights and then test on the 3 remaining pairs; we report average F1 over all possible test ontology pairs. For weight learning, we used the Perceptron algorithm. Our PSL model obtains an F1 score of 0.865, which shows that PSL can learn accurate weights from a single pair of ontologies and generalize to the remaining pairs. We observe that using only a small set of PSL rules, we achieve alignment results that are comparable to the leading ontology matching systems which have been subject to considerable research and implementation effort.

We also experimented with a methodology closer to the one used in the OAEI initiative by manually fixing weights and testing on all ontology pairs. The results were almost identical to the ones in Fig. 9.7. However, because the weights were informed by our observations on the learned weights from above, we consider these results "contaminated."

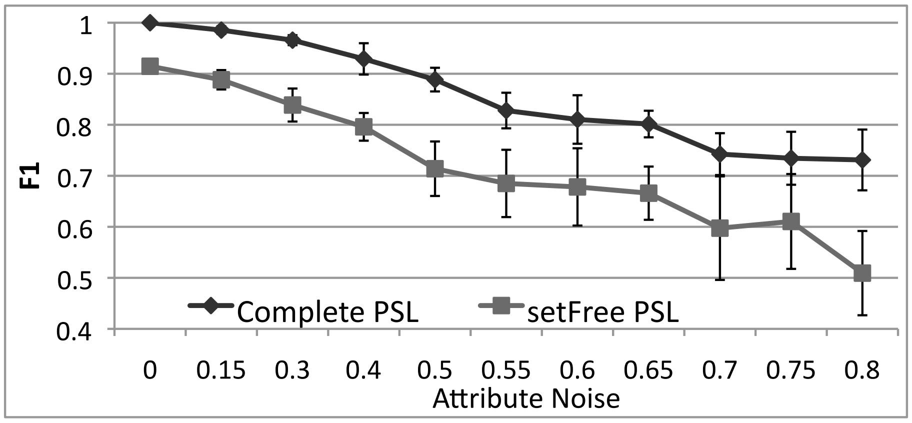
9.8.2.1 Utility of Sets

In this section we quantify the utility of sets for probabilistic relational reasoning on the ontology alignment task. For this purpose, we explored the behavior of the complete and setFree versions of our ontology alignment PSL program on the 20X ontologies of the OAEI benchmark, which contains ontology pairs with randomly inserted attribute and structural noise. We tried to replicate this suite of benchmark ontologies such that we can control the level of noise. For a given

[13]`http://oaei.ontologymatching.org/2008/`



a) 0.2 Structural Noise



b) 0.4 Structural Noise

Figure 9.8: F1 comparison of complete- and setFree PSL on ontology alignment with varying structural and attribute noise

level of attribute noise a and structural noise s, we replace a fraction of a attributes with random strings and remove a fraction of c relationships from the ontology. The results closely match the 20X ontology pairs.

The PSL rules we used for ontology alignment above contain set constructs. For instance, one rule states that if two concepts are similar, then the sets of their respective children overlap. To contrast standard PSL to setFree-PSL we created a second set of rules in which all rules using sets were replaced by their setFree counterparts using the conversion scheme outlined in Section 9.3. We learned rule weights on one generated pair of ontologies and then tested on a different, independently generated pair with the same noise levels. The results are averages over 10 independent trials. Figure 9.8 compares the results for two levels of structural noise, 0.2 in a) and 0.4 in b), and with attribute noise varying from 0 to 0.8. All differences are statistically significant at $p = 0.01$. We observe that complete PSL consistently outperforms the setFree version, yielding improvements from 9.3% to 57% as attribute noise increases.

9.8.3 Similarity and Scalability

Lastly, we discuss the similarity aspect of PSL in the context of the presented experiments. For one, having a semantics centered around similarity allows PSL to easily incorporate a wide range of similarity measures. In our Wikipedia experiments, we integrated cosine similarity and an existing implementation of Naive Bayes. For ontology alignment, we used previously proposed string similarity measures such as Levenshtein, Dice, and others.

However, because our gold standard evaluation data did not contain similarity values but was in terms of hard truth statements, we were unable to evaluate the quality of the similarity values inferred by PSL. Instead, to achieve comparability to the data at hand, we used similarity post-processing, e.g., in ontology alignment we treated two concepts as exactly aligned if their similarity was greater than 0.5.

This raises the question of whether a discrete formulation of the problem within PSL would lead to better performance. To answer this question, we implemented a discrete version of PSL called 0-1 PSL based on mixed integer conic programming which requires all result atoms to be either 0 or 1, i.e. *true* or *false*.

We repeated the Wikipedia category prediction experiments using 0-1 PSL[14]. The measured performance of 0-1 PSL was equal to the continuous formulation (i.e. differences were statistically insignificant) on the similarity propagation task and slightly worse compared to standard PSL (at $p = 0.02$) on collective classification.

While the performance was virtually identical, PSL inference of the discrete formulation took significantly longer. On the similarity propagation task, PSL inference in the original continuous setting took an average of 83 seconds including data loading and preparation. Discrete inference took more than 11 times longer at an average of 974 seconds. Similarly, for collective classification, PSL inference took 18.5 minutes for a complete run including classifier training, whereas 0-1 PSL required almost an hour (54 minutes) on average. Since the ontology alignment task is much smaller, inference times were under 5 seconds with most of the time spend on parsing and data loading. These statistics also demonstrate the efficiency and scalability of standard PSL inference.

9.8.4 Marginal Inference Experiemnts

Now we turn to the empirical evaluation of the proposed sampling algorithm for marginal inference on the problem of category prediction for Wikipedia documents based on similarity as discussed above. We demonstrate that the computed marginal distributions effectively predict document categories. Moreover, we show that analysis of the marginal distribution provides an indicator for the confidence in those predictions. Finally, we investigate the convergence rate and runtime performance of the algorithm in detail.

For our evaluation dataset, we considered a subset of 1717 documents Wikipedia documents assigned to the 7 most popular categories. After stemming and stop-word

[14]For ontology alignment, we were unable to tune the discrete solver to find an optimal solution, possibly due to the more complex relational structure and wide usage of set constructs.

removal, we represented the text of each document as a tf/idf-weighted word vector. To measure the similarity between documents, we used the popular cosine metric on the weighted word vectors. The data contains the relations `Link(fromDoc, toDoc)`, which establishes a hyperlink between two documents. We used K-fold cross-validation for $k = 20, 25, 30, 35$ by splitting the dataset into K non-overlapping subsets each of which is determined using snowball sampling over the link structure from a randomly chosen initial document. For each training and test data subset, we randomly designate 20% of the documents as "seed documents" of which the category is observed and the goal is to predict the categories of the remaining documents.

9.8.4.1 Classification results

K	Baseline	Marginals	Improvement
20	39.5%	55.8%	41.4%
25	39.1%	51.5%	31.7%
30	36.7%	51.1%	39.1%
35	38.8%	56.6%	46.1%

a) Classification Accuracy

K	$\mathbb{P}$(Null Hypothesis)	Relative Difference $\Delta(\sigma)$
20	1.95E-09	38.3%
25	2.40E-13	41.2%
30	<1.00E-16	43.5%
35	4.54E-08	39.0%

b) Std. deviation as an indicator for confidence

Figure 9.9: Wikipedia category predication results

The baseline method uses only the document content by propagating document categories via textual similarity measured by the cosine distance. Using rules and constraints similar to those presented in Table 9.1, we create a joint probabilistic model for collective classification of Wikipedia documents. We use PSL twofold in this process: Firstly, PSL constructs the CCMRF by grounding the rules and constraints against the given data and secondly, we use the perceptron weight learning method provided by PSL to learn the free parameters of the CCMRF from the training data. The sampling algorithm takes the constructed CCMRF and learned parameters as input and computes the marginal distributions for all random variables from 3 million samples. We have one random variable to represent the similarity for each possible document-category pair, that is, one RV for each grounding of the `category` predicate. For each document D we pick the category C with the highest expected similarity as our prediction. The accuracy in prediction of both methods is compared in Table 9.8.4.1 a) over the 4 different splits of the data. We observe that the collective probabilistic model outperforms the baseline by up to 46%. All results are statistically significant at $p = 0.02$.

While this results suggests that the sampling algorithm works in practice, it is not surprising and novel since similar results were achieved using MAP inference on the same dataset in Section 9.8.1. However, the marginal distributions we obtain provide additional information beyond the simple point estimate of its expected value. In particular, we show that the standard deviation of the marginals can serve as an indicator for the confidence in the particular classification prediction. In order to show this, we compute the standard deviation of the marginal distributions for those random variables picked during the prediction stage for each fold. We separate those values into two sets, S_+, S_-, based on whether the prediction turned out to be correct $(+)$ or incorrect $(-)$ when evaluated against the ground truth. Let σ_+, σ_- denote the average standard deviation for those values in S_+, S_- respectively. Our hypothesis is that we have higher confidence in the correct predictions, that is, σ_+ will typically be smaller than σ_-. In other words, we hypothesize that the relative difference between the average deviations, $\Delta(\sigma) = 2\frac{\sigma_- - \sigma_+}{\sigma_+ + \sigma_-}$, is larger than 0. Under the corresponding null hypothesis, we would expect any difference in average standard deviation, and therefore any nonzero $\Delta(\sigma)$, to be purely coincidental or noise. Assuming that such noise in the $\Delta(\sigma)$'s, which we computed for each fold, can be approximated by a Gaussian distribution with 0 mean and unknown variance[15], we test the null hypothesis using a two tailed Z-test with the observed sample variance. The Z-test scores on the 4 differently sized splits are reported in Table 9.8.4.1 b) and allow us to reject the null hypothesis with very high confidence. Table 9.8.4.1 b) also lists $\Delta(\sigma)$ for each split averaged across the multiple folds and shows that σ_- is about 40% larger than σ_+ on average.

9.8.4.2 Marginal Inferenece Algorithm performance

In investigating the performance of the sampling algorithm we are mainly interested in two questions: 1) How many samples does it take to converge on the marginal density functions? and 2) What is the computational cost of sampling? To answer the first question, we collect independent samples of varying size from 31,000 to 2 million and one reference sample with 3 million steps for all folds. For each of the former samples we compare the marginals thus obtained to the ones of the reference sample by measuring their KL divergence. To compute the KL divergence we discretize the density function using a histogram with 10 bins. The center line in Figure 9.10 shows the average KL divergence with respect to the sample size across all folds. To study the impact of dimensionality on convergence, we order the folds by the number of random variables n and show the average KL divergence for the lowest and highest quartile which contain $322 - 413$ and $174 - 224$ random

[15]Even if the standard deviations in S_+, S_- are not normally distributed, the central limit theorem postulates that their averages will eventually follow a normal distributions under independence assumptions.

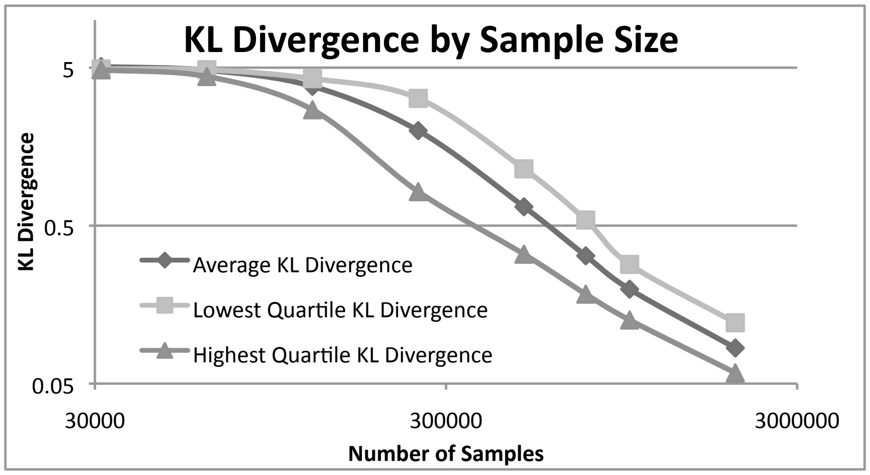


Figure 9.10: KL Divergence by sample size

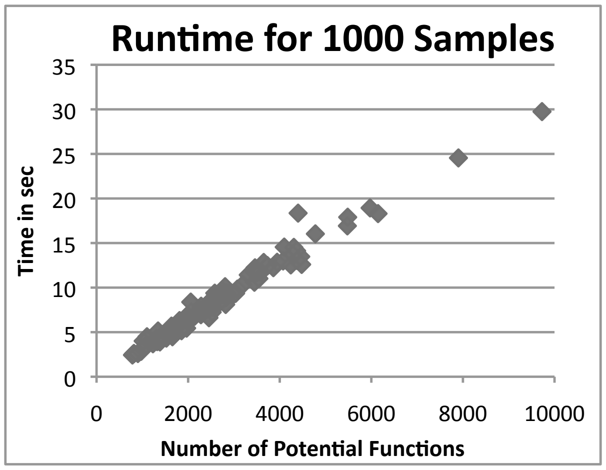


Figure 9.11: Runtime for 1000 samples

variables respectively. The plot is drawn in log-log scale and therefore suggests that each magnitude increase in sample size yields a magnitude improvement in KL divergence. To answer the second question, Figure 9.11 displays the time needed to generate 1000 samples with respect to the number of potential functions in the CCMRF. Computing the induced probability density function along the sampled line segment dominates the cost of each sampling step and the graph shows that this cost grows linearly with the number of potential functions.

9.9 Related Work

PSL builds upon a large body of research in SRL, in which relational structure is parametrized in order to define a probabilistic graphical model over the properties and relations of the entities in a domain, e.g., BLPs [78], PRMs [56], RMNs [147], MLNs [125]. Like these models, PSL also supports probabilistic reasoning over relational structure. However, unlike previous work, PSL additionally supports reasoning about similarities of entities, or sets of entities, and integrates these capabilities into a unified framework. In terms of expressivity, PSL is closest to Hybrid MLNs [153], which allow the use of numeric-valued predicates. However, because Hybrid MLNs were not specifically designed for use with similarities, they do not include support for reasoning about set similarity, and reasoning in them is intractable in general. PSL is related to imperative frameworks, such as, IDFs [98] that use a programming language to define structural dependencies. While being very general, such frameworks are more complex to use and require implementation by the user, such as providing the MCMC sampler with custom-made proposal functions for each application studied. With PSL, a user only specifies the model – inference and learning do not require user input. PSL is also related to approaches, such as kFoil [87], that base similarity computation with kernel functions on relational structure. While also treating similarities as distances, kFoil addresses a different problem from the one studied here, by assuming that instances are independent. In contrast, PSL not only uses relational structure as features, but also to propagate similarities.

The idea of probability of satisfaction of logical formulas was introduced in 1964 in [52] and later studied in seminal papers such as [135] and many other subsequent papers in the last 45 years. Our notion of distance from satisfaction is a variant of such efforts. We also point to the large body of work in probabilistic (e.g. [113]) and fuzzy logic programs (e.g. [42]) which shares conceptual similarities with PSL, in that these approaches combine logical formulas with probabilitic or similarity measures. However, PSL uses a very different probabilistic model which captures cyclic dependencies, handles inconsistencies, and enforces domain constraints. The most important distinction, however, is the fact our formulation of PSL allows rules weights to be efficiently learned from ground truth data as described in Section 9.6.

For marginal computation, non-parametric belief propagation (NBP) [146] has been proposed as a method to estimate marginals for general continuous MRFs. NBP represents the "belief" as a combination of kernel densities which are propagated according to the structure of the MRF. In contrast to NBP, our approach provides polynomial-time approximation guarantees and avoids the representational choice of kernel densities.

9.10 Appendix

In the following we will list some of the PSL rule sets we have used in our experiments as reported in Section 9.8 for the interested reader. We hope this gives the reader a better understanding for how we modeled the individual problems and how a user might apply the PSL framework to different problems.

The syntax closely follows the one used in this chapter and only deviates where the limitations of the ASCII character set requires so. The syntax used below closely mirrors the syntax used in the implementation of PSL available at http://psl.umiacs.umd.edu. Note, that the example rules given in section 9.8 are taken from the rule sets below but might have abbreviated names for predicates and constants.

```
unknown(document:Entity)
known(document:Entity)
document(document:Entity,words:Attribute)
category(category:Entity,name:Attribute)
link(startDoc:Entity,endDoc:Entity)
talk(user:Entity,Document:Entity)
#hasCat = hasCategory
hasCat(document:Entity,category:Entity)

#Similarity Function implementing the respective java interface.
#In our implementation, classifierCat returns the probability,
#as given by the Naive Bayes classifier trained on the holdout set,
#that the given word vector is classified by the given category name
classifierCat(words:Attribute,categoryName:Attribute)




#### RULES ####

(classifyCat(A,N) & category(C,N) ) >> hasCat(A,C)

(hasCat(B,C) & link(A,B) & A!=B ) >> hasCat(A,C)

(talk(D,A) & talk(E,A) & hasCat(E,C) & A!=B) >> hasCat(D,C)


PartialFunctionalConstraint on hasCat(A,C)

Prior on hasCat(A,C)
```

Figure 9.12: Wikipedia category prediction with naive bayes classifier as similarity function.

```
unknown(document:Entity)
known(document:Entity)
document(document:Entity,words:Attribute)
category(category:Entity,name:Attribute)
link(startDoc:Entity,endDoc:Entity)
talk(user:Entity,Document:Entity)
#hasCat = hasCategory
hasCat(document:Entity,category:Entity)

#Similarity Function implementing the respective java interface.
#In our implementation, similarText uses cosine similarity between
#the word vectors to compute the similarity score.
similarText(words1:Attribute, words2: Attribute)


#### RULES ####

(hasCat(B,C) & A!=B & unknown(A) & document(A,T) & document(B,U)
& similarText(T,U)) >> hasCat(A,C)

(hasCat(B,C) & unknown(A) & link(A,B) & A!=B ) >> hasCat(A,C)

(talk(D,A) & talk(E,A) & hasCat(E,C) & unkonwn(D) & A!=B) >> hasCat(D,C)

PartialFunctionalConstraint on hasCat(A,C)

Prior on hasCat(A,C)
```

Figure 9.13: Wikipedia category prediction with similarity propagation using cosine similarity.

```
name(object:Entity, name: Attribute)
fromOntology(object:Entity, ontology : Entity)
comment(object:Entity, comment : Attribute)
label(object:Entity, label: Attribute)
subClassOf(child:Entity, parent: Entity)
subPropertyOf(child:Entity, parent: Entity)
rangeOf(property:Entity, class:Entity)
domainOf(property:Entity, class:Entity)
entityType(object:Entity, type:Entity)
propertyType(property:Entity, propType: Entity)
exactCardRestriction(property:Entity, class: Entity)
minCardRestriction(property:Entity, class: Entity)
maxCardRestriction(property:Entity, class: Entity)
equivalent(objet:Entity, object:Entity)


#The following are string similarity functions that we used
#in ontology alignment. They are all implemented extending
#the PSL provided interface. These functions are mostly
#based off of existing string similarity measures provided
#by the string similarity measure library for Java called
#SimMetric which can be downloaded from:
#http://www.dcs.shef.ac.uk/~sam/stringmetrics.html

subString(s1:Attribute, s2: Attribute)
sameID(s1:Attribute, s2: Attribute)
similarID(s1:Attribute, s2: Attribute)
sameString(s1:Attribute, s2: Attribute)
sameText(s1:Attribute, s2: Attribute)


entityType(A,property) & entityType(B,property) & name(A,X)
& name(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& sameID(X,Y) >> equivalent(A,B)

entityType(A,class) & entityType(B,class) & name(A,X)
& name(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& sameID(X,Y) >> equivalent(A,B)

entityType(A,property) & entityType(B,property) & name(A,X)
& name(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& similarID(X,Y) >> equivalent(A,B)

entityType(A,class) & entityType(B,class) & name(A,X)
& name(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& similarID(X,Y) >> equivalent(A,B)
```

```
entityType(A,T) & entityType(B,T) & label(A,X)
& label(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& sameText(X,Y) >> equivalent(A,B)

entityType(A,T) & entityType(B,T) & comment(A,X)
& comment(B,Y) & fromOntology(B,Q) & fromOntology(A,O) & Q!=O
& sameText(X,Y) >> equivalent(A,B)

# The 'inv' modifier denotes the inverse of a given relation. Hence
# subclassOf(inv) is the inverse of the subclassOf relation.

entityType(A,class) & entityType(B,class) & equivalent(A,B) & A!=B
>> equivalentSet( {A.subclassOf(inv)} , {B.subclassOf(inv)} )
#Where 'equivalentSet' is the adapted jaccard similarity between sets based on
#the equivalent predicate as defined in the description:

setcomparison: equivalentSet, using: SetComparison.Equality

entityType(A,class) & entityType(B,class) & equivalent(A,B) & A!=B
>> equivalentSet( {A.subclassOf} , {B.subclassOf} )

entityType(A,class) & entityType(B,class) & equivalent(A,B) & A!=B
>> equivalentSet( {A.domainOf(inv) + A.exactCardRestriction(inv) +
A.maxCardRestriction(inv) + A.minCardRestriction(inv) , {B.domainOf(inv) +
B.exactCardRestriction(inv) + B.maxCardRestriction(inv) +
B.minCardRestriction(inv)} )

entityType(A,property) & entityType(B,property) & equivalent(A,B) & A!=B
>> equivalentSet( {A.domainOf + A.exactCardRestriction + A.maxCardRestriction
+ A.minCardRestriction}, {B.domainOf +
B.exactCardRestriction + B.maxCardRestriction + B.minCardRestriction} )

entityType(A,property) & entityType(B,property) & equivalent(A,B) & A!=B
>> equivalentSet ({A.subPropertyOf(inv)} , {B.subPropertyOf(inv)} )

entityType(A,property) & entityType(B,property) & equivalent(A,B) & A!=B
>> equivalentSet ({A.subPropertyOf} , {B.subPropertyOf} )


Prior on equivalent(A,B)

PartialFunctionalConstraint on equivalent(A,B)
PartialInverseFunctionConstraint on equivalent(A,B)
```

Figure 9.14: Ontology Alignment

Bibliography

[1] D. J Abadi, A. Marcus, S. R Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, page 411–422, 2007.

[2] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic Web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[3] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. *Workshop on Management of Semistructured Data*, 1997.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[5] S. Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6(3):382392, 1954.

[6] R. Albert and A. L Barabasi. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47–97, 2002.

[7] F. Alizadeh and D. Goldfarb. Second-order cone programming. *Mathematical Programming*, 95(1):3–51, 2003.

[8] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[9] Renzo Angles and Claudio Gutierrez. Querying RDF data from a graph database perspective. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications*, volume 3532, pages 346–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[10] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, volume 57, 1989.

[11] F. Benamara, C. Cesarano, A. Picariello, D. Reforgiato, and V.S. Subrahmanian. Sentiment analysis: Adverbs and adjectives are better than adjectives alone. In *Proc. 2007 Intl. Conf. on the Web and Social Media*, pages 203–206, March 2007.

[12] BerkeleyDB. *http://www.oracle.com/technology/products/berkeley-db.*

[13] V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008:P10008, 2008.

[14] S. Boag, D. Chamberlin, M. F Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. *W3C working draft*, 15, 2002.

[15] P. A Boncz. *Monet. A next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, Netherlands, 2002.

[16] Ulrik Brandes, Patrick Kenis, Jürgen Lerner, and Denise van Raaij. Network analysis of collaboration structure in Wikipedia. In *Proceedings of WWW-09*, 2009.

[17] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *International Semantic Web Conf.*, pages 97–113, 2009.

[18] Matthias Bröcheler, Andrea Pugliese, and V.S. Subrahmanian. COSI: cloud oriented subgraph identification in massive social networks. In *ASONAM*, 2010.

[19] M. Broecheler, L. Mihalkova, and L. Getoor. Probabilistic similarity logic. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[20] M. Broecheler, G. Simari, and VS. Subrahmanian. Using histograms to better answer queries to probabilistic logic programs. *Logic Programming*, page 4054, 2009.

[21] Matthias Broecheler and Lise Getoor. Computing marginal distributions over continuous markov networks for statistical relational learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2010.

[22] Matthias Broecheler, Andrea Pugliese, and V.S. Subrahmanian. Probabilistic Subgraph Matching on Huge Social Networks. In *Proceedings of 2011 International Conference on Advances in Social Network Analysis and Mining (ASONAM 2011)*, 2011.

[23] Matthias Broecheler, VS Subrahmanian, and Andrea Pugliese. A Budget-Based algorithm for efficient subgraph matching on huge networks. In *2nd International Workshop on Graph Data Management: Techniques and Applications*, 2011.

[24] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. In *Spinning the Semantic Web*, pages 197–222, 2003.

[25] Min Cai and Martin Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*, pages 650–657, New York, NY, USA, 2004. ACM.

[26] C. Caracciolo, J. Euzenat, L. Hollink, R. Ichise, A. Isaac, V. Malaise, C. Meilicke, J. Pane, P. Shvaiko, and H. Stuckenschmidt. First results of the ontology alignment evaluation initiative 2008. In *ISWC-2008 Workshop on Ontology Matching*, 2008.

[27] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS Symp.*, pages 34–43. ACM Press, 1998.

[28] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. Technical Report TR-DB-052006-CLJF, Wayne State University, 2006.

[29] James Cheng, Yiping Ke, and Wilfred Ng. Efficient query processing on graph databases. *ACM Trans. Database Syst.*, 34(1), 2009.

[30] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872, Beijing, China, 2007. ACM.

[31] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In *ICDE Conf.*, pages 913–922, 2008.

[32] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 219–228, 2009.

[33] N. Choi, I. Y. Song, and H. Han. A survey on ontology mapping. *ACM SIGMOD Record*, 35(3):34–41, 2006.

[34] C. W. Chung, J. K. Min, and K. Shim. APEX: an adaptive path index for XML data. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 121–132, 2002.

[35] Michael Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of EMNLP-02*, 2002.

[36] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A(sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[37] Luc De Raedt. *Logical and Relational Learning.* Springer Verlag, Berlin, 2008.

[38] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[39] K. Dimitrova, M. El-Sayed, and E. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. *Conceptual Modeling-ER 2003*, page 144–157, 2003.

[40] M. E. Dyer and A. M. Frieze. On the complexity of computing the volume of a polyhedron. *SIAM Journal on Computing*, 17(5):967–974, October 1988.

[41] N. Eagle, A. Pentland, and D. Lazer. Inferring social network structure using mobile phone data. *Proceedings, National Academy of Science*, 6(36):15274–15278.

[42] Rafee Ebrahim. Fuzzy logic programming. *Fuzzy Sets and Systems*, 117(2):215 – 230, 2001.

[43] O. Erling and I. Mikhailov. Towards Web-Scale RDF. *SSWS, Karlsruhe, Germany*, 2008.

[44] S. Eubank, H. Guclu, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429:180–184, 2004.

[45] J. Euzenat. *Ontology matching*. Springer, 2007.

[46] Y. Fa, Y. Chen, T. W Ling, and T. Chen. Materialized view maintenance for XML documents. *Proceedings of Web Information Systems 2004*, page 365–371, 2004.

[47] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM Conf.*, pages 251–262, 1999.

[48] A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, D. Skripin, and D. Shasha. GraphFind: enhancing graph searching by low support data mining techniques. *BMC Bioinformatics*, 9(4):S10, 2008.

[49] P.O. Fjallstrom. Algorithms for Graph Partitioning: A Survey. *Computer and Information Science*, 3(10), 1998.

[50] Flickr. *http://www.flickr.com.*

[51] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[52] H. Gaifman. Concerning measures in first order calculi. *Israel journal of mathematics*, 2(1), 1964.

[53] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[54] M. R Garey and D. S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co. New York, NY, USA, 1979.

[55] Johannes Gehrke, Paul Ginsparg, and Jon M. Kleinberg. Overview of the 2003 KDD cup. *SIGKDD Explorations*, 5(2), 2003.

[56] Lise Getoor, Nir Friedman, Daphne Koller, and Benjamin Taskar. Learning probabilistic models of link structure. *Journal of Machine Learning Research*, 3:679–707, 2002.

[57] Lise Getoor and Ben Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[58] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, June 2002.

[59] R. Giugno and D. Shasha. GraphGrep: a fast and universal method for querying graphs. In *International Conference on Pattern Recognition*, volume 16, pages 112–115, 2002.

[60] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR Conf.*, pages 112–115, 2002.

[61] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. *Proceedings of the International Conference on Very Large Databases*, pages 436–445, 1997.

[62] GovTrack dataset. *http://www.govtrack.us*.

[63] A. Gupta and I.S. Mumick (eds.). *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999.

[64] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Record*, volume 22, 1993.

[65] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.

[66] A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *LA-Web*, pages 71–80, 2005.

[67] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *ISWC*, pages 211–224, 2007.

[68] Huahai He. Closure-Tree: an index structure for graph queries. In *22nd International Conference on Data Engineering (ICDE'06)*, April 2006.

[69] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[70] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552, 2003.

[71] E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, page 717–728, 2005.

[72] Edward Hung, Yu Deng, and V. S. Subrahmanian. Rdf aggregate queries and views. In *ICDE*, pages 717–728, 2005.

[73] JenaTDB. *http://jena.hpl.hp.com/wiki/TDB.*

[74] R. Kannan, L. Lovasz, and M. Simonovits. Random walks and an o*(n5) volume algorithm for convex bodies. *Random structures and algorithms*, 11(1):150, 1997.

[75] D.R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[76] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1999.

[77] Yiping Ke, James Cheng, and Jeffrey Xu Yu. Querying large graph databases. In *DASFAA Conf.*, pages 487–488, 2010.

[78] K. Kersting and L. De Raedt. Bayesian logic programs. Technical report, Albert-Ludwigs University, 2001.

[79] M. Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications*. *The Journal of Logic Programming*, 12(4):335–367, 1992.

[80] R. Kindermann and J. L. Snell. *Markov random fields and their applications.* American Mathematical Society Providence, RI, 1980.

[81] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - a pragmatic semantic repository for OWL. In *WISE Workshops*, pages 182–192, 2005.

[82] Karsten Klein, Nils Kriege, and Petra Mutzel. CT-Index: fingerprint-based graph indexing combining cycles and trees. In *Proceedings of 27th IEEE International Conference on Data Engineering*, Hannover, Germany, 2011.

[83] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

[84] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications.* Prentice Hall, 1995.

[85] Bong-Jun Ko and D. Rubenstein. Distributed self-stabilizing placement of replicated resources in emerging networks. *Networking, IEEE/ACM Transactions on*, 13(3):476–487, 2005.

[86] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of ICDM 2001*, page 313. IEEE Computer Society, 2001.

[87] Niels Landwehr, Andrea Passerini, Luc De Raedt, and Paolo Frasconi. Fast learning of relational kernels. *Machine Learning*, 78:305–342, 2010.

[88] O. Lassila and R. Swick. *Resource Description Framework (RDF) model and syntax specification.* W3C, 1998.

[89] Chulki Lee, Sungchan Park, Dongjoo Lee, Jae-Won Lee, Ok-Ran Jeong, and Sang-Goo Lee. A comparison of ontology reasoning systems using query sequences. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 543–546, Suwon, Korea, 2008. ACM.

[90] H. Liefke and S. Davidson. View maintenance for hierarchical semistructured data. *Data Warehousing and Knowledge Discovery*, page 114–123, 2000.

[91] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret. Applications of second-order cone programming. *Linear Algebra and Its Applications*, 284(1-3):193–228, 1998.

[92] L. Lovasz and S. Vempala. Hit-and-run from a corner. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 310–314, Chicago, IL, USA, 2004. ACM.

[93] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 484–491, Washington, D.C., USA, 2004. ACM.

[94] Li Ma, Chen Wang, Jing Lu, Feng Cao, Yue Pan, and Yong Yu. Effective and efficient semantic web data management over DB2. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1183–1194, Vancouver, Canada, 2008. ACM.

[95] K. Madduri and D. A Bader. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *IEEE International Symposium on Parallel and Distributed Processing, 2009*, 2009.

[96] V. Mahajan, E. Muller, and Y. Wind. *New-product diffusion models*. Kluwer Academic Publishers, 2000.

[97] G. Manjunath, R. Badrinath, and C. Sayers. Temporal views over RDF data. In *WWW*, 2008.

[98] Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Proceedings of NIPS-09*, 2009.

[99] M. Mehta and D. J DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB journal*, 6(1):5372, 1997.

[100] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE transactions on knowledge and data engineering*, 12(2):307–323, 2000.

[101] Tova Milo and Dan Suciu. Index structures for path expressions. In *Database Theory — ICDT'99*, pages 277–295, 1999.

[102] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM New York, NY, USA, 2007.

[103] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.

[104] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Conf.*, pages 29–42, 2007.

[105] T. S. Motzkin and I. J. Schoenberg. The relaxation method for linear inequalities. *IJ Schoenberg: Selected Papers*, page 75, 1988.

[106] Richard Myers, Richard C. Wilson, and Edwin R. Hancock. Bayesian graph edit distance. *IEEE TPAMI*, 22(6):628–635, 2000.

[107] Raffaele Di Natale, Alfredo Ferro, Rosalba Giugno, Misael Mongiovì, Alfredo Pulvirenti, and Dennis Shasha. SING: Subgraph search in non-homogeneous graphs. *BMC Bioinformatics*, 11:96, 2010.

[108] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, page 419–432, New York, NY, USA, 2008. ACM. ACM ID: 1376661.

[109] Y. Nesterov and A. Nemirovsky. Interior point polynomial methods in convex programming. *Studies in applied mathematics*, 13, 1994.

[110] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment archive*, 1(1):647–659, 2008.

[111] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the 35th SIGMOD international conference on Management of data*, page 627–640, 2009.

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

[113] Raymond Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150 – 201, 1992.

[114] N.J. Nilsson. *Principles of artificial intelligence.* Tioga, 1980.

[115] Jorge Nocedal and Stephen Wright. *Numerical Optimization.* Springer, 2nd edition, 2006.

[116] M. Onizuka, F. Y Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *Proceedings of the 14th international conference on World Wide Web*, page 671–681, 2005.

[117] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning using linear programming. In *SC*, pages 458–467, 1994.

[118] A. Owens, A. Seaborne, and N. Gibbins. Clustered TDB: a clustered triple store for jena. 2008.

[119] G. Palshikar and M. Ape. Collusion set detection using graph clustering. *Data Mining and Knowledge Discovery*, 16:135–164, 2008.

[120] Mike Personick. Bigdata: Approaching web scale for the semantic web, 2009.

[121] M. A Porter, J. P Onnela, and P. J Mucha. Communities in networks. *ArXiv*, 902, 2009.

[122] PostgreSQL. *http://www.postgresql.org.*

[123] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pages 524–538, 2008.

[124] D. Reforgiato and V.S. Subrahmanian. Ava: Adjective verb adverb combinations for sentiment analysis. *IEEE Intelligent Systems*, 23(4):43–50, July/Aug 2008.

[125] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.

[126] S. Riedel. Improving the accuracy and efficiency of MAP inference for Markov logic. In *Proceedings of UAI-08*, 2008.

[127] M.J. Riezenman. Cellular security: better, but foes still lurk. *Spectrum, IEEE*, 37(6):39 –42, June 2000.

[128] Mark Roantree, Colm Noonan, and John Murphy. Specifying and optimising XML views. In *Data Management. Data, Data Everywhere*, 2007.

[129] M. A Rodriguez and P. Neubauer. The graph traversal pattern. *Arxiv preprint arXiv:1004.1001*, 2010.

[130] R. Ronen and O. Shmueli. SoQL: a language for querying and creating data in social networks. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, page 1595–1602, 2009.

[131] S. Sakr and G. Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6(2):101–120, 2010.

[132] Sherif Sakr. GraphREL: A decomposition-based and selectivity-aware relational framework for processing sub-graph queries. In *DASFAA Conf.*, pages 123–137, 2009.

[133] A. Sanfeliu and K. S. Fu. Distance measure between attributed relational graphs for pattern recognition. *IEEE Trans. Sys. Man Cyber.*, 13(3):353–362, 1983.

[134] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S Candan. Incremental maintenance of path-expression views. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, page 443–454, 2005.

[135] D. Scott and P. Krauss. Assigning probabilities to logical formulas. *Studies in Logic and the Foundations of Mathematics*, 43, 1966.

[136] Andy Seaborne and Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, January 2008.

[137] Sesame2. *http://www.openrdf.org.*

[138] Michael Sintek and Malte Kiesel. RDFBroker: A signature-based high-performance RDF store. In *ESWC*, pages 363–377, 2006.

[139] R. L. Smith. Efficient monte carlo procedures for generating points uniformly distributed over bounded regions. *Operations Research*, 32(6):1296–1308, 1984.

[140] Slawek Staworko, Iovka Boneva, and Benoît Groz. The view update problem for XML. In *EDBT*, 2010.

[141] S. Stephens, J. Rung, and X. Lopez. Graph data representation in oracle databese 10g: Case studies in life sciences. *IEEE Data Engineering Bulletin*, 27(4):61–66, 2004.

[142] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web*, pages 595–604, Beijing, China, 2008. ACM.

[143] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW Conf.*, pages 595–604, 2008.

[144] M. Stonebraker, D. J Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, page 553–564, 2005.

[145] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *WWW*, pages 631–639, 2004.

[146] E. B Sudderth. *Graphical models for visual object recognition and tracking.* Ph.D. thesis, Massachusetts Institute of Technology, 2006.

[147] B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Proceedings of UAI-02*, 2002.

[148] The Lehigh University Benchmark. *http://swat.cse.lehigh.edu/projects/lubm.*

[149] Y. Tian, R. C. McEachin, and C. Santos. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232, 2007.

[150] Yuanyuan Tian and Jignesh M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[151] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[152] R. Volz, S. Staab, and B. Motik. Incremental maintenance of materialized ontologies. In *CoopIS/DOA/ODBASE*, 2003.

[153] Jue Wang and Pedro Domingos. Hybrid Markov logic networks. In *Proceedings of AAAI-08*, 2008.

[154] D. J. Watts and P. S. Dodds. Influentials, networks, and public opinion formation. *Journal of Consumer Research*, 34(4):441–458, 2007.

[155] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008—1019, 2008.

[156] R. Wetzker, C. Zimmermann, and C. Bauckhage. Analyzing social bookmarking systems: A del. icio. us cookbook. In *Mining Social Data Work.*, pages 26–30, 2008.

[157] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[158] R. Wilson and E. Hancock. Bayesian compatibility model for graph matching. *Pattern Recognition Letters*, 17:263–276, 1996.

[159] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, page 721–724, 2002.

[160] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346, Paris, France, 2004. ACM.

[161] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.

[162] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD Conf.*, pages 766–777, 2005.

[163] S. B Zdonik and D. Maier. *Readings in object-oriented database systems.* Morgan Kaufmann Pub, 1990.

[164] Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT Conf.*, pages 192–203, 2009.

[165] Shijie Zhang, Shirong Li, and Jiong Yang. SUMMA: subgraph matching in massive graphs. In *CIKM Conf.*, pages 1285–1288, 2010.

[166] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

[167] Ke Zhu, Ying Zhang, Xuemin Lin, Gaoping Zhu, and Wei Wang 0011. NOVA: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In *DASFAA Conf.*, pages 140–154, 2010.

[168] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.

[169] L. Zou, L. Chen, J. X Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, page 181–192, 2008.

[170] Lei Zou, Lei Chen, and M. Tamer Özsu. Distancejoin: Pattern match query in a large graph database. *VLDB Conf.*, 2(1):886–897, 2009.