ABSTRACT

Title of dissertation: TOP-K QUERY PROCESSING IN EDGE-LABELED GRAPH DATA

Noseong Park, Doctor of Philosophy, 2016

Dissertation directed by: Professor V.S. Subrahmanian
Department of Computer Science

Edge-labeled graphs have proliferated rapidly over the last decade due to the increased popularity of social networks and the Semantic Web. In social networks, relationships between people are represented by edges and each edge is labeled with a semantic annotation. Hence, a huge single graph can express many different relationships between entities. The Semantic Web represents each single fragment of knowledge as a triple (subject, predicate, object), which is conceptually identical to an edge from subject to object labeled with predicates. A set of triples constitutes an edge-labeled graph on which knowledge inference is performed.

Subgraph matching has been extensively used as a query language for patterns in the context of edge-labeled graphs. For example, in social networks, users can specify a subgraph matching query to find all people that have certain neighborhood relationships. Heavily used fragments of the SPARQL query language for the Semantic Web and graph queries of other graph DBMS can also be viewed as subgraph matching over large graphs.

Though subgraph matching has been extensively studied as a query paradigm

in the Semantic Web and in social networks, a user can get a large number of answers in response to a query. These answers can be shown to the user in accordance with an importance ranking. In this thesis proposal, we present four different scoring models along with scalable algorithms to find the top-k answers via a suite of intelligent pruning techniques. The suggested models consist of a practically important subset of the SPARQL query language augmented with some additional useful features.

The first model called Substitution Importance Query (SIQ) identifies the top-k answers whose scores are calculated from matched vertices' properties in each answer in accordance with a user-specified notion of importance. The second model called Vertex Importance Query (VIQ) identifies important vertices in accordance with a user-defined scoring method that builds on top of various subgraphs articulated by the user. Approximate Importance Query (AIQ), our third model, allows partial and inexact matchings and returns top-k of them with a user-specified approximation terms and scoring functions. In the fourth model called Probabilistic Importance Query (PIQ), a query consists of several sub-blocks: one mandatory block that must be mapped and other blocks that can be opportunistically mapped. The probability is calculated from various aspects of answers such as the number of mapped blocks, vertices' properties in each block and so on and the most top-k probable answers are returned.

An important distinguishing feature of our work is that we allow the user a huge amount of freedom in specifying: (i) what pattern and approximation he considers important, (ii) how to score answers - irrespective of whether they are vertices or substitution, and (iii) how to combine and aggregate scores generated by

multiple patterns and/or multiple substitutions. Because so much power is given to the user, indexing is more challenging than in situations where additional restrictions are imposed on the queries the user can ask.

The proposed algorithms for the first model can also be used for answering SPARQL queries with ORDER BY and LIMIT, and the method for the second model also works for SPARQL queries with GROUP BY, ORDER BY and LIMIT. We test our algorithms on multiple real-world graph databases, showing that our algorithms are far more efficient than popular triple stores.

Top-K Query Processing in Edge-Labeled Graph Data

by

Noseong Park

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
2016

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor V.S. Subrahmanian for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past five years. He has always made himself available for help and advice and there has never been an occasion when I've knocked on his door and he hasn't given me time. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank my research mentors, Professor Sushil Jajodia and Professor Andrea Pugliese. Without their extraordinary supports, my finishing PhD would have been a distant dream. Thanks are due to Professor Rama Chellapa, Professor Amol Deshpande and Professor David Mount for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

My colleagues have enriched my graduate life in many ways and deserve a special mention. Michael Ovelgonne, Edoardo Serra and Francesco Parisi helped me develop theory. Tanmoy Chakraboty, Chanhyun Kang, Srijan Kumar, Jason Fillipou, Anshul Sawant, Francesca Spezzano provided help in various ways.

I would also like to acknowledge help and support from some of the staff members. UMIACS staff members' technical help is highly appreciated, as is the general support from Barbara Lewis.

I owe my deepest thanks to my family - my wife and son who have always stood by and ecouraged me, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them.

I would like to acknowledge financial support from all the projects discussed herein. It is impossible to remember all, and I apologize to those I've inadvertently left out.

Lastly, thank you all!

Table of Contents

List of Figures

List of Abbreviations

Graph DB	Graph DataBase
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
DBMS	Data Base Management System
LUBM	Lehigh University Benchmark (LUBM)
SP^{2}Bench	SPARQL Performance Benchmark
BSBM	Berlin SPARQL BenchMark
SPINN	Suspicion Prediction in Nuclear Networks
IMDB	Internet Movide DataBase
DOGMA	Disk-Oriented Graph Matching Algorithm
SIQ	Substitution Importance Query
VIQ	Vertex Importance Query
AIQ	Approximated Importance Query
PIQ	Probabilistic Importance Query
VPI	Vertex Property Index
VSI	Vertex Step Index
PQ	Pattern Query
S-Query	Scoring Query
lagg	Local Aggregation
gagg	Global Aggregation
lscore	Local Score
optPPQ	Optimal Partial Pattern Query
NWST	Node Weighted Steiner Tree
LP	List-Oriented Pruning

Chapter 1: Preliminaries

1.1 Introduction

The need to efficiently search graph-structured resources such as the Internet and social networks is increasing dramatically. Major companies track what is being said about them on online social networks in order to monitor their products, brands, and reputations. Financial institutions monitor what is being said on social media in order to forecast movements of stock prices, possibly in conjunction with other variables. Law enforcement organizations monitor chatter on online social networks such as Facebook and Twitter to see if violence is being planned at protests. Governments monitor what is being said online in order to track a variety of terrorist groups. Also related is the Semantic Web, where RDF datasets can be viewed as massive graph databases and (large fragments of) SPARQL queries can be viewed as subgraph matching queries, with the problem of query answering being viewed as a kind of matching problem.

There are many applications where a user wishes to identify subsets of vertices in the graph that are of interest to him. One popular way for this is to specify subgraph queries, and multiple answers in the graph databases may end up matching such queries. The user, however, wishes to score the answers in some way, also taking into account various aspects of answers such as vertex properties, the number of answers to which a specific vertex is matched, and so on. If one such query is not enough to express all interests, multiple queries can be defined and scores should be aggregated. But as he knows his mission and application needs better than most designers of graph DBMSs, he wants to specify how to score the answers as part of his query. From the graph DBMS perspective, allowing the user to specify scoring mechanisms poses a challenge because they do not know the query or the scoring method in advance and hence cannot plan indexing and query optimization methods as well as they could if they knew how scoring was being done. While almost all past work on top-k queries to graph databases do not allow the user to express scoring methods within his query, in this proposal, we propose a couple of methods that will bring this power to the user.

We first present a few sample application scenarios from the Web and social networks, where such a capability is useful.

<u>Potential fraud in banking</u> Banks are required to report suspicious transactions to their respective governments. The definition of "suspicious" varies from one country to another, but in most developed economies at least, the way money flows through the banking network is one important criterion. A bank official would consider the set of all accounts (or account holders) both in his bank and corresponding banks (i.e. banks from which either inflows or outflows of funds occur) as vertices in a network. There is an edge (v_1, v_2) in this network if vertex v_1 transfers funds to vertex v_2. The edge may be labeled with multiple properties (e.g. amount transferred, date, etc.).

Based on past experience, the banker defines subgraph queries reflecting suspicious accounts and/or fund transfers. To score the answers to these queries, he takes advantage of his knowledge of suspicious banking transactions and defines a scoring function in such queries. The top-k users identified are then presented by the system to him.

Potential job-matches on LinkedIn LinkedIn provides valuable services to companies by matching company job requirements with the skills of users. When it contracts to advertise a job, both LinkedIn and the company involved wish to ensure that the best matches for the job are found. In this case, LinkedIn may define multiple subgraph queries that identify vertices in the LinkedIn graph database (people are vertices, edges are contacts between people) that are of interest. These queries may also require that users have certain skills, satisfy nationality requirements, or require (maybe through another separate query) that the users be linked to people already in the company who have a high job performance score or who have graduated from universities (and with similar GPAs) that the company has had good experiences with in the past. Each requirement is scored by a separate query and the company then specifies how to aggregate these scores into a unified score. For instance, if a particular user Bob is both well-connected with people in the company and comes from a school/university from which the company has had good experiences in the past, they may want to give him a higher score than if he met just one of these criteria.

Identifying the best restaurants in a city on TripAdvisor Suppose TripAdvisor wishes to rank the best restaurants in a city C. In order to do this, they may aggregate scores of all their users. But there are occasional allegations of fraud in TripAdvisor reviews (e.g. the restaurant owner may try to post fraudulent reviews). TripAdvisor is reported to have a team solely to identify such fraud [67]. So TripAdvisor may look not only at the reviews themselves, but also potential inter-relationships between users. In this case, vertices can be used to represent reviews and restaurants, and there is an edge from review rev to restaurant res if rev is about res. There is also an edge between two reviews if they were not written by the same person (or if TripAdvisor inferred this, via some mechanisms). The review may have attributes such as star rating, results of natural language processing, information about the author, and more. TripAdvisor may come up with a set of subgraph queries that identify excellent restaurants in city C. For instance, these subgraphs may include patterns that ensure that good reviews are provided by truly different sources. A scoring function can then assign a score to each restaurant, with one such score for each query — another function merges these scores together.

Identifying suspect vertices on Twitter There are a number of suspicious vertices on Twitter. Such vertices include ones that are bots or engage in fraudulent click-through behavior. For instance, a popular strategy on Twitter is for fraudulent vertices to copy a popular tweet containing a micro-URL from a legitimate source (e.g. a news source) and replace the micro-URL to one where the advertiser pays them for a click [29]. An advertiser or the government may wish to find such fraudulent links. Often, fraudulent accounts have a number of other fraudulent accounts following them, artificially inflating their counts. They also follow a number of fraudulent accounts as well as some legitimate ones in order to evade detection [26]. A Twitter executive (or the ripped off advertiser) trying to identify such fraudulent accounts may define a number of subgraph queries, along with constraints on the vertices' tweeting behavior. He then develops way of scoring the answers to such queries satisfying the queries (and constraints) via appropriate scoring functions.

All these applications share two common factors; users specify i) subgraph patterns and ii) how vertices/substitutions involved in the resulting match are scored. The system doesn't know this in advance, only at query time.

In Chapter 2, to this end, we propose an aggregate top-k computation model called *Vertex Importance Query (VIQ)* whose goal is to identify top-k vertices by user defined scoring and aggregation terms and which is a mixture of a meaningful subset of SPARQL 1.1 and some additional features not supported by SPARQL 1.1. Each answer is separately scored based on its mapped vertices' properties and projected to a vertex, and scores of all answers projected to same vertex are aggregated to define the score of the vertex. In order to calculate an upper bound score of a vertex, we propose a novel index structure called *PM_Index* and a suite of pruning algorithms. In particular, the suggested index and algorithms can also be used for answering SPARQL queries with GROUP BY[1], ORDER BY and LIMIT. To the best of our knowledge, the suggested methods are the first ones that answer

[1]Please note that with GROUP BY its query processing is much more complicated than plain top-k queries only with ORDER BY and LIMIT

aggregate top-k subgraph matching queries.

In Chapter 3, yet another top-k computation model called *Substitution Importance Query (SIQ)*, where only top-k answers are returned from among the huge number of answers obtained by subgraph matching queries, is proposed. The score of an answer is defined by mapped vertices' properties, e.g. age, salary, which are combined together by a scoring function specified in the query. This is analogous to SPARQL queries with ORDER BY and LIMIT.

We define Approximate Importance Query in Chapter 4 in order to consider various approximation points. First, an edge in a query can be mapped to a multi-hop path of a graph database. Second, some edges of the query can be missing in a matched pattern. After allowing all these approximations, the number of answer candidates exponentially increases, and thus the necessity of top-k increases as well. All exact matches and approximate matches are scored by a user-written scoring function and only the top-k of matches are returned to users.

Another method in Chapter 5 is to express uncertainty in queries. Sometimes users don't know how to write the best query or there are several different ways to express their interests. In such cases, our method allows users to draw several query-blocks to constitute a user query and patterns of graph database matched to any combination of blocks are valid answers. Users can specify a probability function to find the top-k most probable answers. In general, users may give a higher probability as more blocks are mapped and matched vertices have more appropriate properties.

We make the following contributions throughout the main four chapters.

- We propose a top-k computation model called *Vertex Importance Query (VIQ)* to use projections and aggregations for scoring vertices. In order to decrease response time, we suggest a novel index structure to calculate an upper bound score of vertices and a suite of pruning methods based on them.

- Another top-k computation model called *Substitution Importance Query (SIQ)* is also suggested to score answers, i.e. matched patterns, against queries. Each answer is scored by its mapped vertices' properties. We also propose a specialized index structure and several pruning methods.

- *Approximate Importance Query (AIQ)* is proposed to consider various approximation points in terms of connectivity. With allowing approximations, one query pattern is indeed converted into many similar queries and we proposed a novel index structure and algorithm to find only top-k answer from all those similar query patterns.

- We also propose *Probabilistic Importance Query (PIQ)* to identify the top-k most probable answers against users' uncertainty queries.

- All the methods we propose for *VIQ* can be used for answering SPARQL queries with GROUP BY, ORDER BY and LIMIT, and to the best of my knowledge, this is the first work to specifically focus on solving those types of queries efficiently.

- All the methods we propose for *SIQ* can be used for answering SPARQL queries with ORDER BY and LIMIT.

- We test all methods in 7 real datasets, whose size is up to 234M edges, and we could get meaningful improvements.

In Chapter 5, we conclude this proposal after summarizing our contributions and their significance.

1.2 Related Work

We consider related work in three areas: general subgraph matching, approximate subgraph matching, and top-k query processing.

General subgraph matching The problem of efficiently evaluating subgraph matching queries over huge graphs/networks has been recently addressed in different scenarios, among which social network analysis and RDF database management play an important role [42]. Many social network analysis methods [36, 50] operate in main memory. For social networks of the size of Facebook and Flickr, such an approach is infeasible and disk-based data management becomes mandatory. More importantly, complex queries involving even a few joins can quickly cause such approaches to run into trouble. In the RDF realm, approaches differ with respect to their storage regime and query answering strategies [41]. However, the great majority of RDF databases are triple oriented, in the sense that they focus on the storage and retrieval of individual triples. Some systems (e.g. [16, 57, 1]) use relational databases as their back-end and a relational query engine to answer queries. Others propose native storage formats for RDF (e.g. [13, 39]). [7] shows that storing RDF in a vertical database leads to significant query time improvements, whereas [48] focuses on physical data structures and join processing. [58] uses triple selectivity estimation techniques similar to those used in relational database systems. Other works on general graph data management and subgraph matching (e.g. [20, 32, 38, 54, 69, 70]) typically employ heuristics to predict the cost of answering strategies based on statistics about the dataset and the current state of query processing and then choose a strategy to minimize cost. For instance, [72] proposes to transform vertices into points in a vector space, thus converting queries into distance-based multi-way joins over the vector space. In [22] the authors propose a two-step join optimization algorithm based on a cluster based join index. GADDI [68] employs a structural distance based approach and a dynamic matching scheme to minimize redundant computations. GADDI can handle graphs with thousands of vertices, which are common in many biological applications. In [69] the authors propose SUMMA, which improves over GADDI and employs more advanced indices, becoming capable of handling graphs with up to tens of millions of vertices. The algorithm in [70] employs an aggressive pruning strategy based on an index storing label distributions. In [46], 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. COSI [14] built upon DOGMA [13] and proposed a cloud-based algorithm for subgraph matching that works very efficiently on graphs with over 770M edges. [15] extends DOGMA to handle skewed degree distributions in graph databases and shows how to quickly answer complex subgraphs queries on graph data with over 1B edges. None of the above works handle uncertainty or user defined scoring in their queries.

Approximate subgraph matching [49] proposed a basic algorithm for the problem of approximately matching a query against a database of *many small* graphs. The algorithm is based on A^* search whereby each vertex in the search tree represents a vertex pair matching. Each vertex pair matching has an associated cost that is inversely proportional to the "goodness" of the match. The basic notion of *edit distance* [56] between two graphs is used in many approximate subgraph matching systems proposed since. Grafil [66] tries to identify frequently occurring but selective subgraphs in a graph database, which are called *features*. Given a query graph, all features in the query are determined — this allows the computation of a feature similarity score (based on the number of missing features) and return approximate matches. SAGA [60] measures the distance between two graphs as a weighted linear combination of the structural difference, vertex mismatches, and vertex gaps. TALE [61] 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 and then to look up all graphs in the database which contain the vertices in V. The measures proposed in SAGA and TALE for determining the degree of similarity between graphs are guided by the authors' intuition about the problem domain (such as the importance of vertex labels in biological networks). Another line of work [45, 64] proposes a probabilistic model for the alignment between a query graph and a *host* graph. The focus of this work is to describe a probability distribution over the set of all possible alignments using features from either graph and then to find the most likely assignment. None of the above works have demonstrated scalability to operate on graphs with millions of vertices and edges. Probabilistic matching between a query and *one huge* graph database (upto over 1B edges) was shown in [12] — however, the proposed approach only allows setting fixed “probabilistic” scores.

Top-k query processing One way to improve the performance of triplestores for top-k SPARQL queries has been proposed in [40]. They propose a rank-aware join algorithm for queries with additive SUM scoring functions for substitutions. Our framework supports many complex user-defined scoring functions, not only restricted to SUM. [53] suggested using twig queries to score substitutions and a query edge is mapped to a path in the data. The score of a substitution is the sum of the weights of all the edges involved in the edge to path mappings. [24] extends this work to graph data. The k most similar subgraphs to a query graph, where similarity is defined using maximum common subgraphs, are targeted in [71]. [65] lets users score vertices via a function f that represents a “relevance” for each vertex. Once these relevance scores are computed for all vertices, a higher level is computed as $\sum_{u \in Nbr(v,h)} f(u)$, where $Nbr(v,h)$ is a set of h-hop neighbors of v. Top-k vertex set search by graph simulation has been suggested in [30], where the score of the output variable of a query is defined as the size of the maximum graph simulation matching the output variable to a vertex. Both [65] and [30] do not allow the user to choose the final scoring function. The features of some of the above works are summarized in Table 1.1.

As Table 1.1 shows, three different top-k computation models that will be introduced in the following chapters have much sophisticated features in comparison

	Query blocks	Scored object	Full user defined scoring	Advanced scoring (Aggreg. or Normaliz.)	Approx. Matching
[40]	✗	Substitution	✓	✗	✗
[53]	✗	Substitution	✗	✗	✗
[24]	✗	Substitution	✗	✗	✗
[71]	✗	Substitution	✗	✗	✗
[65]	✗	Vertex	✗	✓	✗
[30]	✗	Vertex	✗	✗	✗
Chapter 2	✗	Vertex	✓	✓	✗
Chapter 3	✗	Substitution	✓	✗	✗
Chapter 4	✗	Substitution	✓	✗	✓
Chapter 5	✓	Substitution	✓	✓	✗

Table 1.1: Comparison with related works

with past work. We believe that these bring a better opportunity for users to mine their interested patterns more accurately and precisely.

1.3 Basic Definitions

In this section, we define edge-labeled graphs and some basic terms and notations. We assume the existence of four arbitrary, but fixed, disjoint sets V, $\mathcal{L}$, $\mathcal{P}, \mathcal{T}$ of *vertex names*, *edge labels*, *vertex properties*, and *type names*, respectively. Each $p \in \mathcal{P}$ has an associated domain $dom(p)$ which is a set (disjoint from V, $\mathcal{L}$, and $\mathcal{P}$) of values that can be assigned to p. We assume the existence of a special property $type \in \mathcal{P}$ with domain $\mathcal{T}$. A graph database is defined as follows.

Definition 1.3.1 (Graph Database). *A* Graph Database $\mathcal{G}$ *is a triple* $\langle V, E, \wp \rangle$, *where*

- $E \subseteq V \times \mathcal{L} \times V$ *is a set of labeled edges;*
- $\wp : V \times \mathcal{P} \to \bigcup_{p \in \mathcal{P}} dom(p)$ *is a partial* property assignment *function, such that*

(i) if $\wp(v,p)$ is defined, then $\wp(v,p) \in dom(p)$ and (ii) $\wp(v, type)$ is defined for all $v \in V$. [2]

A sample graph database is shown in Figure 2.2. If $\mathcal{P} = \{type,\ level\}$ and $dom(type) = \{person,\ product,\ company\}$, the property function assigns the value *person* to the pair (*Steve, type*), meaning that *person* is the type of the vertex named *Steve*.

Definition 1.3.2 (Term; Numeric Term)**.** *A* (numerical) term *is a value of a property, a property of a query variable, a real number, or a polynomial-time computable function of numerical terms.*

- *Every member of $\bigcup_{p \in \mathcal{P}} dom(p)$ is a term. If $nt \in \bigcup_{p \in \mathcal{P}} dom(p) \cap \mathbb{R}$, then nt is a* numeric term*.*
- *If $?x \in \mathsf{VAR}$ and $p \in \mathcal{P}$, then $?x.p$ is a term. If $dom(p) \subseteq \mathbb{R}$, then $?x.p$ is a* numeric term*.*
- *If nt_1, nt_2 are numeric terms, then $f(nt_1, nt_2)$ is a numeric term if f is a polynomial-time computable function.*

A term is ground *if no variables occur in it. We say a term t is* solely about variable *$?x$ if $?x$ is the only variable occurring in t. For instance, $(?x.level - 1)$, $(?x.level - 10) * (?y.level + 2)$ are both terms. The importance query (definition follows) shown in Figure 3.2 contains the numeric term $?r.stars$. We assume that all* ground *numeric terms are evaluated, e.g. the numeric term $2 + 3$ is evaluated to 5.*

[2] $\wp$ is partial function as it is not necessarily defined for all pairs $(v, p) \in V \times \mathcal{P}$.

Definition 1.3.3 (Constraint)**.** *(i) If t_1, t_2 are terms, then $t_1 = t_2$ and $t_1 \neq t_2$ are constraints.*

(ii) If nt_1, nt_2 are numeric terms, then $nt_1 < nt_2, nt_1 \leq nt_2, nt_1 > nt_2, nt_1 \geq nt_2$ are constraints.

(iii) If c_1, c_2 are constraints, then $c_1 \wedge c_2$ is a constraint.

For instance, $?x.level > ?y.level$ and $(?x.level > 5 \wedge ?x.level \leq y.level + 3)$ are both constraints. The example importance query in Figure 3.2 contains for variable $?r$ the constraint $?r.type = restaurant$. We say constraint C is solely about variable *$?x$ if $?x$ is the only variable occurring in C.*

Let VAR be a set of variable symbols (we use strings starting with question marks as for variables). We assume VAR is disjoint from V, $\mathcal{L}$, $\mathcal{P}$, $\mathcal{T}$ and from $\bigcup_{p \in \mathcal{P}} dom(p)$. The idea is that variables must be "bound" to vertices in the graph database.

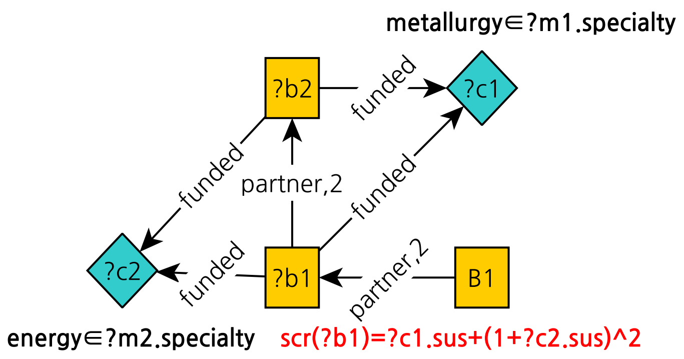
Chapter 2: Vertex Importance Queries

We consider a simple nuclear proliferation example in which an analyst is trying to identify suspicious entities. The analyst already has a network consisting of vertices (people, companies) and links between them. For instance, SPINN [8] extracts this network by first finding individuals/companies who have been sanctioned by the US and other similar governments for export control violations. SPINN then uses LinkedIn and Bloomberg's worldwide lists of directors and officers of companies to flesh this network out completely. Of course, not everyone in the SPINN's network of over 74K vertices and over 1M edges is necessarily bad. The suspiciousness level is represented as one of $\{1, 0, -1\}$ — each represents "suspicious", "unknown", and "non-suspicious", respectively.

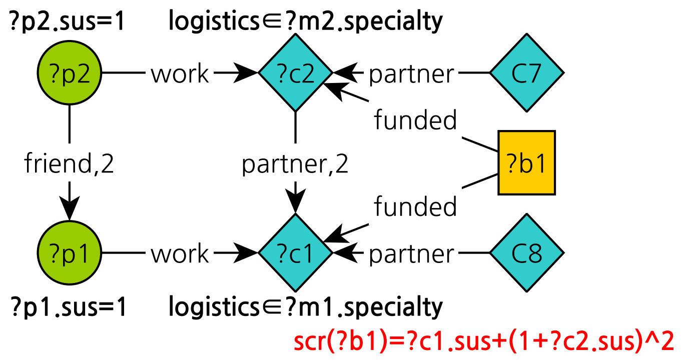
Our knowledge of nuclear proliferation networks that have been previously discovered suggests some typical suspicious connections among suspect vertices. Two anonymized example pattern queries created from them are shown in Figure 2.1, one corresponding to a suspicious financial network, and another corresponding to a suspicious logistics network — both these can be illicit networks because money and materials must both be "moved" in order to achieve a desired goal. The query PQ_1 looking for suspicious financial networks starts by focusing on the suspicious bank

B1 — which has come to the notice of nuclear regulators for a reason. They are looking for banks matched for $?b1$ (through a path of "partner" links of maximum length 2) which has financed an energy company $?c2$ and a metallurgy company $?c1$. Such a subnetwork might be suspicious because bank $?b1$ is financing nuclear supply producers. PQ_2, similarly, is looking for logistics companies $?c1$ and $?c2$ that are partners of the suspicious logistics companies C7 and C8, as well as two people $?p1$, $?p2$, and a bank $?b1$. Bank $?b1$ must have funded both $?c1$ and $?c2$. $?p1$ and $?p2$ must have friendship and are both linked to the suspicious logistics companies. If the same bank appears as a solution to $?b1$ in both queries, then it would naturally be of greater interest to the analyst.

To this end, we score matched patterns of each query with $?c1.sus + (1{+}?c2.sus)^2$ and project to $?b1$. In both queries, $?c2$ plays a more important role for nuclear proliferation. A vertex score in each query is then defined as an aggregated value (we use SUM in the example) over the scores of all patterns projected to the same vertex. Finally, we total the vertex score of each query up. With the simple SPINN network example of Figure 2.2, one pattern of PQ_1 maps $?b1$ to B2 — $?b2$ is mapped to B4, $?c1$ to C2 and $?c2$ to C1, so B2's score is $1 + (1 + 1)^2 = 5$ for PQ_1. Two patterns of PQ_2 map $?b1$ to B2. In the first one, $?c1$ is mapped to C6, $?c2$ to C1, $?p1$ to P4, and $?p2$ to P1, so B2's score is $0 + (1 + 1)^2 = 4$. In the second one, $?c1$ is mapped to C2, $?c2$ to C1, $?p1$ to P3, and $?p2$ to P1, so B2's score is $1 + (1 + 1)^2 = 5$. Thus, the overall score of B2 w.r.t. PQ_2 is 9, and its final score is 14 if we use SUM to aggregate scores (the final scores of B3, B4, and B6 are 6, 4, and 1, respectively).



(a) PQ_1



(b) PQ_2

Figure 2.1: Example query. Circles represent people, diamonds represent companies, and rectangles represent banks. An edge annotated with an integer number w can be matched to paths with maximum length w.

Queries such as this require the following features. (i) The user must be able to specify a set of patterns of interest to him. (ii) The user must be able to specify a scoring function that extracts properties of a vertex ($?c1.sus$ above) and use it to define a scoring metric. (iii) The user must be able to specify maximum path lengths in his query. (iv) The user must be able to "bump up" the score of a vertex that satisfies multiple criteria by aggregating them (as is done by adding up the two scores above).

The use of patterns similar to those in the SPINN nuclear trafficking application applies in many different commercial fraud investigations as well. For instance,

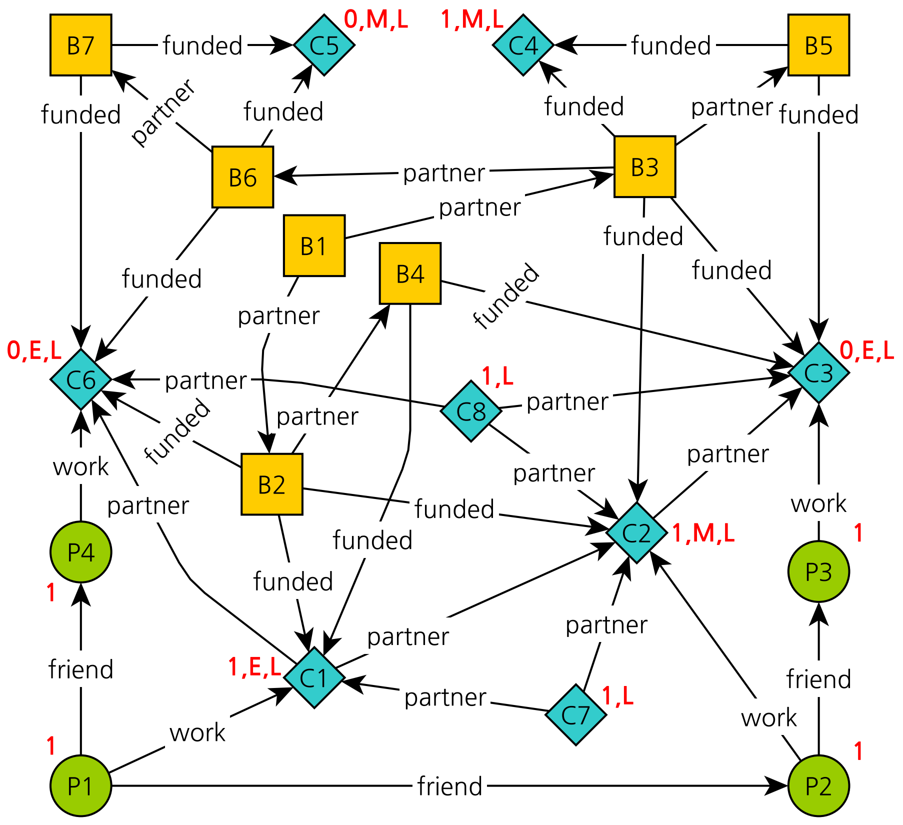


Figure 2.2: Example graph database $\mathcal{G}$. Vertices are annotated with the values of their *suspicious* and *specialty* properties; integers represent suspiciousness levels; "M" stands for "Metallurgy", "L" stands for "Logistics", and "E" stands for "Energy".

one type of medical insurance fraud (MIF) involves staging fake accidents. The individuals and organizations involved in such MIF networks involve a network of individuals (called "runners") who stage accidents and bring fake "victims" to one or more "clinics". Another network of individuals handle the legal side of things — they involve networks of lawyers and claimants who work closely together in the scam. A third network of individuals include the medical professionals and doctors who deliver the medical services — doctors involved in the fraud are often connected with other doctors. And a fourth network ties the clinics involved that ultimately bill the insurance company. Often times, the same patient visits many different clinics, and the same doctors are tied to different clinics involved in the fraud, and so are the lawyers. A person who ends up occurring in many of these networks is generally more suspicious than others. Insurance investigators wish to specify methods to focus on the most suspicious individuals — something that can be expressed via the scoring queries proposed in this paper — and they are generally looking for the top-k suspicious people as they are limited in their investigative resources.[1]

The rest of this section describes related work. Then, in Sections 1.3 and 2.1, we define graph databases, propose *Scoring Queries* that capture the above features, and define the top-k answers to scoring queries. Section 2.2 describes our scoring query processing algorithm. The algorithm includes a method to estimate an upper bound on the score of any vertex which allows us to prune the search for top-k answers. The estimation of these upper bounds is critical, but unfortunately, exactly finding these upper bounds is NP-hard. Because of this, we present two algorithms: the PScore_LP algorithm where the optimal upper bound is found using a list-oriented pruning method, and the PScore_NWST algorithm which leverages Node-Weighted Steiner Trees to quickly find very good upper bounds (though they may not be optimal). Section 2.3 discusses how our approaches can be extended to encompass any maximum path length constraints expressed in the query. Section 4.4 summarizes our experiments on 7 real-world data sets involving graphs with upto 234M edges.

[1]See *http://linkanalysisnow.com/2009/04/analyzing-medical-fraud.html* for a detailed analysis of how such scams work. This site also provides links to lots of other fraudulent activities where scoring queries may play a natural role.

2.1 Formal Definition of Vertex Importance Queries

In this section, we formally define scoring queries, an example of which was informally given in the Introduction. A *Scoring Query* (S-query) consists of a set of *Pattern Map Queries* (PM-queries) as shown by the graphs in Figure 2.1 and a global aggregation function according to which the scores of answers generated by the individual PM-queries can be evaluated. Intuitively, a PM-query is used to specify a way to (locally) score a class of sub-graphs of $\mathcal{G}$ having some characteristic (structure, vertex properties, or edge labels) the user wants to look for.

2.1.1 Pattern Map Queries

As we defined in Section 1.3, VAR is a set of variable symbols (we use strings starting with question marks as for variables). We assume VAR is disjoint from V, $\mathcal{L}$, $\mathcal{P}$, $\mathcal{T}$ and from $\bigcup_{p\in\mathcal{P}} dom(p)$. The idea is that variables must be "bound" to vertices in the graph database. A *(numerical) term* is a value of a property, a property of a query variable, a real number, or a polynomial-time computable function of terms. For instance, $(?x.sus - 1)$, $(?x.sus + 1) * (?y.sus + 2)$ are both terms. A *constraint* is a comparison between terms, the application of a set operator to terms, or the application of a logical operator ($\wedge$, $\vee$, or $\neg$) to constraints. For instance, $Energy \in ?x.specialty$, $?x.sus > ?y.sus$, and $(?x.sus > 0 \wedge ?x.sus \leq y.sus + 1)$ are all constraints.

We are now ready to introduce the notion of pattern map query. Let $*$ be a distinguished "wildcard" label not occurring in $\mathcal{L}$.

Definition 2.1.1 (Pattern Map (PM) Query)**.** *A* pattern map query *(PM-query) is a 5-tuple* $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \mathtt{lagg} \rangle$*, where:*

- $V_{PQ} \subseteq V \cup \mathsf{VAR}$ *with* $V_{PQ} \cap V \neq \emptyset$.[2]
- $E_{PQ} \subseteq V_{PQ} \times (2^{\mathcal{L}} \cup \{*\}) \times \mathbb{Z}^{+} \times V_{PQ}$.
- χ *is a finite set of constraints over variables in* V_{PQ}.
- ϱ *is a partial function which associates variables in* V_{PQ} *with numerical terms.*
- $\mathtt{lagg} \in \{MIN, MAX, AVG, SUM\}$ *is a "local" aggregation function.*

We assume that every variable in V_{PQ} *has a type constraint. In addition, if* $\varrho(?x)$ *is defined and its value is* t*, then we say that* $?x$ *is an* output variable*,* t *is a* scoring term*, and any variable occurring in* t *is a* scoring variable*. Moreover, if* E_{PQ} *contains an edge* $e = (v, L, w, v')$ *with* $w > 1$*, then we say that* e *is* annotated *and* PQ *is* edge-annotated*.*

The first two items of Definition 2.1.1 say that a PM-query is a directed edge-labeled graph where (i) each vertex is either a vertex name or a variable symbol, and (ii) each edge is labeled with a pair consisting of a set of edge labels (or $*$) and an integer. Intuitively, the user uses this graph to specify the structure of the sub-graphs of $\mathcal{G}$ he is looking for (where $*$ matches any label), including constraints about the maximum length of paths. The constraints in χ express requirements

[2]We only consider *anchored* queries, that involve at least one constant. Anchored queries are considered of fundamental importance in many scenarios — for instance, anchors are present in (i) all queries in both the *DBPedia* [44] and *LUBM* [3] SPARQL benchmarks, (ii) 14 out of 20 queries in the *interactive* use case of the *Social Network Intelligence BenchMark* [5], (iii) 19 out of 20 queries in the *explore* and *business intelligence* use cases of the *Berlin* benchmark [11], and (iv) 14 out of 15 queries in the *SP*^{2}*Bench* benchmark [6].

about vertex properties. Function ϱ provides a flexible way for ranking the vertices the user is interested in by means of (possibly complex) scoring terms, while `lagg` is a local aggregation function that tells us how scoring terms are aggregated. Finally, output variables are those that will be mapped to the vertices the user wants to compute local scores for, whereas scoring variables are those that will be mapped to the vertices that contribute to such scores.

A scoring query consists of a set of PM-queries along with a global aggregation function that combines the scores returned by each PM-query.

Definition 2.1.2 (Scoring Query). *A* scoring query *(S-query) is a pair $SQ = \langle PQS, \texttt{gagg} \rangle$, where PQS is a finite set of PM-queries and $\texttt{gagg} \in \{MIN, MAX, AVG, SUM\}$ is a "global" aggregation function.*

For instance, if we consider the PM-queries PQ_1 and PQ_2 of our running example, $SQ = \langle \{PQ_1, PQ_2\}, SUM \rangle$ is an S-query that ranks banks by the sum of the scores obtained from PQ_1 and PQ_2.

In order to define the *answer* to a PM-query and an S-query, we need some intermediate concepts.

A *substitution* θ is a mapping $\theta : \mathsf{VAR} \rightarrow V$. Applying θ to an expression e (vertex, term, or constraint) means replacing every variable $?x$ in e with vertex name $\theta(?x)$, and the ground (i.e. variable-free) expression resulting from such an application is denoted $e\theta$. Ground terms and ground constraints are evaluated in the obvious way.

Definition 2.1.3 (Answer Subst. for a PM-query). *Given a PM-query $PQ =$*

$\langle V_{PQ}, E_{PQ}, \chi, \varrho, \texttt{lagg}\rangle$, *an* answer substitution θ *for* PQ *is a mapping* $\theta : V_{PQ} \to V$ *such that*

- *for each* $(\alpha, L, m, \beta) \in E_{PQ}$, *there is path* $(v_1, \ell_1, v_2), (v_2, \ell_2, v_3), \ldots, (v_{m'}, \ell_{m'}, v_{m'+1})$ *in* $\mathcal{G}$ *with* $m' \leq m$ *such that* $v_1 = \alpha\theta$, $v_{m'+1} = \beta\theta$, *and, for each* $i \in [1..m']$, $\ell_i \in L$ *if* $L \neq \{*\}$*;*

- *for each constraint* $C \in \chi$, $C\theta$ *evaluates to true.*

For instance, two answer substitutions for PQ_1 of our running example are $\{?b1/\text{B3}, ?c1/\text{C4}, ?c2/\text{C3}, ?b2/\text{B5}\}$ and $\{?b1/\text{B6}, ?c1/\text{C5}, ?c2/\text{C6}, ?b2/\text{B7}\}$.[3]

The existence of an answer substitution θ for a PM-query PQ guarantees that there is a way of mapping all the variables in PQ to vertices of $\mathcal{G}$ so that it satisfies all other requirements specified by PQ: presence of vertex names specified in the query, paths having a given maximum length with appropriate edge labels, and vertex properties satisfying the constraints.

Given an answer substitution θ for PQ, and an output variable $?x$ of PQ, the score of a vertex $v = ?x\theta$ in $\mathcal{G}$ can be computed by evaluating $\varrho(?x)\theta$, which results in a numerical value (obtained when the numeric term $\varrho(?x)$ is instantiated by θ). Thus, if a vertex $v \in \mathcal{G}$ is the result of applying an answer substitution θ to a variable $?x \in V_{PQ}$, then it has a (local) score obtained by using `lagg` to aggregate all the values $\varrho(?x)\theta$. This is captured formally below.

Definition 2.1.4 (Local Score)**.** *Let* v *be a vertex in* $\mathcal{G}$, *and* $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \texttt{lagg}\rangle$ *a PM-query. Let* S *be the set* $\{\varrho(?x)\theta \mid \theta$ *is an answer substi-*

[3]We use *multisets* of substitutions in general, since two same substitutions with different intermediate edges between vertices, when edges are annotated, are counted twice [34].

tution for PQ and $v = ?x\theta\}$. The local score of v w.r.t. *PQ, denoted $\mathtt{lscore}(v, PQ)$, is $\mathtt{lagg}(S)$ if $S \neq \emptyset$, and $\perp$ (undefined) otherwise.*

For instance, in our running example, $\mathtt{lscore}(\text{B2}, PQ_1) = 5$ and $\mathtt{lscore}(\text{B2}, PQ_2) = 9$.

Given an S-query $SQ = \langle PQS, \mathsf{gagg}\rangle$ and a vertex v of $\mathcal{G}$, we compute a single score for v w.r.t. SQ by aggregating (via the global aggregate function) the local scores of v w.r.t. each PM-query $PQ \in PQS$.

Definition 2.1.5 (Score)**.** *Let v a vertex in $\mathcal{G}$ and $SQ = \langle PQS, \mathtt{gagg}\rangle$ an S-query. Let S be the set $\{\mathtt{lscore}(v, PQ) \mid PQ \in PQS$ and $\mathtt{lscore}(v, PQ) \neq \perp\}$. The* score *of v* w.r.t. *SQ is defined as $\mathtt{score}(v, SQ) = \mathtt{gagg}(S)$ if $S \neq \emptyset$, and 0 otherwise.*

In our running example $\mathtt{score}(\text{B2}, SQ) = 14$, $\mathtt{score}(\text{B3}, SQ) = 6$, and $\mathtt{score}(\text{B4}, SQ) = 4$.

The top-k answer to an S-query SQ is given by the set of vertices of $\mathcal{G}$ having the k highest scores.

Definition 2.1.6 (Top-k Answer to an S-query)**.** *Let $SQ = \langle PQS, \mathtt{gagg}\rangle$ be an S-query and $k \leq |V|$ a positive integer. The* top-k answer *to SQ is the set $Ans(SQ) = \{v_1, \ldots, v_k\} \subseteq V$ such that (i) for each $i \in [1..k-1]$, $\mathtt{score}(v_i, SQ) \geq \mathtt{score}(v_{i+1}, SQ)$; (ii) there is no vertex $v \in V \setminus Ans(SQ)$ such that $\mathtt{score}(v, SQ) > \mathtt{score}(v_k, SQ)$.*

In our running example, the top-2 answer to SQ is the set {B2, B3}.

2.2 Query Processing

To efficiently compute the top-k answer to an S-query $SQ = \langle PQS, \texttt{gagg} \rangle$, we perform the following steps:

1. We *partially* match the PM-queries in PQS.
2. We estimate *upper bounds* on the scores obtainable by vertices in $\mathcal{G}$.
3. We then progressively *extend* these partial matches. In the process, vertices that are no longer candidates to be in the top-k answer (since their upper bounds are lower than the current top-k) are pruned.

In this section we describe the techniques we developed to support the above process in the case of non-annotated queries. Later, in Section 2.3, we discuss the extensions for the case of edge-annotated queries. Table 2.1 summarizes the symbols used.

2.2.1 Partial Matching

In our approach, partial matching is performed by computing substitutions for *a subset of* the variables in a given PM-query PQ. A query variable is said to be *essential* if it is either an output variable or a scoring variable in PQ. We use $\mathcal{E}(PQ)$ to denote the set of all essential variables in PQ. For instance, in both the PM-queries of our running example, $?b1$, $?c1$, and $?c2$ are essential variables. Formally, a partial matching is the application of a substitution to a *partial PM-*

Table 2.1: Symbols used in Sections 2.2 and 2.3.

Symbol	*Meaning*
PQ	PM-query
$SQ = \langle PQS, \texttt{gagg}\rangle$	S-query (PQS is a set of PM-queries)
$\texttt{lscore}(v, PQ)$	Local score of v w.r.t. PQ
$\texttt{score}(v, SQ)$	Score of v w.r.t. SQ
$Ans(SQ)$	Top-k answer to SQ
$\mathcal{E}(PQ)$	Set of essential variables in PQ
PPQ	Partial PM-query for PQ
$UB(v)$	Upper bound on $\texttt{score}(v, SQ)$
$\#subst(PQ)$	Upper bound on the number of answer substitutions for PQ
$\texttt{cand}(v, \delta, \pi, (l, d))$	Number of vertices that are connected to v via an undirected path of length δ ending with an edge labeled ℓ of type π with direction d
$cand^{ub}(PQ, ?x)$	Upper bound on the number of substitutions for variable $?x$ in PQ
$\texttt{optPPQ}(PQ)$	Chosen partial PM-query for PQ
$\#subst(PPQ, PQ)$	Upper bound on the number of substitutions for the variables in PQ that are not in PPQ
$EQ(PQ)$	Equivalent set of non-annotated "expanded" PM-queries for PQ
$\texttt{cov}(PPQ)$	Subset of $EQ(PQ)$ such that, by extending a substitution for PPQ, full substitutions for all the queries in the subset can be obtained

query extracted from PQ that includes a subset of vertices of PQ (containing all essential variables) and a subset of edges of PQ, as defined below.

Definition 2.2.1 (Partial PM-query). *Given a PM-query* $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \boldsymbol{lagg}\rangle$, *a* partial PM-query *$PPQ$ for PQ is a 5-tuple $PPQ = \langle V_{PPQ}, E_{PPQ}, \chi, \varrho, \boldsymbol{lagg}\rangle$ such that:*

- $\mathcal{E}(PQ) \subseteq V_{PPQ} \subseteq V_{PQ}$*;*
- $V_{PPQ} \cap V \neq \emptyset$*;*
- $E_{PPQ} \subseteq E_{PQ}$ *is such that there exists at least one undirected path between every pair of essential variables in $\mathcal{E}(PQ)$, and at least one undirected path between each essential variable in $\mathcal{E}(PQ)$ and a constant vertex in V_{PPQ}.*[4]

As an example, $?c1 \leftarrow ?b1$ & $?c2 \leftarrow ?b1 \leftarrow$B1 is a partial PM-query for PQ_1 of

[4]It should be observed that these paths *may* include a *subset* of the non-essential variables in PQ.

our running example (we sometimes abuse notation and use this shorthand notation for brevity).

Definition 2.2.2 (Essential Partial Substitution)**.** *Given a PM-query PQ, an es-sential partial substitution θ for PQ is an answer substitution for a partial PM-query of PQ.*

Example 2.2.3. *Consider the PM-queries of our running example, ignoring edge annotations for now. Observe that* ?c1 $\leftarrow$?b1 *&* ?c2 $\leftarrow$?b1 $\leftarrow$*B1 is a partial PM-query for PQ_1, and* ?c1 $\leftarrow$?b1 $\rightarrow$?c2 *&* C8 $\rightarrow$?c1 $\leftarrow$?c2 $\leftarrow$*C7 is a partial PM-query for PQ_2. The essential partial substitutions obtained in this case for PQ_1 and PQ_2 are reported in Table 2.2. Now, each vertex onto which a substitution maps variable* ?b1 *has a chance to be in the top-k answer. This chance may depend on the number of full substitutions for the whole PM-queries PQ_1 and PQ_2 to which an essential partial substitution can be extended after mapping variables which do not belong to the partial PM-queries.*

Table 2.2: Essential partial substitutions for PQ_1 and PQ_2 (without annotations). $\theta_{i,j}$ is the j-th partial substitution for PQ_i.

Substitution	?$b1$	?c1 (?c1.sus)	?c2 (?c2.sus)
$\theta_{1,1}$	B3	C4 (1)	C3 (0)
$\theta_{1,2}$	B3	C2 (1)	C3 (0)
$\theta_{1,3}$	B2	C2 (1)	C1 (1)
$\theta_{1,4}$	B2	C2 (1)	C6 (0)
$\theta_{2,1}$	B2	C6 (0)	C1 (1)
$\theta_{2,2}$	B2	C2 (1)	C1 (1)
$\theta_{2,3}$	B4	C2 (1)	C1 (1)
$\theta_{2,4}$	B3	C3 (0)	C2 (1)
$\theta_{2,5}$	B2	C3 (0)	C2 (1)

2.2.2 Computing Upper Bounds on Scores

Let us assume for now that, given an S-query $SQ = \langle PQS, \texttt{gagg} \rangle$, we have chosen a partial PM-query for each $PQ_i \in PQS$ and we also have at hand all the possible essential partial substitutions for each of the partial PM-queries chosen.

We use these essential partial substitutions to compute upper bounds on local and global scores by creating a *candidate output table* where for each candidate output vertex v, we maintain a list of essential partial substitutions which map an output variable to v, along with the values of the scoring terms obtained from the application of such substitutions.

For instance, in the case of our running example, we create the candidate output table reported in Table 2.3.

Table 2.3: Candidate output table for PQ_1 and PQ_2 (without annotations).

Candidate output	Substitutions	Scoring term values
B2	$\theta_{1,3}, \theta_{1,4}, \theta_{2,1}, \theta_{2,2}, \theta_{2,5}$	5, 2, 4, 5, 4
B3	$\theta_{1,1}, \theta_{1,2}, \theta_{2,4}$	2, 2, 4
B4	$\theta_{2,3}$	4

Now, given a PM-query $PQ_i \in PQS$ with $\texttt{lagg} \in \{MIN, MAX, AVG\}$, we observe that an upper bound on $\texttt{lscore}(v, PQ_i)$ is the maximum among the scoring term values of v provided by all essential partial substitutions for PQ_i that map an output variable to v. Moreover, an upper bound on $\texttt{score}(v, SQ)$, denoted $UB(v)$ in the remainder, can be obtained by using $\texttt{gagg}$ to aggregate the upper bounds on local scores of v.

If $\texttt{lagg} = SUM$, it does not suffice to just include the scoring term value associated with a vertex v by an essential partial substitution θ for PQ_i in the computation of an upper bound on $\texttt{lscore}(v, PQ_i)$. Assume θ is computed for a partial PM-query PPQ_i of PQ_i. The scoring term value provided by θ must be multiplied by *an upper bound on the number of substitutions θ can be extended to after mapping the variables in PQ_i that were not chosen as variables of PPQ_i*. We compute this upper bound as $\#subst(PQ_i\theta)$, where $\#subst$ is a function that, given any PM-query PQ,[5] returns an upper bound on the number of answer substitutions for the query.

Computing Upper Bounds on Substitutions

To quickly compute $\#subst(PQ)$ we employ a structure called *PM_Index*. This index provides a value $\texttt{cand}(v, \delta, \pi, e)$ for each vertex $v \in \mathcal{G}$, distance δ, value $\pi \in dom(type)$, and pair $e = (\ell, d)$, where $\ell \in \mathcal{L}$ and $d \in \{in, out\}$. This value is computed as follows:

- $\texttt{cand}(v, 1, \pi, e)$ is the cardinality of the set of vertices v' such that (i) $v'.type = \pi$ and (ii) $\mathcal{G}$ contains an edge (v, l, v') if $d = out$, an edge (v', l, v) otherwise.
- $\texttt{cand}(v, \delta, \pi, e) = \max\{\texttt{cand}(u, 1, \pi, e) |$ there is an undirected path of length $\delta - 1$ from v to u in $\mathcal{G}\}$.

Intuitively, for a given $(v, 1, \pi, e)$ quadruple (i.e. distance 1), the index contains, for each label ℓ, the number of in- and out-neighbors of v that are connected via an edge labeled ℓ that are of type π. The second part of the definition updates

[5]Observe that $PQ_i\theta$ is a PM-query itself.

this for the case when $\delta > 1$ by recursively building on top of smaller values of δ.[6] Some values of function `cand` for the graph database of our running example are shown in Table 4.1.

Table 2.4: Example values of function `cand`, where $\xleftarrow{\ell}$ stands for (ℓ, in) and $\xrightarrow{\ell}$ for (ℓ, out).

v	δ	π	e	`cand`
B1	1	*bank*	$\xrightarrow{partner}$	2
B1	2	*company*	$\xrightarrow{funded}$	3
C2	1	*company*	$\xleftarrow{partner}$	3
C2	2	*bank*	$\xleftarrow{partner}$	1

Now, for each variable $?x$ in PQ, we can define an upper bound on the number of substitutions for $?x$ as $cand^{ub}(PQ, ?x) = \texttt{cand}(v, \delta, \pi, e)$, where v is a constant vertex of PQ whose distance δ from $?x$ is minimum w.r.t. the other constant vertices of PQ, π is the value of the *type* property of $?x$, and the last edge of a shortest path between v and $?x$ is $(\cdot, \ell, ?x)$ if $e = (\ell, out)$, and $(?x, \ell, \cdot)$ otherwise.[7]

Finally, we can compute $\#subst(PQ)$ by multiplying the values of $cand^{ub}(PQ, ?x)$ for all variables $?x$ in PQ. The following theorem ensures the correctness of this computation. Its proof can be found in the Appendix.

Theorem 2.2.4. *Suppose* $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \texttt{lagg} \rangle$ *is a PM-query. Then* $\#subst(PQ) = \prod_{?x \in V_{PQ} \cap \mathsf{VAR}} cand^{ub}(PQ, ?x)$ *is an upper bound on the number of substitutions for PQ.*

Example 2.2.5. *In our running example, we have* $\#subst(PQ_2) = \prod_{?V \in \{?b1, ?c1, ?c2, ?p1, ?p2\}} cand^{ub}(PQ_2, ?V) = 12$ *because* $cand^{ub}(PQ_2, ?c1) =$

[6] In the worst case, the construction of the $\mathsf{PM\_Index}$ takes $O(|V|^2)$ time and its size is $O(|V|)$. However, our experiments on real datasets will show that actual construction times and index sizes are satisfactory in practice.

[7] We chose to use the shortest path in the definition of $cand^{ub}(PQ, ?x)$ for simplicity. We could have used any other kind of path in the definition — Theorem 2.2.4 can easily be extended to hold anyway.

$\textbf{\textit{cand}}(C8, 1, company, \xrightarrow{partner}) = \mathit{3}$, $cand^{ub}(PQ_2, ?c2) = \textbf{\textit{cand}}(C7, 1, company, \xrightarrow{partner}) = \mathit{2}$, *and* $cand^{ub}(PQ_2, ?b1) = \textbf{\textit{cand}}(C7, 2, bank, \xleftarrow{funded}) = \mathit{2}$, $cand^{ub}(PQ_2, ?p1) = \textbf{\textit{cand}}(C8, 2, person, \xleftarrow{work}) = \mathit{1}$, $cand^{ub}(PQ_2, ?p2) = \textbf{\textit{cand}}(C7, 2, person, \xleftarrow{work}) = \mathit{1}$. *Note that $C7$ is the closest constant from $?c2$, $?p2$ and $?b1$ while $C8$ is the closest one from $?c1$ and $?p1$.*

Later, our experiments will show that this way of computing upper bounds provides reasonably good bounds in practice.

2.2.3 Choosing Partial PM-queries

Given a PM-query PQ, the problem of choosing a partial PM-query $PPQ = \texttt{optPPQ}(PQ)$ for it is that of ensuring that mapping the variables of PPQ against $\mathcal{G}$ results in *maximizing* $\#subst(PQ\theta)$ where θ is an answer substitution for PPQ.[8] In other words, we want to define function $optPPQ$ in such a way that $\#subst(PPQ, PQ) = \prod_{?x \in (V_{PQ} \setminus V_{PPQ}) \cap \mathsf{VAR}} cand^{ub}(PQ, ?x)$, that is the upper bound on the number of substitutions for the variables in PQ that were not chosen as variables of PPQ, is maximized.

We first show that the problem of computing $\texttt{optPPQ}(PQ)$ is NP-hard, then we propose two approaches for addressing it.

To show NP-hardness of $\texttt{optPPQ}(PQ)$, we start by introducing a decisional variant $\texttt{optPPQ}(PQ)^{B}_{log}$ defined as the problem of deciding whether there is a partial

[8]The reason why we want to maximize $\#subst(PPQ, PQ)$ is related to how pruning is applied. Intuitively, during query processing, if we pick the vertices that will possibly get a higher final score, we increase the chances that more vertices will drop below the current threshold. Further details are provided in Section 2.2.4.

PM-query PPQ for PQ such that $\log_2 \#subst(PPQ, PQ) \geq B$.[9] Theorem 2.2.6 characterizes the complexity of this variant. The proof of the theorem can be found in the Appendix.

Theorem 2.2.6. *$\texttt{optPPQ}(PQ)^B_{log}$ is NP-complete.*

Now, it is easy to observe that $\texttt{optPPQ}(PQ)^B_{log}$ can be polynomially decided after solving the original problem $\texttt{optPPQ}(PQ)$. Indeed, $\texttt{optPPQ}(PQ)^B_{log}$ is true iff $\texttt{optPPQ}(PQ)$ returns a partial PM-query PPQ such that $\log_2 \#subst(PPQ, PQ) \geq B$. Thus, $\texttt{optPPQ}(PQ)$ is NP-hard.

To compute $\texttt{optPPQ}(PQ)$ we propose two alternative approaches: a list-oriented pruning method and a reduction to the Node-Weighted Steiner Tree problem.

List-Oriented Pruning

Given a PM-query PQ, a partial PM-query for PQ can be generated by choosing a set $V_s \subset V_{PQ}$, and then checking whether Definition 2.2.1 holds for the partial PM-query induced by the vertices in V_s (denoted $PQ(V_s)$). The list-oriented pruning method starts by building the list of all subsets V_s of V_{PQ} s.t. $|V_s| = |V_{PQ}| - 1$. Then, the list is processed as follows. For each unprocessed subset V_s in the list, if $PQ(V_s)$ satisfies Definition 2.2.1, then the algorithm adds all the subsets of V_s with cardinality $|V_s| - 1$ to the list — otherwise, V_s is immediately pruned from the list. When all the subsets in the list have been processed, the algorithm selects the one

[9]For technical reasons, we use log. It allows us to consider B such that its size is polynomial w.r.t. $|V_{PQ}|$ in the reduction of Theorem 2.2.6.

for which $\#subst(PQ(V_s), PQ)$ is maximum.

It is easy to see that if V_s does not comply with Definition 2.2.1, then no subset of V_s can do: (i) if $\mathcal{E}(PQ) \nsubseteq V_s$, then we have $\mathcal{E}(PQ) \nsubseteq V'_s$ for any $V'_s \subset V_s$; (ii) if a path between the essential variables of PQ, or between an essential variable and a constant vertex, cannot be defined using the vertices in V_s, then no path can be defined using any subset of V_s.

For instance, if we consider PQ_2 of our running example, $V_s = \{\text{C7}, ?b1, ?c2, ?p2\}$ does not comply with Definition 2.2.1 because $?c1$ is a scoring variable, so the algorithm avoids listing all of its subsets.

Reduction to the NWST Problem

The second approach reduces our problem to the Node-Weighted Steiner Tree (NWST) problem and then uses well-known approximate algorithms for NWST in order to compute (sub)optimal partial PM-queries.

The NWST problem is defined as follows. Given an undirected graph $\widehat{G} = \langle \widehat{V}, \widehat{E} \rangle$, a node-weight function $\omega : \widehat{V} \rightarrow \mathbb{N}$, and a subset of vertices $S \subseteq \widehat{V}$ (called *terminals*), compute the minimum weight subtree of $\widehat{G}$ that includes all the vertices in S.

We reduce the problem of computing `optPPQ`(PQ) to an instance $NWST(PQ)$ of NWST as defined below.

Definition 2.2.7 (NWST(PQ))**.** *Given a PM-query $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \texttt{lagg} \rangle$, an instance $NWST(PQ)$ of NWST is defined as follows: (i) $\widehat{V}$ contains the* *same vertices as PQ; (ii) $\widehat{E}$ contains an undirected edge (v_1, v_2) for each edge $(v_1, \{\ell\}, 1, v_2)$ or $(v_2, \{\ell\}, 1, v_1)$ in E_{PQ}; (iii) $S = \mathcal{E}(PQ) \cup (V_{PQ} \cap V)$, i.e., S consists of all the essential variables and constant vertices of PQ; (iv) $\forall v \in S,\ \omega(v) = 0$ and $\forall ?x \in \widehat{V} \setminus S,\ \omega(v) = \ln cand^{ub}(PQ, ?x)$.*

It turns out that solving $NWST(PQ)$ suffices to solve `optPPQ`(PQ).

Theorem 2.2.8. *Given a PM-query PQ, every solution T of $NWST(PQ)$ one-to-one corresponds to a partial PM-query $PPQ(T)$ for PQ such that $\#subst(PPQ(T), PQ)$ is maximum.*

The proof of Theorem 2.2.8 can be found in the Appendix.

Thus, many well-known approximate algorithms for NWST can be used to compute `optPPQ`(PQ). For instance, [33] presents two algorithms with $1.35\,ln\,|S|$ and $1.65\,ln\,|S|$ as the worst case approximation ratios. Interestingly, these approximation ratios do not depend on the size of the graph but only on the number of terminal vertices, which corresponds in our case to the number of constant vertices and essential variables. In our implementation, we used the faster $1.65\,ln\,|S|$ algorithm.

2.2.4 The PScore Algorithm

We conclude the section by showing how the various techniques described in the previous sections fit together in our overall query processing algorithm, named PScore (Algorithm 1).

The algorithm, on Line 3, applies one of the two optimization methods de-

Algorithm 1: PScore

```
1  FUNCTION PScore
   Input: S-query $SQ = \langle PQS, \mathsf{gagg} \rangle$
   Data: Candidate output table $T$
   Output: Anwer Set $Ans$ containing $k$ vertices along with their exact scores
2  foreach $PQ_i \in PQS$ do
3      $PPQ_i \leftarrow \texttt{optPPQ}(PQ_i)$
4      compute all answer substitutions for $PPQ_i$
5  initialize $T$ using the substitutions obtained at Line 4
6  foreach vertex $v$ in $T$ do
7      compute $UB(v)$
8  $H \leftarrow$ the $k$ vertices in $T$ having the higher upper bounds
9  foreach $v \in H$ do
10     compute $\texttt{score}(v, SQ)$
11 $Ans \leftarrow H$
12 while $T$ is not empty do
13     $v \leftarrow$ vertex with maximum $UB(v)$ in $T$
14     remove $v$ from $T$
15     $v_k \leftarrow$ vertex in $Ans$ with minimum score
16     if $UB(v) \leq \texttt{score}(v_k, SQ)$ then
17         return $Ans$
18     compute $\texttt{score}(v, SQ)$
19     if $\texttt{score}(v, SQ) > \texttt{score}(v_k, SQ)$ then
20         replace $v_k$ with $v$ in $Ans$
21 return $Ans$
```

scribed in Section 2.2.3. If the list-oriented pruning method is applied, we call the algorithm PScore_LP; otherwise, we call it PScore_NWST.

The computation of exact scores (Lines 10 and 18) is done by trying to extend every partial substitution θ associated with a vertex in T to a "full" substitution (i.e. by computing an answer substitution for $PQ_i\theta$ if θ is an answer substitution for PPQ_i). To this aim, the algorithm employs the *map* algorithm described in Section 2.3.2.

It should be observed that PScore proceeds in decreasing order of upper bounds because of the way vertices in the table are pruned. If the condition on Line 19 is true, then a vertex v has an exact score that makes it part of the current top-k answer. When this happens, the upper bounds of one or more vertices in the table drop below the current threshold — such vertices are automatically pruned. It can easily be observed that the higher the score of v, the higher the likelihood that a larger set of vertices will be pruned.

Proposition 2.2.1. *Given an S-query $SQ = \langle PQS, \mathtt{gagg}\rangle$, the* PScore *algorithm terminates and correctly computes $Ans(SQ)$. Its worst-case asymptotical time complexity is $O(T_{map} \cdot |V|^M)$ where $M = \max_{PQ \in PQS} |V_{PQ} \cap \mathsf{VAR}|$ and $T_{\mathtt{map}}$ is the complexity of the* `map` *algorithm.*

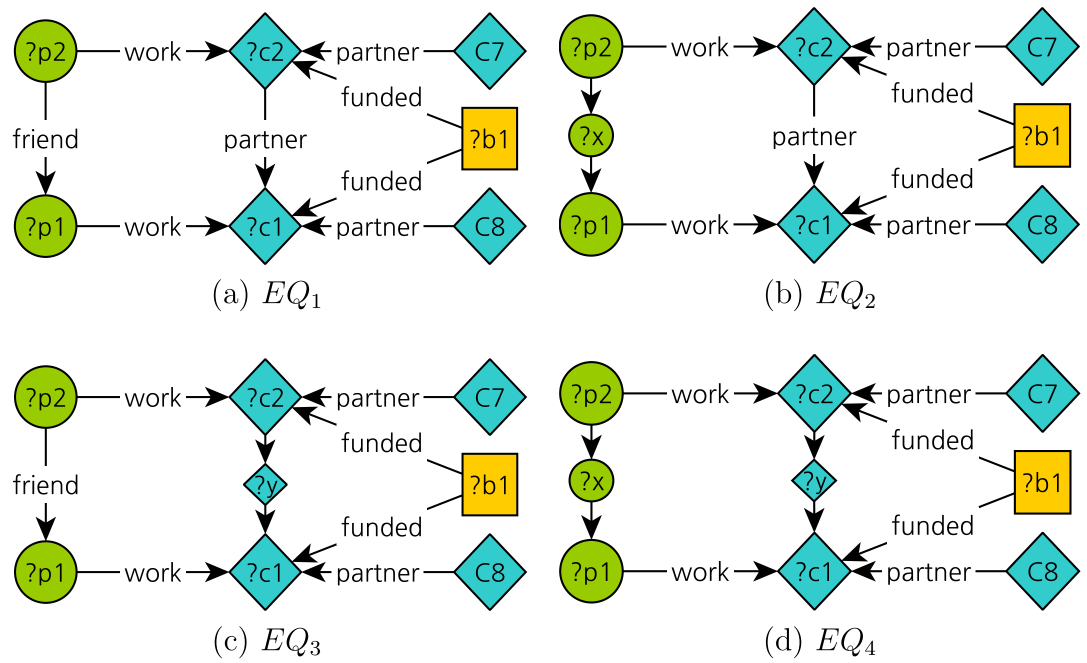
2.3 Managing Edge-Annotated Queries

In this section we discuss how we extend the techniques presented in Section 2.2 to handle the case of edge-annotated PM-queries. In particular, Section 2.3.1 shows our approach to the computation of upper bounds and Section 2.3.2 describes the algorithm for computing substitutions.

2.3.1 Computing Upper Bounds on Scores

The computation of upper bounds for edge-annotated queries can take advantage of specific relationships among partial substitutions that are present in this case.

Consider for instance PQ_2 of our running example. An upper bound for it could be computed by (1) expanding it into an equivalent set $EQ(PQ_2) = \{EQ_1, EQ_2, EQ_3, EQ_4\}$ (shown in Figure 2.3) of non-annotated "expanded" PM-queries for PQ, (2) processing such expanded queries one-by-one, and then (3) taking the sum of the resulting upper bounds (since the set of substitutions for PQ_2 is the union of the sets of substitutions for the expanded queries).



(a) EQ_1 (b) EQ_2

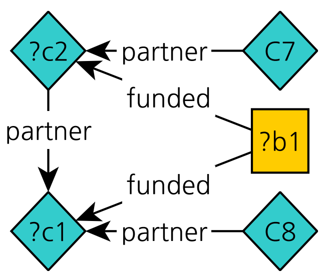
(c) EQ_3 (d) EQ_4

Figure 2.3: Expanded queries for PQ_2 of Figure 2.1.

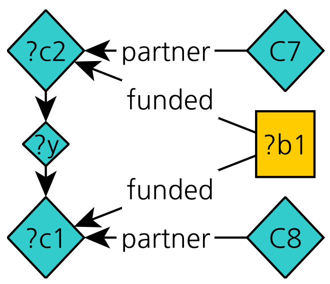
However, this would require processing a partial PM-query for each expanded query, and the number of expanded queries grows very quickly. Consider an edge-annotated PM-query $PQ = \langle V_{PQ}, E_{PQ}, \chi, \varrho, \mathtt{lagg}\rangle$. For each annotated edge $e_i = (v_i, L_i, w_i, v_i')$ in E_{PQ}, let $exp(e_i)$ be the set of edges $\{(v_i, L_i, 1, v_i'), (v_i, L_i, 2, v_i')\}, \ldots, (v_i, L_i, w_i, v_i')\}$. Now, observe that each expanded query in $EQ(PQ)$ contains a different set of edges from $exp(e_1) \times \cdots \times exp(e_n)$ where n is the number of annotated edges in E_{PQ}. Thus, there are $\prod_{i \in [1..n]} w_i$ expanded queries for PQ.

We therefore propose a different approach that heuristically chooses a set of partial PM-queries such that by extending a substitution for one of them, we can obtain full substitutions for more than one expanded query.

Example 2.3.1. *By extending an essential partial substitution for $PPQ_{2,1}$ in Figure 2.4 (resp., $PPQ_{2,2}$), full substitutions for EQ_1 and EQ_3 (resp., EQ_2 and EQ_4) can be built. Now, suppose we compute a substitution θ_1 for $PPQ_{2,1}$. This substitution will not map variables $?p1$ and $?p2$ of EQ_1 and variable $?x$ of EQ_3. The upper bound on the number of substitutions obtainable by extending θ_1 is therefore $\#subst(EQ_1\theta_1) + \#subst(EQ_3\theta_1) = cand^{ub}(EQ_1\theta_1, ?p1) \times cand^{ub}(EQ_1\theta_1, ?p2) + cand^{ub}(EQ_3\theta_1, ?p1) \times cand^{ub}(EQ_3\theta_1, ?p2) \times cand^{ub}(EQ_3\theta_1, ?x)$. The same applies to any substitution θ_2 for $PPQ_{2,2}$: its upper bound is $\#subst(EQ_2\theta_2) + \#subst(EQ_4\theta_2) = cand^{ub}(EQ_2\theta_2, ?p1) \times cand^{ub}(EQ_2\theta_2, ?p2) + cand^{ub}(EQ_4\theta_2, ?p1) \times cand^{ub}(EQ_2\theta_2, ?p2) \times cand^{ub}(EQ_4\theta_2, ?x)$.*



(a) $PPQ_{2,1}$



(b) $PPQ_{2,2}$

Figure 2.4: Two partial PM-queries for PQ_2.

In general, let PQ be a PM-query and PPQ a partial PM-query for it. Moreover, let $\texttt{cov}(PPQ) \subseteq EQ(PQ)$ be the set of expanded queries such that by extending a substitution for PPQ, we obtain full substitutions for all the queries in $\texttt{cov}(PPQ)$ — for instance, in our example we have $\texttt{cov}(PPQ_{2,1}) = \{EQ_1, EQ_3\}$. Given a substitution θ for PPQ, we have $\#subst(PQ\theta) = \sum_{EQ \in \texttt{cov}(PPQ)} \#subst(EQ\theta)$.

We now describe our heuristics to choose a set $\mathcal{Q}$ of partial PM-queries for an edge-annotated PM-query PQ. Obviously, $\mathcal{Q}$ must "fully cover" the set of expanded queries for PQ, that is, it must satisfy $\cup_{PPQ \in \mathcal{Q}} \mathtt{cov}(PPQ) = EQ(PQ)$.

To ensure this, and to possibly provide high-quality partial PM-queries, we first build a modified version PQ' of PQ (by replacing annotated edges with non-annotated ones), then we compute $\mathtt{optPPQ}(PQ')$ and use the result to compute $\mathcal{Q}$.

Set $E_{PQ'}$ is built from E_{PQ} by replacing each annotated edge $e = (v, L, w, v')$ with two edges $\mathtt{mod}_1(e) = (v, L, 1, ?V)$ and $\mathtt{mod}_2(e) = (?V, L, 1, v')$ with $?V$ being an artificial variable not in V_{PQ} – the number of such new edges is $2n$, i.e. twice the number of annotated edges in E_{PQ}. Now, to compute $PPQ' = \mathtt{optPPQ}(PQ')$ we need to choose a good value for $cand^{ub}(PQ', ?V)$ — this value cannot be derived from the PM_Index as $?V$ is an artificial variable we introduce just for dealing with annotated edges. Consider the "longest expansion" of e that consists of the following path of w non-annotated edges: $(v, L, 1, ?V_1), (?V_1, L, 1, ?V_2), \ldots, (?V_{w-1}, L, 1, v')$ with each $?V_i$ being an artificial variable not in V_{PQ}. We define $cand^{ub}(PQ', ?V)$ as $\frac{1}{|EQ(PQ')|} \sum_{i \in [1..w]} \sum_{j \in [1..|EQ(PQ')|]} cand^{ub}(EQ_j, ?V_i)$.

Observe that due to the properties of the problem of choosing optimal partial PM-queries, function $\mathtt{optPPQ}$ will either include both edges $\mathtt{mod}_1(e)$ and $\mathtt{mod}_2(e)$ in PPQ', or none of them.

The final set $\mathcal{Q}$ can then be computed from PPQ' as follows. For each annotated edge $e_i = (v_i, L_i, w_i, v_i')$ in E_{PQ}, we define a set of edges $E_{\mathcal{Q}}(e_i)$ as $exp(e_i)$ if $\mathtt{mod}_1(e_i)$ is in PPQ', and $\emptyset$ otherwise. Now we consider the set $EC = E_{\mathcal{Q}}(e_1) \times \cdot \times E_{\mathcal{Q}}(e_n)$. Set $\mathcal{Q}$ is initialized with $|EC|$ partial PM-queries, containing only the non-annotated edges in E_{PQ}. Each of these queries is then extended with a different set of edges from EC. Observe that $|\mathcal{Q}| = |EC| = \prod_{i \in [1..n], \texttt{mod}_1(e_i) \text{ in } PPQ'} w_i$. It is easy to see that this approach ensures that $\mathcal{Q}$ cover the set of expanded queries for PQ.

For instance, in our running example, if $\texttt{optPPQ}(PQ_2)$ is a partial PM-query of the form $?c1 \leftarrow ?b1 \rightarrow ?c2$ & C8 $\rightarrow$?c1 $\leftarrow$?c2 $\leftarrow$C7, then the final partial PM-queries are those reported in Figure 2.4.

As our experiments will show, this approach chooses good-quality partial PM-queries. Moreover, the structure of the map algorithm presented in Section 2.3.2 allows us to exploit commonalities between the queries in $\mathcal{Q}$. In fact, the actual set of queries map must run over can just include the edges with the maximum-length annotation (i.e., $(v_i, L_i,\ w_i,\ v_i')$ in set $E_{\mathcal{Q}}(e_i)$ above). The number of such queries is $|\{e_i \text{ in } PQ' \mid \texttt{mod}_1(e_i) \text{ in } PPQ'\}|$. In our running example, the input to the map algorithm would be the single partial PM-query $?c1 \leftarrow ?b1 \rightarrow ?c2$ & C8 $\rightarrow$?c1 $\overset{2}{\leftarrow}$?c2 $\leftarrow$C7.

2.3.2 Mapping Algorithm

Algorithm 2 computes all answer substitutions for a given (edge-annotated) PM-query PQ against the graph database $\mathcal{G}$.

The algorithm essentially performs a depth-first search and, when it finds an edge e with an annotation greater than 1 (denoted $\texttt{ann}(e) > 1$) it starts a separate depth-first search using a copy of the query obtained by decreasing the annotation by 1. In the algorithm, S is the set of substitutions being computed. Moreover, the algorithm uses an external function $\texttt{mcand}(u, L, ?v)$ that returns the set of vertices of $\mathcal{G}$ that can be matched to $?v$, i.e., those that are directly connected to u through an edge with a label in L. Note that $|\texttt{mcand}(u, L, ?v)|$ can be retrieved from the PM_Index. Access to graph data on disk is managed through the DOGMA index [13].

conf/semweb/BrochelerPS09

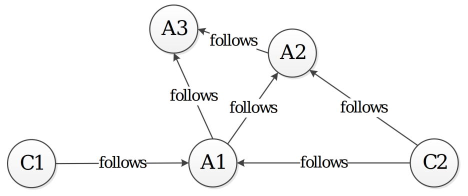
In the algorithm, Lines 5–15 deal with the case of fully mapped edges with an annotation greater than 1, whereas Lines 17–32 deal with unmapped edges. In the first case, given an edge e to process, we create a copy PQ' of PQ with $\texttt{ann}(e) = 1$, and call $\texttt{map}(PQ')$ recursively to continue its mapping (we call this the "Branch" operation). We also create a PQ'' where e is expanded to two edges whose annotations are 1 and $\texttt{ann}(e) - 1$ respectively, and call $\texttt{map}(PQ'')$ recursively ("Spawn" operation). In the second case, we have to select one unmapped edge to find substitutions for (Line 17). We select the edge $(u, L, ?v)$ for which the number of vertices that can be matched to $?v$ is minimized. Then, for each vertex $c \in \texttt{mcand}(u, L, ?v)$, we create PQ' and PQ'' as we did in the first case, and call map recursively.

Subgraph isomorphism is checked on Lines 8 and 23. It should be noted that, until there is no mapped edge with annotation greater than 1, the processing of unmapped edges is deferred. We made this choice to maximize the chance to prune a branch on Line 8. It should also be observed that any SPARQL engine or subgraph matching algorithm could be used to check subgraph isomorphism — in our implementation we used the algorithm proposed for the DOGMA index [13].

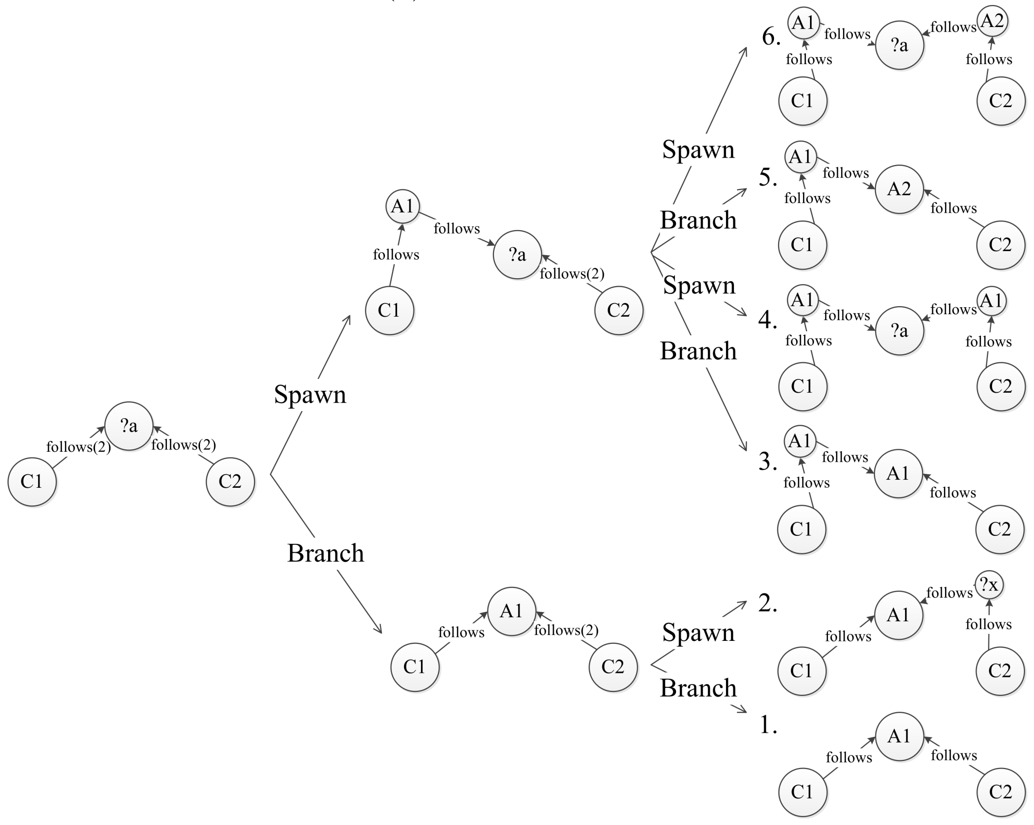
Algorithm 2: `map`

```
FUNCTION map
Input: PM-query $PQ$
Output: answer substitution $S$
$S \leftarrow \emptyset$ if all vars are mapped and all edges' annotations are 1 then
    add the mapping of $PQ$ to $S$
    return $S$
if $\exists(c_1, L, c_2)$ in $PQ$ with ann$(PQ, (c_1, L, c_2)) \geq 2$ then
    $PQ' \leftarrow PQ$
    ann$(PQ', (c_1, L, c_2)) \leftarrow 1$
    if $(c_1, L, c_2)$ matches $\mathcal{G}$ then
        add the result of map$(PQ')$ to $S$
    $PQ'' \leftarrow PQ$ after removing $(c_1, L, c_2)$
    add $(c_1, L, ?x)$ to $PQ''$
    ann$(PQ'', (c_1, L, ?x)) \leftarrow 1$
    add $(?x, L, c_2)$ to $PQ''$
    ann$(PQ'', (?x, L, c_2)) \leftarrow$ ann$(PQ, (c_1, L, c_2)) - 1$
    add the result of map$(PQ'')$ to $S$
    else
        $(u, L, ?v) \leftarrow$ edge that minimizes $|$mcand$(u, L, ?v)|$
        (always prefer edges with annotations equal to 1)
        foreach $c \in$ mcand$(u, L, ?v)$ do
            $PQ' \leftarrow PQ$
            substitute $?v$ with $c$ in all edges of $PQ'$ that contain $?v$
            ann$(PQ', (u, L, c)) \leftarrow 1$
            if all mapped edges of $PQ'$ with ann $= 1$ match $\mathcal{G}$ then
                if $\chi$ is satisfied after substituting $?v$ with $c$ then
                    add the result of map$(PQ')$ to $S$
            if ann$(PQ, (u, L, ?v)) \geq 2$ then
                $PQ'' \leftarrow PQ$ after removing $(u, L, ?v)$
                add $(u, L, c)$ to $PQ''$
                ann$(PQ'', (u, L, c)) \leftarrow 1$
                add $(c, L, ?v)$ to $PQ''$
                ann$(PQ'', (c, L, ?v)) \leftarrow$ ann$(PQ, (u, L, ?v)) - 1$
                add the result of map$(PQ'')$ to $S$
return $S$
```

Example 2.3.2. *Consider the graph database in Fig. 2.5(a) and the PM-query at the left of Fig. 2.5(b). The algorithm starts by processing $C1 \xrightarrow{follows(2)} ?a$. $A1$ is the only neighbor of $C1$. For the branch operation, the algorithm substitutes $?a$ with $A1$. For the spawn operation, it expands the edge and substitutes the intermediate variable* *with A1. Note that expanded edges have annotations equal to 1 and* $\texttt{ann}(C1 \xrightarrow{follows(2)} ?a) - 1 = 1$*, respectively. After branching,* $C2 \xrightarrow{follows(2)} A1$ *is processed. In this case, for the branch operation the algorithm checks whether* $C2$ *and* $A1$ *are connected through* $follows$ *in* $\mathcal{G}$*. If so, we have found an answer substitution (case 1. in the figure). For the spawn operation, an intermediate variable* $?x$ *is added and the mapping continues. The process goes on until all substitutions (such as cases 1. and 5. in the figure) are found.*



(a) Graph database



(b) Branch and spawn operations

Figure 2.5: Example graph database (a) and execution of the `map` algorithm upto the second level of recursion (b).

Proposition 2.3.1. *Given a PM-query PQ, Algorithm 2 terminates and correctly computes all answer substitutions for PQ in time $O(|EQ(PQ)| \cdot |V|^{|V_{PQ} \cap \mathsf{VAR}| + \nu})$, where $\nu = \sum_{(v,S,w,v') \in E_{PQ}} (w - 1)$.*

2.4 Experimental Evaluation

We tested our framework using the 7 graph databases in Table 2.5. For all databases but BSBM, we generated 3K S-queries, each containing 3 PM-queries. In turn, for each PM-query, we started by randomly retrieving a connected subgraph (or subtree) from the database with a fixed number of vertices and edges. We then replaced a fixed portion of the vertices with variables. We generated four different types of queries, whose numbers of vertices, edges, and constant vertices are reported in Table 2.5. For the IMDb database, which is mostly tree-structured, we only built tree-structured queries. For the BSBM dataset, we used a modified version of the 6 test queries in [40], to which we added GROUP BY statements and anchors in order to create aggregate top-k queries.

Table 2.5: Graph databases (top) and query types (bottom). In the table, n_V is the number of vertices in each PM-query ($|V_{PQ}|$), n_E the number of edges ($|E_{PQ}|$), and n_C the number of constant vertices ($|V_{PQ} \cap V|$).

	$\|V\|$	$\|E\|$		$\|V\|$	$\|E\|$
Nuclear [8]	74K	1.1M	Flickr [19]	6.2M	15.2M
CiteSeerX [27]	0.93M	2.9M	BSBM [11]	14.4M	37.2M
IMDb [37]	2.1M	7.7M	Orkut [43]	3.7M	234M
YouTube [25]	4.6M	14.9M			

	Small tree			Small graph			Large tree			Large graph		
	n_V	n_E	n_C	n_V	n_E	n_C	n_V	n_E	n_C	n_V	n_E	n_C
Nuclear	5	4	1	5	8	1	10	9	4	10	15	4
CiteSeerX	5	4	1	5	8	1	10	9	4	10	15	4
IMDb	5	4	2	–	–	–	9	8	4	–	–	–
YouTube	6	5	1	6	10	1	10	9	3	10	15	3
Flickr	5	4	1	5	6	1	10	9	4	10	11	4
Orkut	5	4	2	5	8	2	10	9	5	10	15	5

In order to vary the selectivity of the queries, we used a parameter $\gamma \in \{0.3, 0.7\}$ that represents the percentage of variables with constraints (i.e., variables for which function ϱ of Definition 2.1.1 is defined) — for a variable $?v$ that replaced a vertex u with property $u.p = z$, we wrote a constraint whose form is either $?v.p \geq z$ or $?v.p \leq z$ (if u had multiple numerical properties, we chose one randomly). This way, we also guaranteed that there exists at least one substitution for each query (we actually obtained upto tens of millions of substitutions). Moreover, for each query, we randomly chose a subset of variables and designated them as output variables (upto 50%, with an average of 25%). Functions `lagg` and `gagg` were chosen randomly as well.

We generated queries by starting from a group of 1K S-queries containing only non-annotated PM-queries with single labels on all edges. We then generated 1K annotated S-queries by randomly adding annotations to edges: 60% of edges had $w = 1$, 25% had $w = 2$, 10% had $w = 3$, and 5% had $w = 4$.[10] Finally, we generated 1K annotated multiple-label S-queries by adding further labels on 40% of the edges.

We evaluated the performance of PScore_LP and PScore_NWST with different query types and values of γ. We compared our algorithms with (i) a PScore_Base algorithm which computes all answers to the query, scores them, and selects the top-k,[11] and (ii) two of the most popular RDF query engines, Jena [1, 18, 63] and GraphDB [51, 10] (formerly known as OWLim). We point out that none of these

[10] We recall that w is the integer value associated with annotated edges (Definition 2.1.1). Connections involving more than 4 edges have been shown to have very limited usefulness in practical scenarios [9].

[11] It should be observed that, although the PScore_Base algorithm does not apply any kind of pruning over the set of substitutions, it still benefits from the efficient disk access employed by the `map` algorithm.

systems is specifically targeted at aggregate top-k computation, whereas we apply a specialized approach to the problem — thus, we expected our algorithms to outperform such systems. Moreover, GraphDB strictly adheres to the SPARQL 1.1 standard, where path edges cannot have length constraints — thus, we tested GraphDB by first expanding each annotated PM-query PQ into its equivalent set of non-annotated queries $EQ(PQ)$ as described in Section 2.3.1. Finally, Jena supports path edges with limited path length as we do in our PM-queries, but it suffers from too much table-join overheads.

All the experiments were run on a Xeon 5140 CPU clocked at 2.33GHz, equipped with 16GB RAM and running RedHat Linux.

2.4.1 Results

In a first round of experiments, we measured the query evaluation times obtained with $k = 10$ using annotated single-label queries. The results are reported in Figure 2.6.

The evaluation times we obtained are very satisfactory in general. On all datasets but IMDb, the performance advantage of PScore_Base over GraphDB ranges between 35.8% and 75.8%, with an average of 58.4%. The only cases where PScore_Base and GraphDB show similar performance are those involving tree-structured queries on the IMDb dataset. Jena was not able to complete the query evaluation process in over half the cases (for each query, we fixed a timeout equal to 4 times the query evaluation time of PScore_Base) except for the BSBM

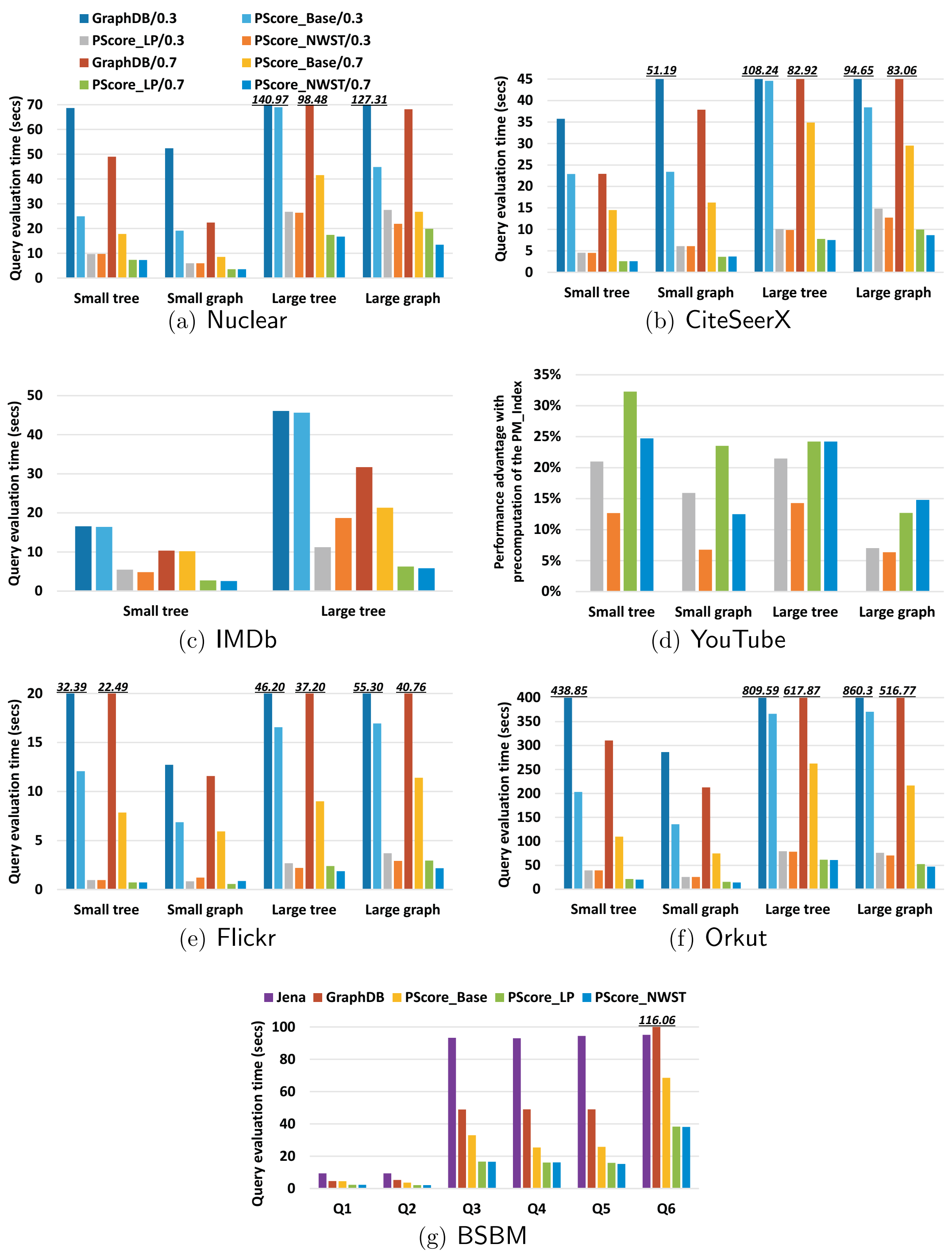


Figure 2.6: Query evaluation time for different algorithms, values of γ, and query types — label A/x means algorithm A with $\gamma = x$.

dataset. The percentage of timeouts w.r.t. the total number of queries of a certain type/size/selectivity (see Table 2.6) was upto 90.5%. Interestingly, the percentage of timeouts decreased of around 20% on average when we used equivalent sets of non-annotated queries.

Table 2.6: Percentage of Jena timeouts.

γ	Small tree	Small graph	Large tree	Large graph
0.3	59.4%	79.4%	76.6%	88%
0.7	61.4%	81%	83.7%	90.5%

In addition, PScore_LP and PScore_NWST largely outperform PScore_Base on all datasets. The performance advantage ranges between 53.7% and 95.8%, with an average of 75.8%. This is mainly a consequence of the much smaller number of full substitutions PScore_LP and PScore_NWST need to compute. Figure 2.7 reports statistics about the average percentage of substitutions they processed w.r.t. PScore_Base during this round of experiments. The results show that both the algorithms save over 60% in the majority of cases, and over 90% in more than 30% of cases.

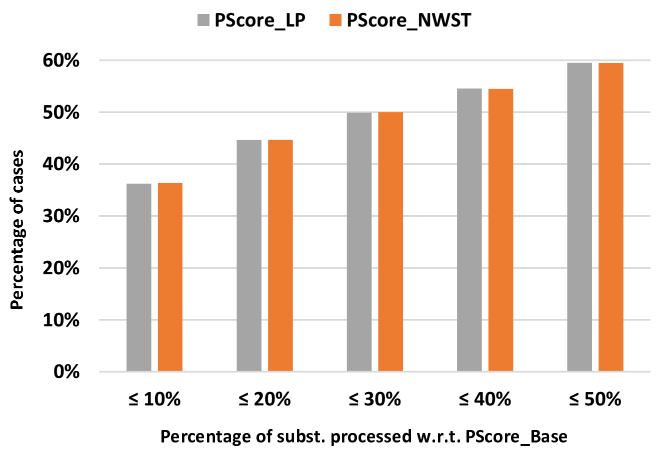
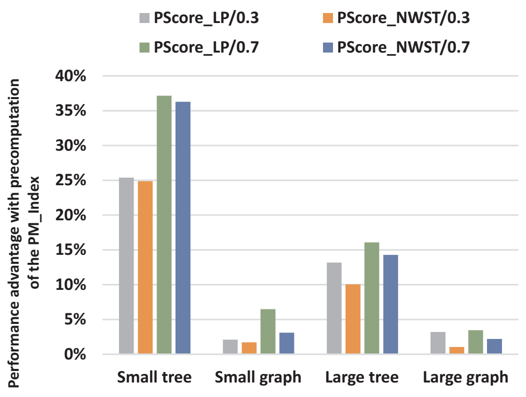


Figure 2.7: Average percentage of substitutions processed by PScore_LP and PScore_NWST w.r.t. PScore_Base.

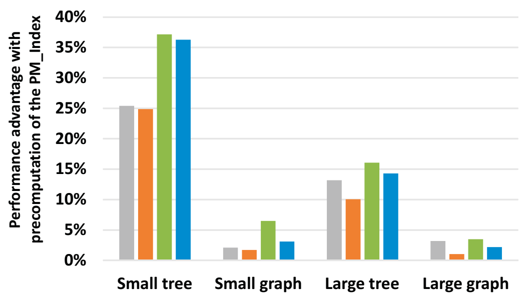
Generally, the relative performance of the PScore_LP and PScore_NWST depends on the query type. In small tree queries, the quality of the partial PM-queries computed by PScore_NWST is very close to that obtained by PScore_LP, while requiring shorter computation times. Thus, PScore_NWST is faster. In small graph queries, the advantage provided by PScore_LP's better partial PM-queries appears to compensate for its longer computation time, and PScore_LP is faster in more than half of the cases. In large tree and large graph queries, PScore_LP takes much longer to compute the partial PM-queries, and the better quality of these PM-queries appears to have less impact than the longer computation times. Thus, PScore_NWST is again the algorithm of choice. Interestingly, in the majority of cases the performance advantage of PScore_NWST over PScore_LP slightly increases (around 2%) when $\gamma = 0.7$.

We also specifically assessed the benefits provided by the use of the PM_Index as well as the quality of the upper bounds computed using the index, as described in Section 2.2.2. Figure 2.8 shows the performance advantage obtained when we precompute the PM_Index (which is the default case in this experimental assessment) w.r.t. the case where the information provided by the index is computed on-the-fly. The results confirm noticeable benefits: the advantage is around 17.5% on average, and more than 30% in some cases on the CiteSeerX, IMDb, YouTube, and Flickr datasets. Figure 2.9 shows that the ratios between upper bounds and real scores of the substitutions processed are satisfactory (under 1.5 in the majority of cases).

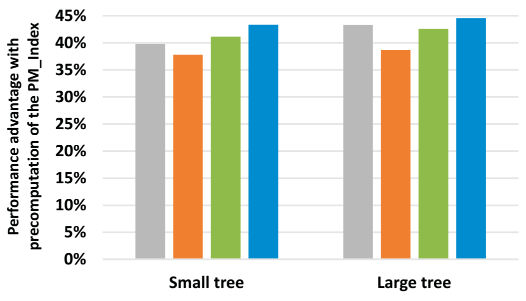
Our second round of experiments (Figure 2.10) assessed how the performance of PScore_NWST and PScore_LP varies with the value of k. We measured query evaluation times obtained for different values of k and γ, using annotated single-



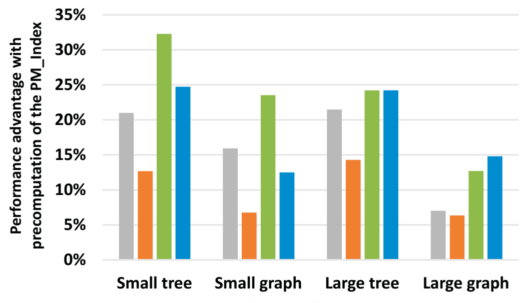
(a) Nuclear



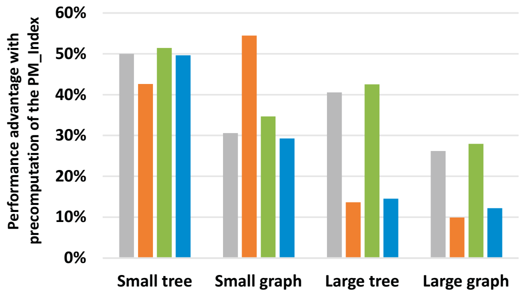
(b) CiteSeerX



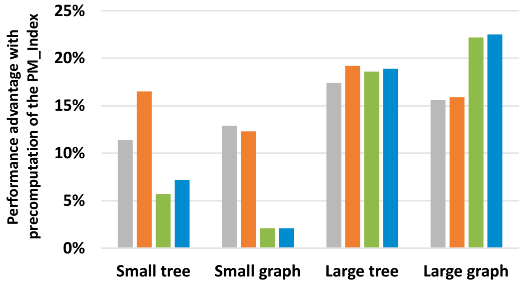
(c) IMDb



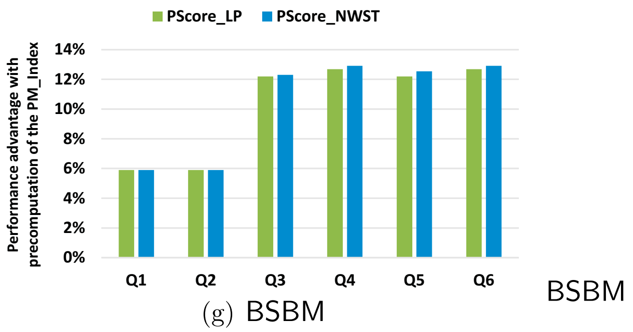
(d) YouTube



(e) Flickr



(f) Orkut



(g) BSBM

Figure 2.8: Performance advantage with precomputation of the PM_Index.

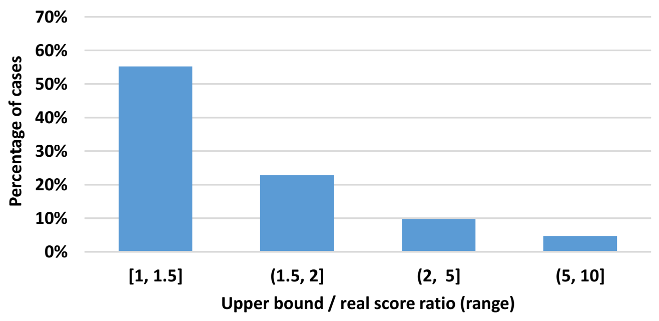


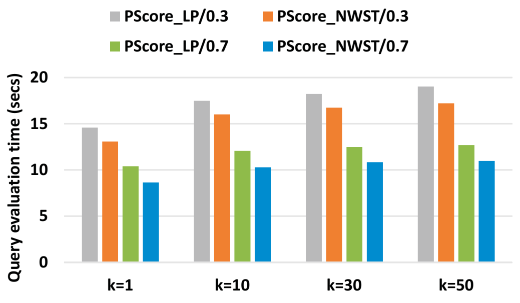
Figure 2.9: Average ratio between upper bounds and real scores of the substitutions processed by PScore_LP and PScore_NWST.

label queries. As expected, since pruning happens later when k increases, evaluation times increase with k. Both our algorithms scale gracefully with k, and PScore_NWST is faster than PScore_LP in the majority of cases with $\gamma = 0.3$ and in all cases with $\gamma = 0.7$. PScore_Base, GraphDB and Jena do not prune, so their times cannot vary significantly w.r.t. k — on average, the performance advantage of PScore_NWST over GraphDB was 85.1%.

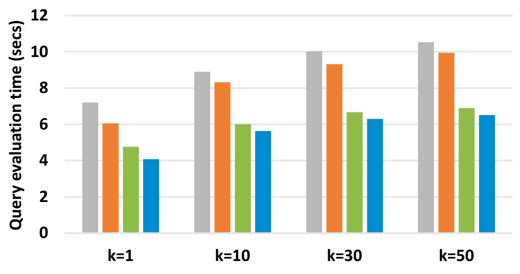
In a third round of experiments, we assessed how the performance of PScore_NWST varies w.r.t. different query groups (non-annotated/single-label, annotated/single-label, annotated/multiple-label). The results we obtained with $k = 10$ on the Flickr dataset (that is the largest dataset with multiple labels for which we built synthetic queries) are reported in Figure 2.11. As expected, the impact of adding annotations and multiple labels is higher when $\gamma = 0.3$, due to the higher number of substitutions the algorithm must handle.

Finally, we also measured the size and the build time of the PM_Index for different values of maximum distance. The results are reported in Table 2.7.[12] Here,

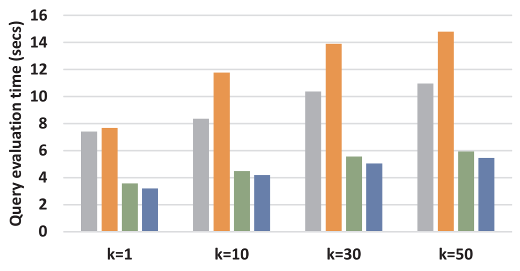
[12] $h \geq 4$ was not needed for the BSBM queries.



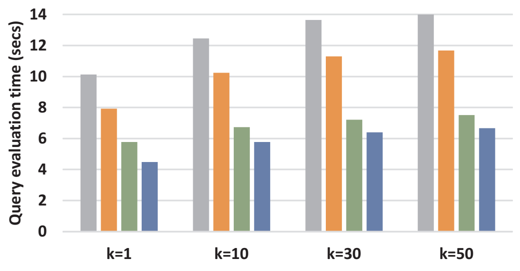
(a) Nuclear



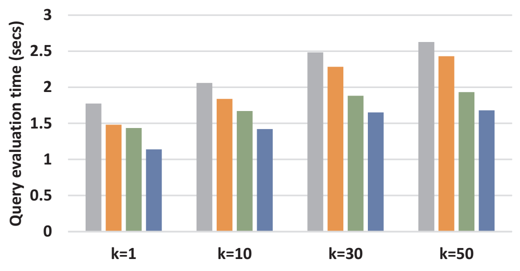
(b) CiteSeerX



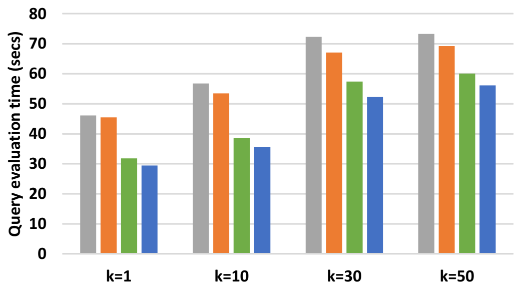
(c) IMDb



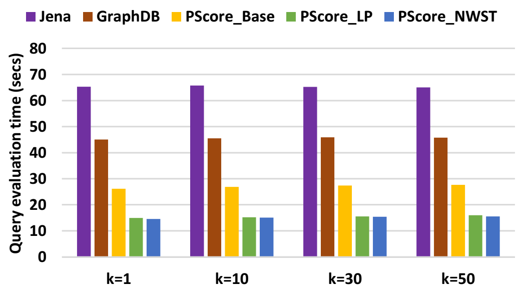
(d) YouTube



(e) Flickr



(f) Orkut



(g) BSBM

Figure 2.10: Query evaluation time for different algorithms and values of k and γ.

we can observe that the size of the index (whose increase w.r.t. distance depends on the average distance between vertices) "stabilizes" at a value of maximum distance around 3. Build times appear satisfactory if we consider the size of the datasets

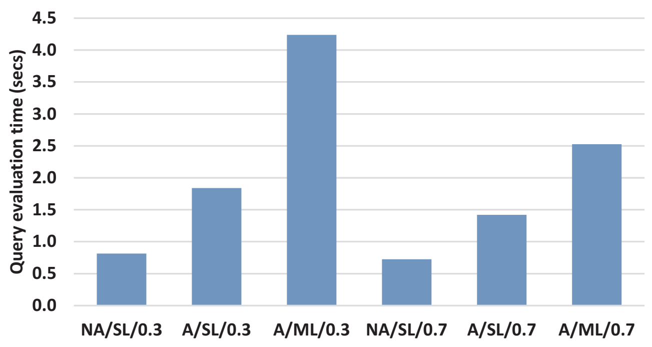


Figure 2.11: Query evaluation time for different query groups and values of γ. Label $X/Y/x$ means query group X/Y with $\gamma = x$. Annotated (resp. non-annotated) queries are denoted as A (resp. NA). Single-label (resp. multiple-label) queries are denoted as SL (resp. ML).

and, more importantly, the fact that the PM_Index (if used) is likely to be built in an offline fashion.

Table 2.7: Size (number of rows) and build time of the PM_Index for different values of maximum distance.

	1	2	3	4	5	Time
Nuclear	0.52M	0.9M	1.29M	1.5M	2.38M	5 mins
CiteSeerX	2.15M	3.94M	5.17M	5.84M	6.08M	8 mins
IMDb	2.51M	8.81M	14.02M	22.31M	24.46M	40 mins
YouTube	5.94M	9.95M	11.25M	11.31M	11.31M	19 mins
Flickr	8.94M	23.60M	35.06M	35.74M	36.04M	59 mins
BSBM	30.33M	109.72M	151.62M	–	–	28 mins
Orkut	24.1M	96.89M	110.01M	112.34M	114.2M	78 mins

Chapter 3: Substitution Importance Queries

In this chapter, we propose another form of importance queries. Unlike the Vertex Importance Query, we find top-k substitution in this chapter, which can be easily expressed in SPARQL using FILTER and ORDER BY clauses. In classical subgraph queries, the user specifies a query subgraph – and all matches of that subgraph with subgraphs of the graph database are considered equally important. However, when the nodes in the graph have associated semantic labels, then there are cases where the user may specify an importance measure that marks some matches as being "more important" than others.

A query for restaurants a person's friends have been to can return hundreds or thousands of answers, as many Facebook users have hundreds of friends. However, users will prefer a short list of the *most relevant* restaurants. We want to provide users a tool to find the most relevant answers from their perspective. An *substitution importance query* is an extended subgraph query with a scoring mechanism. With importance queries users can, e.g. search for restaurants with the *highest star ranking* their friends like or the *largest* cities friends of the parents have been to.

Figure 3.1 shows a Facebook-style graph with four types of edges (friend of, resident of, located in, likes). Each vertex in this graph has different kinds of properties such as the type of the vertex (person, restaurant, city), the age and

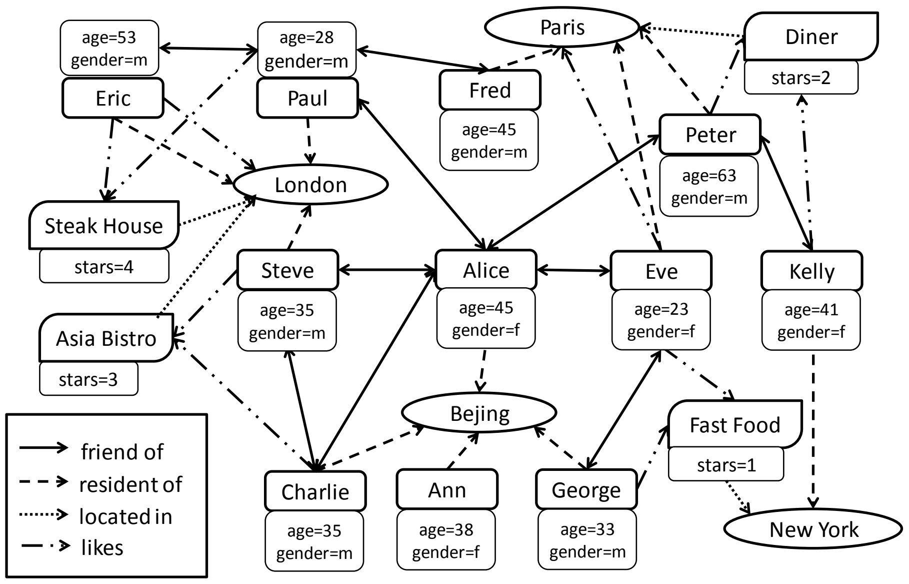


Figure 3.1: Example of a data graph.

gender of persons, and the star rating of restaurants. A possible query on this graph is: Which are the restaurants with the highest star ranking in London that my friends who live in London like?

In this proposal we extend the subgraph matching problem and try to find the *most important* matches (according to a user provided definition) in attributed graphs (i.e. graphs with edge labels and where vertices may have associated properties). We make the following contributions.

- First, we formally define importance queries and define answers (and the top-k answers) to such queries (Section 3.1).

- We then define a simple baseline algorithm to solve such queries, followed by our more sophisticated OptIQ algorithm that can efficiently prune part of the search space and scale our top-k algorithms to find answers to importance queries (Sections 3.2 and 3.3).

- We present the results of experiments to analyze the influence of query properties on the performance of query algorithms (Section 3.4). Our experiments – on CiteSeerX, YouTube, Flickrand GovTrack data, show that our algorithms scale well to data sets containing up to 6.2M vertices and 15.2M edges. We also show that popular triple stores are much slower in answering importance queries.

3.1 Formal Definition of Substitution Importance Queries

In this section, we formalize the concept of *importance queries* – the query type we developed fast answering algorithms for.

Definition 3.1.1 (Importance Query). *An* importance query *is a 4-tuple* $PQ = (SQ, \chi, \varrho, agg)$ *where:*

1. SQ *is a pair* $SQ = (QV, QE)$ *where* $QV \subseteq V \cup \mathsf{VAR}$ *and* $QE \subseteq (V \cup \mathsf{VAR}, \mathcal{L}, V \cup \mathsf{VAR})$. *Because* V *and* VAR *are finite sets,* QV *and* QE *are finite sets as well.* SQ *is called a* subgraph query.
2. χ *associates a constraint that is solely about* $?x$ *with each variable* $?x \in QV \cap \mathsf{VAR}$.[1]
3. ϱ *is a partial function from* $QV \cap VAR$ *to numeric terms s.t. there is at least one* $?x \in QV \cap VAR$ *which is mapped to a numeric term with* $?x$ *occurring in it.*
4. *agg is one of four aggregation function MIN, MAX, SUM or AVG.*[2]

Suppose $SQ = (QV, QE)$ is a subgraph query. A *substitution* is a mapping $\theta : QV \cap \mathsf{VAR} \to V$. Thus, substitutions assign vertices in a GDB $\mathcal{G}$ to variables in QV. The *application* of a substitution θ to a term t, denoted $t\theta$, is the result of replacing all variables $?x$ in t by $\theta(?x)$. When t contains no variables, then $t\theta = t$.

[1] If we do not wish to associate a constraint with a particular variable $?x$, then $\chi(?x)$ can simply be set to a tautologous constraint like $2 = 2$.

[2] These functions map multisets of reals to the reals and are defined in the usual way.

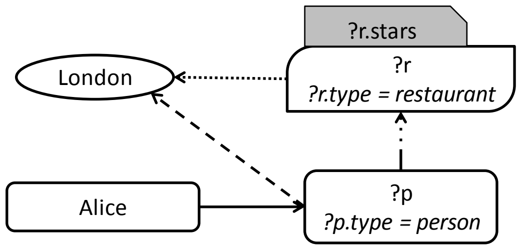


Figure 3.2: Example of an importance query described by a subgraph query, constraints (italic) and an IQ-term (gray box).

If we consider the sample query Q shown in Figure 3.2, it has two answers w.r.t. the graph database shown in Figure 3.1:

$$\theta_1 \equiv ?p = Steve, ?r = AsiaBistro$$

$$\theta_2 \equiv ?p = Paul, ?r = SteakHouse$$

Definition 3.1.2 (Answer; Answer Value)**.** *Suppose $\mathcal{G}$ is a GDB, $PQ = (SQ, \chi, \varrho, agg)$ is an importance query, and θ is a substitution w.r.t. SQ. θ is an* answer *of PQ w.r.t. $\mathcal{G}$ if:*
(i) for every edge $(v_1, ep, v_2) \in QE$, it is the case that $(v_1\theta, ep, v_2\theta) \in E$ and
(ii) for each vertex $?x \in QV \cap \mathsf{VAR}$, the constraint $\chi(?x)\theta$ is true.

The answer value *of a substitution θ, denoted $\mathsf{Aval}(\theta, PQ, \mathcal{G}) = agg(\{(\varrho(?x))\theta \mid ?x \in dom(\varrho)\})$. When the set on the right hand side is empty, $\mathsf{Aval}(\theta, PQ, \mathcal{G}) = 0$.*

We use $\mathsf{ANS}(PQ, \mathcal{G})$ to denote the set of all answers of importance query PQ w.r.t. GDB $\mathcal{G}$.

In our example ϱ assigns the very simple IQ-term $?r.stars$ to the variable $?r$. So the answer value of θ_1 is 3 and the answer value of θ_2 is 4.

3.2 Baseline Best Answer Algorithm

In Section 3.1 we defined importance queries. Depending on the size of the data graph and the constrained IQ-query, the number of results can be very large. We defined the notion of importance queries as users are usually only interested in the most important query answers. Consequently, we will discuss top-k query answer algorithms.

A straightforward algorithm to compute the answers of an importance query follows the definition of importance queries and first computes all subgraph query answers, filters the set of answers to those answers that satisfy the constraints and then computes the IQ-values. Any subgraph matching algorithm could be used here. However, we use an implementation that considers our specific problem situation (queries with anchors and large disk-residing graphs). Subgraph matching algorithms are branch-and-bound algorithms that follow a search tree. In our case, first, an anchor is selected. Then an unmapped neighbor of an anchor or a mapped variable in the query graph gets selected, and all candidates for this variable in the data graph are determined. For every candidate the variable is mapped to the candidate, and the search with the next unmapped variable is continued recursively. We only use the I/O- efficient pruning on vertex degrees because determining vertex degrees does not require one to read extra data. Loading index data for advanced indexes from disk usually does not pay off.

3.3 Optimized (OptIQ) Algorithm

The baseline algorithm (Sec. 3.2) performs the 4 steps (1) Subgraph Matching, (2) Constraint Checking, (3) Scoring and (4) Top-k Selection sequentially and independently. An obvious improvement is the integration of the uncoupled steps. If we check the constraints in the subgraph matching step, then we do not have to create

Algorithm 3: Optimized Importance Query (OptIQ) Algorithm

```
1  FUNCTION AnswerQuery
   Input: Data Graph $G$, Importance Query $q = (SQ = (QV, QE), \chi, \varrho, agg)$, partial
          substitution $\theta$, result size $k$
   Output: answered stored in global variable $A$: set of tuples $(vertex, score)$
2  if $\theta$ maps every variable to a ground term then
3      $A \leftarrow A \cup \{\theta\}$
4      if $|A| > k$ then
5          $A \leftarrow A \setminus \{\theta \in A$ with minimal score($\theta$) $\}$
6  $nextvars \leftarrow \{(c, ?v) | ?v \rightarrow c$ or $c \rightarrow ?v \in QE\}$ //edges with one mapped endpoint
7  foreach $(c, ?v) \in nextvars$ do
8      $R_{c,?v} \leftarrow$ getNeighborNum($G$, $c$, getEdgeLabel($(c, ?v)$))
9      $B_{c,?v} \leftarrow$ getExpBenefit($G$, $q$, $(c, ?v), \theta$) // for WCOST only
10     if $R_{c,?v} = 0$ then return
11 $(c, ?w) \leftarrow (c, ?v) \in nextvars$ with max $B_{c,?v}$ // for WCOST
12                                         with min $R_{c,?v}$ // otherwise
13 $N_{?w} \leftarrow$GetValidNeighbors($G$, $q$, $(c, ?w)$)
14 foreach $m \in N_{?w}$ in decreasing order of score $\varrho(m)$ do
15     $\theta' \leftarrow \theta \cup (?w \rightarrow m)$
16     $s \leftarrow$ calculateMaxScore ($G$, $\theta'$)
17     if $|A| > k$ and $s <$ lowest score of any $\theta \in A$ then continue
18     AnswerQuery($G$, $q\theta'$,$\theta'$,$k$)
19 FUNCTION GetValidNeighbors
   Input: Data Graph $G$, query $q$, tuple (vertex $c$, variable $?w$)
   Output: vertices that can be mapped to $?w$ among all $c$'s neighbors
20 FUNCTION getNeighborNum
   Input: Data Graph $G$, vertex $c$, edge label $l$
   Output: The number of $c$'s neighbors which are connected through an edge of the
           label $l$
21 FUNCTION getExpBenefit
   Input: Data Graph $G$, query $q$, tuple (vertex $c$, variable $?w$), partial mapping $\theta$
   Output: getExpScore ($G$, $q$, $(c, ?w), \theta$) / getCost ($G$, $(c, ?w)$)
22 FUNCTION getExpScore
   Input: Data Graph $G$, query $q$, tuple (vertex $c$, variable $?w$), partial mapping $\theta$
   Output: $agg(\{?v \in QV \cap VAR : value(?v)\})$, where $value(?v) = \varrho(?v)\theta$ if $\theta$ maps
           $?v$, $value(?v) =$ localAvg($?v$) if all candidates for $?v$ are in a known,
           cached subgraph of the subgraph index of $G$, and $value(?v) = 0$ otherwise
23 FUNCTION getCost
   Input: Data Graph $G$, tuple (vertex $c$, variable $?w$)
   Output: $n \log n$, where $n$=getNeighborNum ($G$, $(c, ?w)$), i.e. sorting time in l. 14
24 FUNCTION calculateMaxScore
   Input: Data Graph $G$, partial mapping $\theta$
   Output: $agg(\{?v \in QV \cap VAR : value(?v)\})$, where $value(?v) = \varrho(?v)\theta$ if $\theta$ maps
           $?v$, $value(?v) =$ localMax($?v$) if all candidates for $?v$ are in a known,
           cached subgraph of the subgraph index of $G$, and
           $value(?v) =$ globalMax($?v$) otherwise
```

a possibly large list of subgraph matches that needs to be checked for meeting the constraints. Likewise, we can maintain a sorted list of top-k substitutions. Every time a new substitution with a score greater than the lowest score in the top-k list has been identified, we update the list.

Algorithm 3 shows the integrated algorithm. All blue code segments are extensions to improve the performance, but are not necessary to compute answers to IQ-queries *per se*. We will discuss these improvements in Sections 3.3.2–3.3.4.

Lines 2–5 check whether a complete substitution has been generated, and add a complete substitution to the answer set if its score is among the top-k. Lines 6–10 inspect every edge of the query graph whose one end is mapped to a vertex of the data graph and whose other end is not. In $R_{c,?v}$, we store the number of c's neighbors in the data graph that are connected through an edge with the same label between c and $?v$, i.e. $R_{c,?v}$ is the number of candidates for $?v$. In line 11 we select the query graph edge with the lowest number of candidates. `GetValidNeighbors`() returns the set of all valid vertices that can be mapped to $?w$. Here, we use DOGMA's pruning technique based on IPD values to filter neighbors that cannot be part of a valid answer. Other pruning strategies (see e.g. [62, 28, 68]) could be used as well. In line 14–18, we substitute $?w$ with each candidate m and recursively continue the assembly of answers.

Before we can discuss the performance improving techniques shown in the blue code segments of Algorithm 3, we need to introduce our graph database index.

3.3.1 Database Index

To efficiently answer importance queries on large graphs, we use a disk-based index inspired by the DOGMA index [13]. We decompose the data graph into a large number of small, densely connected subgraphs and store them in an index. Our partitioning algorithm follows the multi-level graph partitioning scheme [35]. Like DOGMA, we iteratively halve the number of vertices by merging randomly selected vertices with all of their neighbors. When the resulting graph has less than 100 vertices, we iteratively expand the graph using the GGGP algorithm from the METIS algorithm package [31] to bisect the graph components at each level.[3] For every block of the partition we extract the subgraph it induces from the graph database and stores it as one block of data to disk.

The objective of the DOGMA index is two-fold: (1) to increase the I/O-efficiency by exploiting data locality – only those parts of the graph that are necessary to answer a query have to be retrieved from disk (2) DOGMA stores for every vertex the internal partition distance (IPD), i.e. the number of hops from a vertex to the nearest other vertex outside the subgraph. Using the IPD, we can quickly compute a minimum distance between vertices, and prune candidates if their distance is higher than the distance between their respective query graph variables.

We extend this concept and store additional information in the index for the advanced pruning strategies. First, we store global maximum values for every vertex property. Additionally, we store together with each induced subgraph G_s the

[3]We also conducted preliminary experiments with other partitioning algorithms but they showed no significant difference for the query processing performance.

edges that connect it to other subgraphs (inter-subgraph edges) and aggregated information (maximum and average) of the predicate values of the vertices in G_s and of those vertices not in G_s but adjacent to a vertex in G_s (denoted as the set of boundary vertices B_s).

3.3.2 Simple Top-k Pruning on Scores

The optimized baseline algorithm does not exploit the fact that we are only interested in the top-k answers. During the stepwise assembly of substitutions, there will be partial substitutions which cannot make it into the top-k given the scores of the full substitutions that are already in the answer set. If we identify them, we can prune the respective branch of the search tree and save computation time.

First, the set $N_{?w}$ should be sorted by score in line 14. I.e., if $?w$ is scored by an IQ-term, $N_{?w}$ is sorted in decreasing order of the value of the term. This ensures that we evaluate the most promising candidates first.

The IQ-score of a substitution is the value of the aggregation function agg on the values assigned by the IQ-terms to the variables (see Def. 3.1.2). For a partial substitution θ, we can compute an upper bound of its answer value Aval by using upper bounds for $\varrho(?v)\theta$ of all unmapped variables. That means, we calculate an upper bound of the answer value by using the exact term score for every previously mapped variable and upper bounds for currently unmapped variables. This is performed by `calculateMaxScore`(). Our simple top-k pruning strategy uses precomputed global upper bounds, i.e. $max_{x \in V} \wp(x, p_i)$, for each vertex property p_i.

When the variance of vertex property values is high, using the global upper bound of a vertex property will not allow us to prune many branches of the search tree. A tighter upper bound is desirable. The mappings of the partial substitutions restrict the set of valid candidates for the currently unmapped variables. What we need is a fast way to find tight upper bounds for vertex property values given the mappings in the partial substitutions.

3.3.3 Advanced Top-k Pruning on Scores

In Section 3.3.2 we presented a simple top-k pruning strategy using upper bounds for the reachable substitution scores. Using the proposed database index, we can find tighter upper bounds that provide a higher pruning power.

For the candidate set $N_{?v}$ of $?v$, we can compute the upper bound for $\varrho(?v)$ using $max_{x \in N_{?v}} \wp(x, p_i)$. But computing the upper bound in this way would require us to read the property scores of all vertices in $N_{?v}$ which is prohibitively expensive. However, if we store the maximal property scores of a subgraph in the index, we can find a good upper bound in a reasonable amount of time.

In `calculateMaxScore`(), we compute the upper bound of the answer value of a partial mapping θ by computing the upper bound for each variable $?v$'s $\varrho(?v)$ (denoted as `value`()). If θ maps a variable $?v$ to a vertex of the data graph, we know the exact value of $\varrho(?v)\theta$. For currently unmapped $?v$'s, we look at its distance to already mapped variables c, $dist(c, ?v)$. If $dist(c, ?v) < IPD(c)$ for some c, we know that $?v$ has to be mapped to the same subgraph as c. Then, we use `localMax` to compute `value` using the local maximum values of the subgraph of c. If $dist(c, ?v) < IPD(c) + 1$, we do the same but using the maximum values of the subgraph and its boundary. However, if $dist(c, ?v) > IPD(c) + 1$ for all c, we have no local information and `globalMax` computes `value` using global maximum values.

3.3.4 Processing Order

The baseline algorithm iteratively selects the unmapped variable with the smallest candidate set for processing. However, for importance queries this strategy sometimes leads to the late discovery of top-k answers. Selecting a variable with a higher number of candidates might not be bad when most candidates can be pruned very early. To weigh the different objectives (low number of branches to follow, following more promising paths first) we compute the benefit score $B_{c,?v}$ in line 9 and process candidates in decreasing order of their benefits. We define the benefit of substituting a variable $?w$ with n candidates in a partial substitution θ' as $wexp(\theta')/f(n)$, where $f(n)$ is the cost to process n candidates and $wexp(\theta')$ is the expected score of θ'. In Algorithm 3, `getExpBenefit`() calculates this score.

As in the case of computing upper bounds for substitution scores for pruning (Section 3.3.3), we compute $wexp(\theta')$ with the precomputed property scores of subgraphs. But additionally we weigh the expected term scores using the indegree of a candidate. The indegree is a simple heuristic for the probability that the variable will be mapped to a vertex. As I/O-efficiency is the primary problem of our algorithm, we use only information already read from disk to determine the expected score. Unavailable vertex property score estimates are replaced by 0.

To compute the expected value we proceed as follows. We classify unmapped variables in the query graph in two groups.

- If a variable $?v$ has no vertex c whose $hop(?v, c)$ is less than or equal to c's IPD value, we assume the property scores are 0 (or ∞ if the MIN aggregation is used). Computing an expected value would require reading many additional disk pages (which we want to avoid) or using global averages. But underestimating the real expected score is in this case favorable because it puts variables whose mapping requires additional disk access at the end of the priority list.

- Otherwise, we use the weight expected value of the subgraph c is residing in. We know that all query variables whose distance from c is less than or equal to c's IPD value will be mapped in the same subgraph. So, we can use the precomputed weighted average property values of the subgraph as the expected value.

3.4 Experiments

In the following, we present an evaluation of the previously introduced top-k algorithms. We conducted experiments with 5 algorithm variants: the non-integrated baseline algorithm Base, the optimized importance query (OptIQ), the extension of OptIQ by simple top-k pruning (GMax), the extension of OptIQ by advanced top-k pruning (LMax), and the extension of LMax by the improved processing order (WCOST).

To see how our algorithms perform in relation to triple stores, we ran additional experiments using Apache Jena TDB 2.10.0 [1] and OWLIM-SE 5.3.5925 [39]. We considered using RDF-3x [47] as well, but had to omit RDF-3x because it cannot answer queries with cross-products because of a bug that still exists in the latest

Name	#Vertices	#Edges	#V.Prop.	#E.Labels
CiteSeerX	0.93M	2.9M	5	4
YouTube	4.6M	14.9M	8	3
Flickr	6.2M	15.2M	4	3
GovTrack	120K	1.1M	5	6

Table 3.1: Evaluation datasets

release. Importance queries can be easily written in SPARQL with its FILTER and ORDER BY clauses. To evaluate our algorithms, we use four real-world datasets. Basic properties of these datasets are shown in Table 3.1.

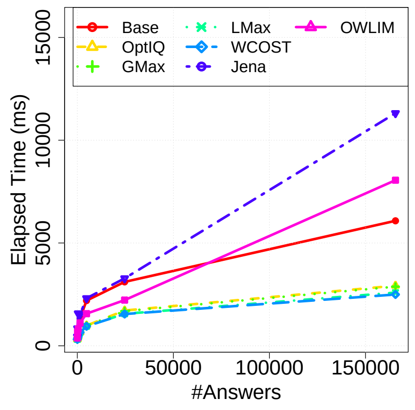
We analyze the performance of the algorithms with randomly generated importance queries. We created the queries by selecting random subgraphs of the data graph with n vertices and m edges. Random subgraphs are created by starting with a random vertex of the data graph. We iteratively add a randomly selected vertex from the neighborhood of any previously selected vertices. From the random subgraphs we created IQ-queries in the following way. We randomly selected c vertices of the subgraph, defining them as anchors, and mapped to the respective vertices of the data graph. The remaining $n - c$ vertices of the randomly selected subgraph are defined as variables. The edges (including the edge labels) of the subgraph are edges in the query. With a probability p, a constraint is created from a numeric property of a vertex in the random subgraph. With a equal probability a constraint is a $>$ or $<$ constraint. The reference value of a $>$ constraint is the property value in the subgraph - 1, and the reference value of a $<$ constraint is the property value in the subgraph + 1. Scoring terms are created similarly to constraints. With a probability t, a numeric property of a vertex is select to be included in an IQ-term. If an IQ-term consists of more than one property, the properties are concatenated with a +. The aggregation function is MAX or SUM with equal probability.

This query generation process ensures that all queries have at least one solution (which is the random subgraph the query has been generated from) and that (in probability) the distribution of structural patterns and properties used in constraints and query terms in a set of random queries resembles the respective distributions in the data graph.

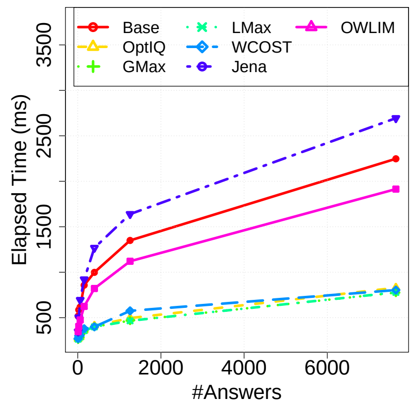
3.4.1 Experimental Results

We evaluated our system by the selectivity of a query (i.e. the number of answers a query has), the size of the query (i.e. the number of vertices and edges the subgraph query has) and the number of desired answers. We used a set of 1000 random queries for the experiments.

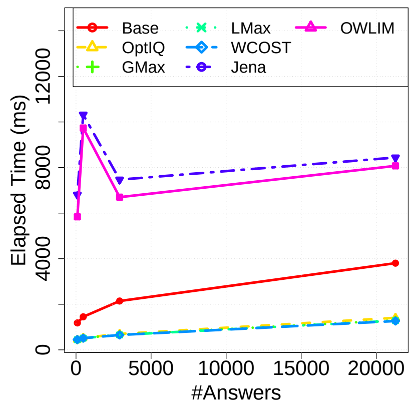
Results by selectivity Figure 3.3 shows the runtime in relation to the answer size of the subgraph query. All algorithms show a sub-linear increase in the runtime with an increasing answer size. Reading subgraphs from disk is a dominating factor of the total runtime. The number of answers to the subgraph query increases much faster than the required number of subgraph reads because usually many answers lie in the same subgraphs. Compared to the baseline more sophisticated algorithms like WCOST and LMax can receive good speed-ups in some but not all settings - especially when the answer size is high. For non-selective queries our algorithms are up to one order of magnitude faster than the evaluated triple stores. For some datasets (Flickr, GovTrack) triple stores perform considerably worse even for very selective queries.



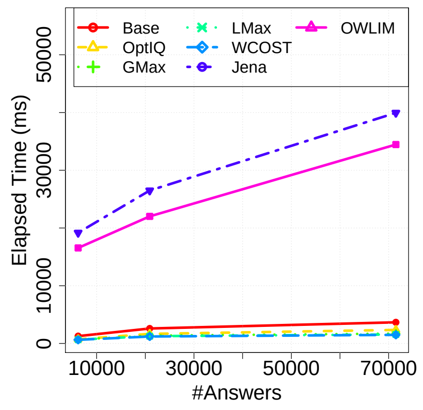
(a) CiteSeerX(10,9,4)



(b) YouTube(10,9,3)



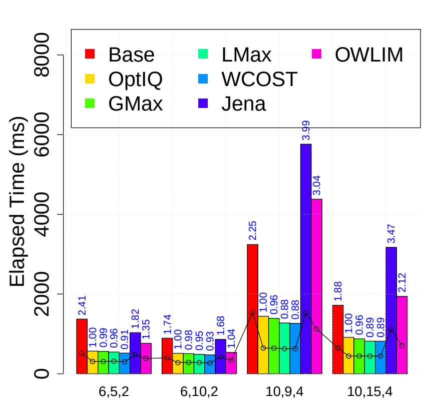
(c) Flickr(10,9,4)



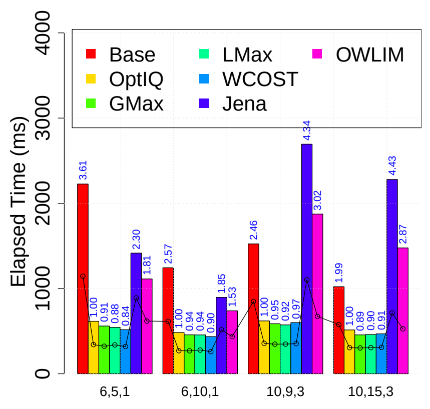
(d) GovTrack(6,5,3)

Figure 3.3: Results by selectivity. Each caption shows the query size, e.g. 6,5,2 means 6 vertices, 5 edges, and 2 anchors.

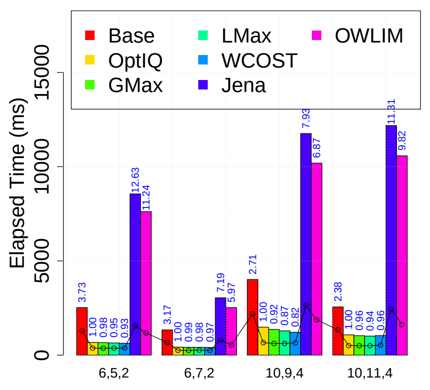
Results by subgraph query size For experiments on the subgraph query size, we use 2 pairs of query types (1000 random queries each) where each pair differs in the number of edges. The results of Figure 3.4 show that our algorithms scale very well in the number of vertices. As we increase the number of edges in a query, the runtime usually decreases as the query gets more selective (i.e. has fewer answers). Once again we see that our algorithms perform much better than the triple stores, and the performance difference is especially high for complex queries.



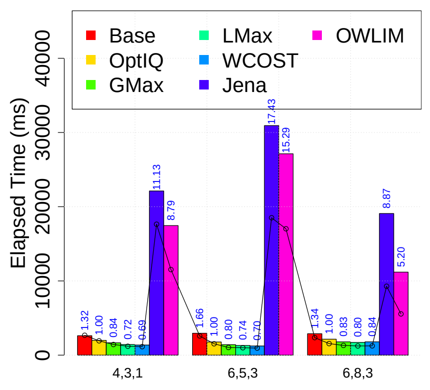
(a) CiteSeerX



(b) YouTube



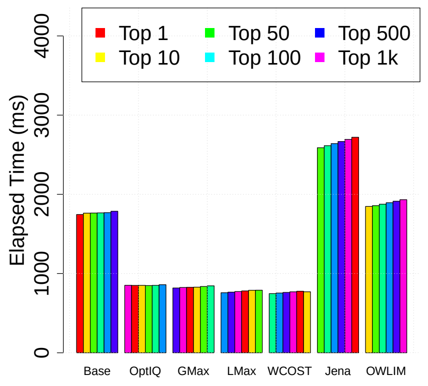
(c) Flickr



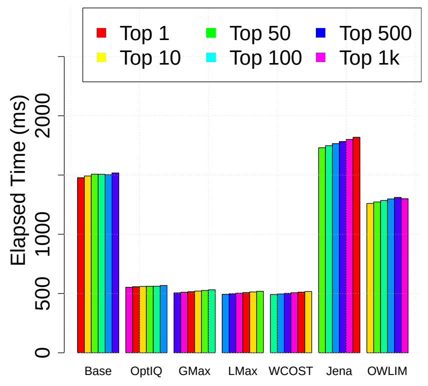
(d) GovTrack

Figure 3.4: Results by query size (Top 10, bar: mean, line: median).

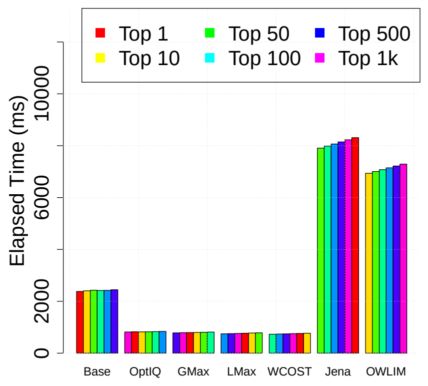
Results by parameter k We analyzed the impact of the desired number of answers on runtime. Overall the scaling of our algorithms with respect to the number of answers is very good (see Figure 3.5), and is much less marked than for the triple stores. The almost constant runtime of our algorithms for low values of k is once again the result of the domination of the total runtime by the time needed to read a subgraph from disk. When most subgraphs in the neighborhood of the anchors have to be read to find the top-1 answer then the time to create a few additional solutions is low.



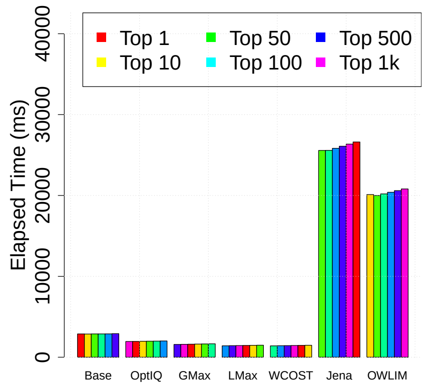
(a) CiteSeerX



(b) YouTube



(c) Flickr



(d) GovTrack

Figure 3.5: Results by parameter k of top-k queries.

Chapter 4: Approximate Importance Queries

Existing subgraph queries largely require "exact" matches. However, in the real-world, a user may not know the data well enough to pose a query that exactly matches his needs. (i) A non-CS user may not know or care how the graph data is exactly stored. For instance, Figure 4.1 shows an example IMDb RDF graph DB. If a movie studio wants to identify potential movies that are sequels to certain high-revenue movies by popular directors, they may not know that specifying a direct link between two movies may be a too restrictive constraint. Thus, a notion of a partial match is needed. (ii) In addition, the user may get a very large number of answers to his subgraph query — he may prefer to only see a list of size k for some fixed k and of course, he would like these to reflect the "best" k answers to his query. (iii) The user may want to define how to score answer quality so that it meets his specific needs, rather than have the DB tell him that certain answers are better than others without taking the user's needs, mission, and objectives into account.

Finding top-k answers to queries, solving (iii) above, has been done in the past [65, 53, 2]. However, none of these approaches takes the semantics of nodes and edges into account, and none of them allows inexact or "approximate" subgraph matches where only part of the query constraints are satisfied.

In the RDF graph DB Figure 4.1, blue vertices are movies and orange vertices are directors. Movie-movie edges tell us which movies are similar to which other movies, while Director-movie edges tells us who directed a given movie. Edges are annotated with strength values ranging from 0 to 1. The strength of an edge (u, v) can be calculated in many ways. For instance, if we consider an edge such as (M1,M4) between two movies, the strength of this edge might be captured by the similarity between the two movies, which in turn could be defined as the inverse of the Euclidean distance between a vector of property values of M1 and a vector of property values of M4. Alternatively, the strength of the edge (D1,M6) might be captured as the number of movies "similar" to M6 that director D1 has directed. These are just two examples. Many others are possible.[1]

A movie studio is looking for movies $?m0$ (see Figure 4.2 (a)), with the property `genre=`Action, which is connected to two movies with revenue over \$10M via at most 2-hop "related" edges. Moreover, $?m0$ should be made by a director $?d$ who has made at least 5 movies. The movie studio considers such a movie and the director as a possible candidate for a new sequel to be produced by them.

Such a subgraph query might match several subgraphs. Figure 4.2(b), (c), (d) show three possible matching subgraphs of the IMDb graph DB. In Figure 4.2(b), the answer substitution returned is $\theta = \{?d/\text{D1},\ ?m0/\text{M1},\ ?m1/\text{M2},\ ?m2/\text{M4}\}$. All edges in the subgraph query are exactly matched by an edge in the graph DB. But in the answer shown in Figure 4.2(c) and (d), some of the constraints in the query

[1]RDF does not allow edges to have properties (other than the relationship captured by the edge) but this is easily achieved by adding a dummy vertex representing strength for each edge or similar graph models such as Neo4j's *property graph*.

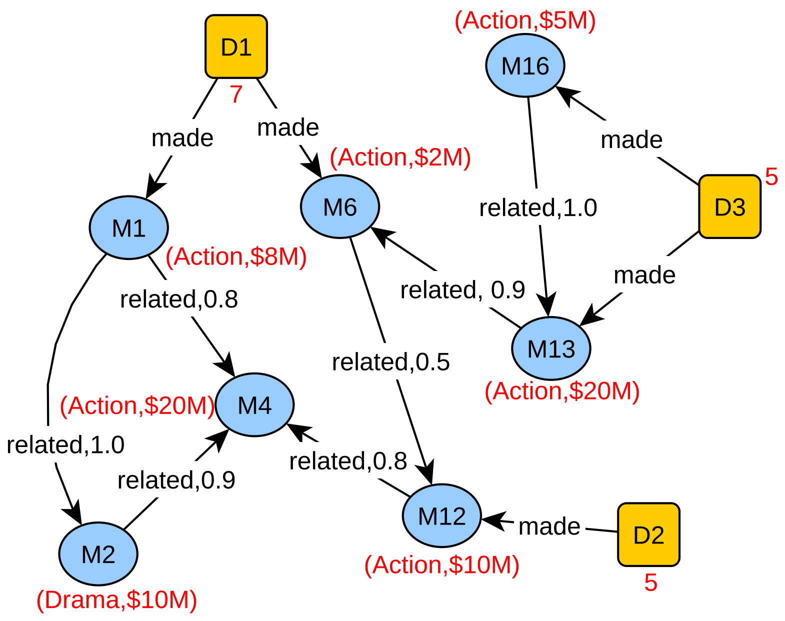
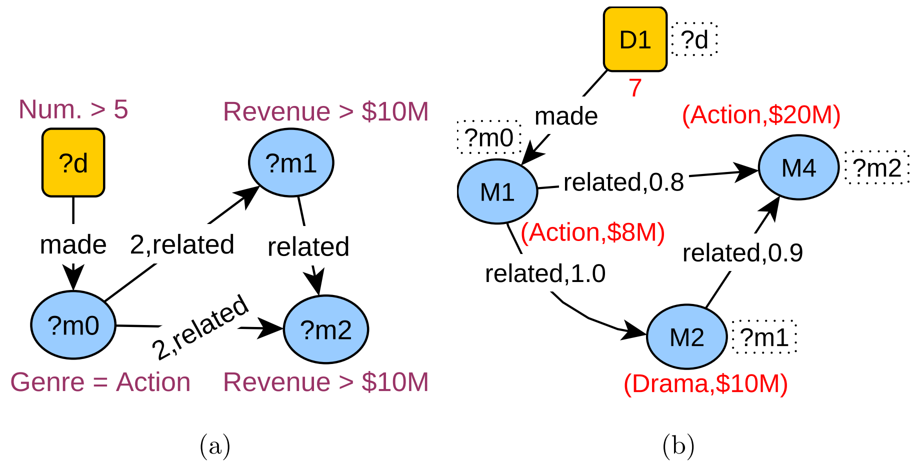


Figure 4.1: IMDb graph database. Blue vertices are movies, orange vertices are directors. Movies are annotated with (genre, revenue) pairs and directors with the number of movies directed. Between movies, there are "related" edges annotated with strength values. Note that all "made" edges have a strength of 1.0.

are dropped and moreover, some query edges are matched by paths. Figure 4.2(c) has only two edges when the query has 4, so some edges are dropped. Figure 4.2(c) allows the path M13-M6-M12 of length two to replace the edge $?m0 - ?m1$ in the query.

The quality of a match is scored via two functions, f and scr. When a path in the graph database is used to replace an edge as above, the function f is an edge strength aggregation function that combines the edge strengths together. Continuing the case with Figure 4.2(c), the function f would aggregate the edge strengths of the edges M13-M6 and M6-M12. The function scr returns a combined score by aggregating semantic properties of the matching subgraph and the quality of the match. The scr function in the example may look at the product of the revenue of



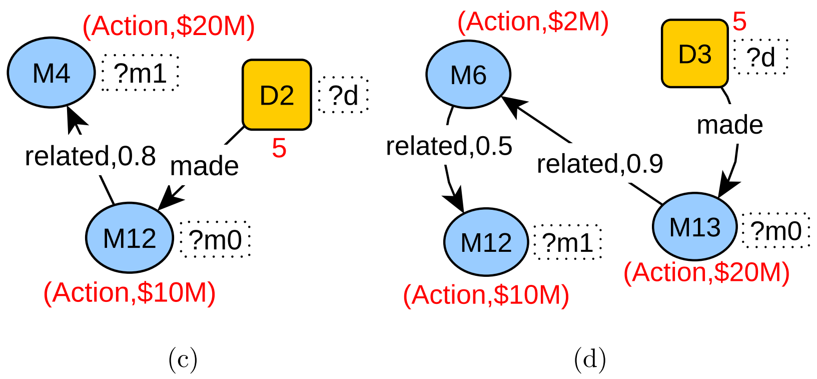


Figure 4.2: (a) An example top-k approximate (AIQ) query. Vertices are annotated with property constraints. (b) An exact match. Matched variables are annotated next to vertices. (c) Substitution where two edges are not mapped. (d) Substitution where M13 and M12 are connected through two edges.

the movie and the strengths of the path substitutions involved.

Section 4.1 formally defines graph DBs and our approximate top-k AIQ queries. Section 4.2 describes a set of four baseline approaches to process AIQ queries that build on top of 4 state of the art systems. All our baseline approaches involve a fair amount of pruning, as opposed to using a naive method.

Our AIQ-1 and AIQ-2 algorithms (Section 4.3) utilize two major index structures. The *Vertex Property Index* (VPI) captures maximal values of certain properties of vertices of various types that lie within the neighborhood of a given vertex. This enables one round of pruning. The second, *Vertex Step Index* (VSI), tells us whether at least one vertex in the neighborhood of a given vertex has a property value lying in a given range or not. The AIQ-1 and AIQ-2 algorithms include a subgraph matching method that utilizes these two specific indexes. These methods also include two novel contributions. An *extensibility* property allows us to eliminate redundant subgraph processing, thus increasing efficiency. *Pull Requests* allow us to incrementally retrieve the top-1, top-2, ..., top-k answers so that the top-k of an approximate query will be found after performing as few subgraph matching tasks as posisble. Section 4.4 reports on our experiments on three real-world data sets (CiteSeerX,IMDb,YouTube) comparing AIQ-1 and AIQ-2 with intelligent extensions of three existing algorithms in the literature: JenaTopN, SRank, and DOGMATopK. We show that both AIQ-1 and AIQ-2 are faster than all of these state of the art past algorithms. In particular, AIQ-2 is 7.3-82 times faster than these past algorithms. When comparing AIQ-1 and AIQ-2, we show that AIQ-2 is 2.3-3.3 times faster than AIQ-1.

4.1 Formal Definition of Approximated Importance Queries

We first formally define *Property Graph*, where both vertex and edge can have properties, whereas only vertex is allowed to have them in RDF graphs. The proposed approximate query is defined on property graph.

4.1.1 Definition of Property Graph

We assume the existence of four arbitrary, but fixed, disjoint sets $\mathcal{V}$, $\mathcal{L}$, $\mathcal{P}$, and VAR of *vertex names*, *edge labels*, *vertex properties*, and *variables*, respectively. Each $p \in \mathcal{P}$ has an associated domain $dom(p)$ which is a set (disjoint from V, $\mathcal{L}$, and $\mathcal{P}$) of values that can be assigned to p. We assume the existence of a special property $type \in \mathcal{P}$. A graph database is defined as follows.

Definition 4.1.1. *A* Graph Database $\mathcal{G}$ *is a tuple* $\langle V, E, \omega, \wp \rangle$*, where*

- $V \subseteq \mathcal{V}$ *is a set of vertices;*
- $E \subseteq V \times \mathcal{L} \times V$ *is a set of labeled edges;*
- $\omega : E \to [0, 1]$ *is a* strength assignment *function;*
- $\wp : V \times \mathcal{P} \to \bigcup_{p \in \mathcal{P}} dom(p)$ *is a (partial)* property assignment *function s.t. (i) if* $\wp(v, p)$ *is defined, then* $\wp(v, p) \in dom(p)$*, and (ii)* $\wp(v, type)$ *is defined for all* $v \in V$*.*

An example of such a graph database is shown in Figure 4.1. For instance, in this IMDb database, we have two types, “movie” and “director” and $\wp(v, type)$ assigns one of these two types to each vertex v. In the remainder of this paper, we assume that an arbitrary but fixed graph database $\mathcal{G}$ is given.

There are multiple ways to implement such graph DBs. The first candidate is the RDF model [51, 63], which has been extensively studied. The model requires an additional dummy vertex with a strength property to represent edges. A more suitable method is Neo4j’s *property graph model* [4], where properties can be assigned to both vertices and edges. We leverage both models.

4.1.2 Definitions of Approximate Importance Query

The goal of a subgraph matching query is to find all matched substitutions[2] w.r.t. a graph database $\mathcal{G}$ and a query graph (to be defined) that captures a notion of subgraph isomorphism.

Our proposed form of *top-k approximate query* is defined below. In the definition, we use the notion of *(numerical) property constraint* that is the application of a logical operator ($\wedge$, $\vee$, or $\neg$) to constraints or a comparison between *(numerical) terms* – a (numerical) term is a value of a property, a property of a query variable, a real number, or a polynomial-time computable function of numerical terms.

Definition 4.1.2 (Top-k Approximate Query). *A* top-k approximate query AQ *is a tuple* $\langle V_{AQ}, E_{AQ}, \chi, f, scr, k, m \rangle$*, where*

- $V_{AQ} \subseteq \mathsf{VAR}$;[3]
- $E_{AQ} \subseteq V_{AQ} \times \mathcal{L} \times \mathbb{Z} \times V_{AQ}$ *is a set of labeled edges annotated with* length limit $\ell \in \mathbb{Z}$ *– a query edge* $e \in E_{AQ}$ *can be mapped to a path containing up to* ℓ *edges by a substitution* θ*, i.e.* $e\theta = \{e_1, \cdots, e_{\ell' \leq \ell}\}$*,*[4] *where* $e_i \in E$*, and* e_{i-1} *and* e_i *are connected in* $\mathcal{G}$*;*
- χ *is a set of property constraints (*$\chi(?v)$ *will denote the constraints applied to variable* $?v$*);*
- $f : 2^E \rightarrow [0,1]$ *is a* strength aggregation *function that returns a number in* $[0,1]$ *for any given path* $\{e_1, \cdots, e_n\}$ *with* $e_i \in E$*, by aggregating* $\{\omega(e_1), \cdots, \omega(e_n)\}$ *– moreover, it always holds that* $f(\{\omega(e_1), \cdots, \omega(e_n)\}) \leq \max(\{\omega(e_1), \cdots, \omega(e_n)\})$*;*[5]
- $scr : \Theta \rightarrow \mathbb{R}$*, where* Θ *is the set of all possible substitutions, is a* scoring *function – moreover, given two substitutions* θ_1 *and* θ_2*, if* $\wp(?v\theta_1, p) \leq \wp(?v\theta_2, p)$ *and* $f(e\theta_1) \leq f(e\theta_2)$ *for all* $?v \in V_{AQ}$ *and* $e \in E_{AQ}$*, then* $scr(\theta_1) \leq scr(\theta_2)$*;*

[2]A substitution is traditionally defined to be a mapping function from query vertices to vertices of $\mathcal{G}$ such that all edge relationships from the query are preserved among the matched vertices of $\mathcal{G}$.

[3]In this paper, variables always start with "?".

[4]$?v\theta$ denotes the vertex in $\mathcal{G}$ mapped to $?v \in V_{AQ}$ by θ, and $e\theta$, where $e \in E_{AQ}$, denotes a path in $\mathcal{G}$ as described in the definition.

[5]This property of f is natural for multi-hop relationships: the strength of a path is bounded by the maximum strength in the path.

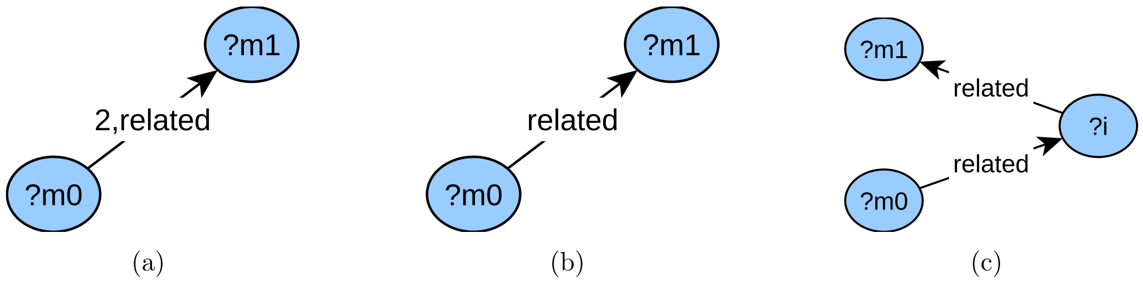


Figure 4.3: The set of substitutions for the query (a) is the union of the sets of substitutions for (b) and (c).

- *k is the number of substitutions to be computed;*
- *m is the number of edges that must be mapped by the substitutions for AQ – we assume $\wp(?v\theta, p) = 0$ (resp. $f(e\theta) = 0$) if $?v \in V_{AQ}$ (resp. $e \in E_{AQ}$) is not mapped by θ.*

There are different ways to deal with edge length limits. In [23], only the shortest path between two vertices is considered. The RDF model considers paths with different intermediate edges to be different substitutions [34, 9]. This is because the RDF model uses the bag (or multiset) semantics, which we adopt as well. For instance, in the case represented in Figure 4.3, substitutions for the queries in Figures 4.3(b) and (c) are also substitutions for the query in Figure 4.3(a).

In the query of Figure 4.2(a), which we will use as our running example, we assume $m = 2$, $f(e\theta) = \prod\limits_{e_i \in e\theta} \omega(e_i)$, and $scr(\theta) = \sum\limits_{?m} \wp(?m\theta, revenue) \times \sum\limits_{e \in E_{AQ}} f(e\theta)$. The substitutions for the query against the graph in Figure 4.1 are shown in Figures 4.2(b), 4.2(c), and 4.2(d). The score of the three substitutions in Figure 4.2 are 140.6M, 54M, 43.5M, respectively.

Of course, in a different query, the user might want to score answer substitutions differently. For instance, he might set $f(e\theta) = \underset{e_i \in e\theta}{max} \omega(e_i)$, and $scr(\theta) =$

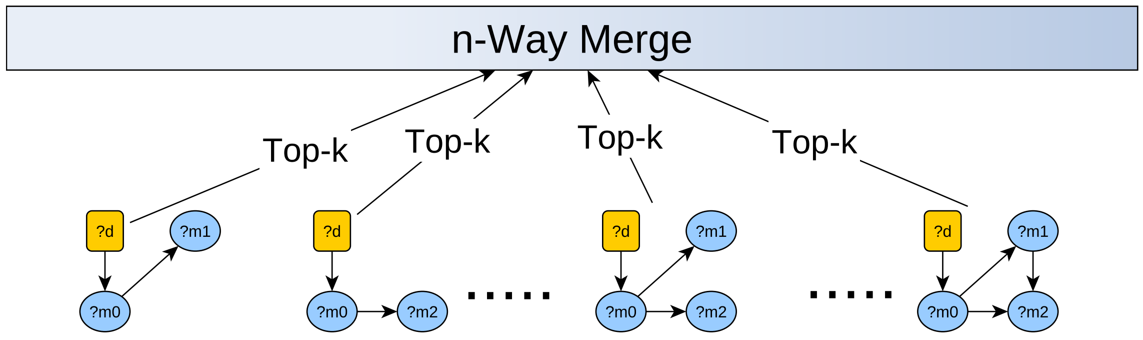


Figure 4.4: Query processing diagram for the Base approach.

$\sum_{?m} \frac{\wp(?m\theta, revenue)}{20M} + \sum_{e \in E_{AQ}} f(e\theta)$. In this case, the score of the three substitutions in Figure 4.2 would be 5.6, 3.3, and 3.4, respectively. This means that depending on which functions f and scr that the user specifies, the user can get very different top-k answers.

4.2 Baseline Approaches

Given a top-k approximate query $AQ = \langle V_{AQ}, E_{AQ}, \chi, f, scr, k, m \rangle$, an *exact sub-query* of AQ can be defined as a subgraph of (V_{AQ}, E_{AQ}) containing exactly m edges — all other components of the query remain unchanged. Clearly, a single query can generate $\binom{|E_{AQ}|}{m}$ exact sub-queries. In an *n-way merge* operation, we compute the set of top-k answers to each subquery, and merge them into a single set by loading items from each sorted list one-by-one and then performing sort-and-output operations among loaded items. It is easy to see that the top-k substitutions for AQ (*global top-k*) will always be contained in the union of all these top-k substitutions for the exact subqueries. The Base approach (Figure 4.4) performs this n-way merge.

An advantage of this approach is that we can leverage several state-of-the-art top-k subgraph matching algorithms. We show how we extended JenaTopN [2], SRank [40], Neo4j's Cypher query language [4], and DOGMATopK [52] to build on the Base.

We start by observing that most existing approaches to top-k computation are based on the threshold algorithm proposed in [17]. While finding substitutions, the top-k ones are maintained on the fly. If an upper bound of the score of a *partial* substitution (where some variables are not yet mapped, but an upper bound can be calculated by determining upper bounds for unmapped variables and edges) is smaller than the current k-th, then the partial substitution can be pruned.

Jena, the most popular open source RDF triplestore, supports edge length limits[6] without reporting intermediate path details, i.e. it only reports the start and end vertices of a matched path when $\ell > 1$. It is therefore not possible to retrieve intermediate edge strength values. Jena can process the proposed approximate query model only when all length limits are set to 1, and in that case, converting a subquery for Jena just amounts to adding dummy variables as a means to extract edge strength values, as mentioned in Section 4.1.1.

One optimization we applied to Jena is the addition of dynamic FILTER statements to the queries. When processing a subquery, we filter out the substitutions whose exact scores (or upper bounds, in the case of partial substitutions) are lower than the minimum score in the current global top-k. Upper bounds

[6]The edge length limit is not part of SPARQL 1.1 [34], but it is supported by the Jena implementation.

for partial substitutions are estimated using the maximum values of properties and strength in $\mathcal{G}$. For example, if we consider the first scoring function of the query in our running example, we can estimate an upper bound for $?m2$ as $\left(\sum_{?m \in V_{AQ} \setminus \{?m2\}} \wp(?m\theta, revenue) + 20M\right) \times \left(|IN(?m2)| + \sum_{e \in E_{AQ} \setminus IN(?m2)} f(e\theta)\right)$, where $IN(?m2)$ is the set of query edges incident on $?m2$, since the maximum revenue in $\mathcal{G}$ is 20M and the maximum strength is 1.0.

In addition to this, JenaTopN [2] uses a priority queue whose maximum size is limited to k in order to constantly maintain the top-k substitutions while avoiding the need to perform sorting.

SRank [40], which is an application of SQL's rank-aware join to SPARQL, is one of the most popular methods for top-k processing. SRank incorporates JenaTopN.

Neo4j [4] provides Cypher, a graph query language that is designed to be comparable to SQL and can process subqueries with length limits. MATCH statements for pattern matching make use of a recursive graph-traversal matching with backtracking. In Neo4j's underlying data structure, every node is connected to its neighbors by pointers. This index-free approach reduces needs for index lookups and requires an index to be applied only on the starting node during matching. Neo4j removes table joins from the picture and allows for scalable performance w.r.t. the size of the dataset – on the contrary, Jena suffers from performance degradation as data size increases, due to the number of table joins needed. We applied to Cypher the dynamic insertion of WHERE statements, that are the Cypher counterpart of SPARQL's FILTER.

Finally, we ported the DOGMATopK [52] algorithm to the Neo4j framework, as

DOGMATopK and Neo4j are based on a similar graph-traversal method, and more importantly, Neo4j can process queries with edge length limits.

4.3 Proposed Approaches

In this section we show how we extend the Base approach in two ways. We start by proposing a new query processing algorithm called AIQ-1 that makes use of two index structures, namely the *Vertex Property Index* and the *Vertex Step Index*. The vertex property index is used to compute tight upper bounds, while the vertex step index (and the materalized views created from it) provide a list of initial mapping candidates for each variable. These candidates will not only provide starting points for the matching process, but also drastically reduce the search space of substitutions. Later on, we further extend our approach and propose the AIQ-2 algorithm.

4.3.1 The Vertex Property Index

Our AIQ-1 algorithm makes use of a data structure called *Vertex Property Index* (VPI) that, for each vertex $v \in V$, stores the maximum value of each property p of each vertex of type t connected to v through a h-hop path whose last edge has direction d (with $d \in \{in, out\}$) w.r.t. v and label l. We denote this value as $\phi(v, h, (d, l), t, p)$. The index also stores the maximum value of edge strengths, denoted as $\phi'(v, h, (d, l), t, p)$.

The index is constructed as follows. For $h = 1$, we just look at the neighbors of each vertex v, i.e. we set $\phi(v, 1, (d, l), t, p) = \max_{u \in N(v,(d,l),t)} \wp(u, p)$ where $N(v, (d, l), t)$ is the set of vertices of type t that are connected to v through an edge (d, l). For $h > 1$, we build the index recursively by setting $\phi(v, h, (d, l), t, p) = \max_{u \in N(v)} \phi(v, h-1, (d, l), t, p)$, where $N(v)$ is the set of all neighbors of v. The same applies to ϕ': we set $\phi'(v, 1, (d, l), t, p) = \max_{u \in N(v,(d,l),t)} \omega(e)$ and, for $h > 1$, $\phi'(v, h, (d, l), t, p) = \max\limits_{u \in N(v)} \phi'(v, h-1, (d, l), t, p)$. Observe that the VPI for a property p can be constructed in time $O(h \times \delta \times |V|)$, where δ is the average degree of vertices in $\mathcal{G}$.

A portion of the VPI for our running example is reported in Table 4.1.

Table 4.1: Portion of the VPI for the running example. $\xleftarrow{l}$ stands for in-edge labeled l and $\xrightarrow{l}$ for out-edge labeled l.

$\phi(D1, 1, \xrightarrow{made}, movie, revenue) = 8M$
$\phi(M4, 1, \xleftarrow{related}, movie, revenue) = 10M$
$\phi(D3, 2, \xrightarrow{related}, movie, revenue) = 20M$
$\phi(M12, 2, \xleftarrow{related}, movie, revenue) = 20M$

Algorithm 4 uses the VPI to compute upper bounds on scores for partial substitutions.

Algorithm 4: Computation of upper bounds on scores. The pseudocode needed for considering strength values is omitted for simplicity.

```
  Input: Subquery $Q$, partial substitution $\theta$
  Output: Upper bound $\overline{scr(\theta)}$ on the score of $\theta$
1 $max_{?r} \leftarrow \infty$, for each unmapped variable $?r$
2 foreach $v$ mapped to $?v \in V_Q$ do
3     foreach unmapped variable $?r \in V_Q$ do
4         $tmax_{?r} \leftarrow 0$
5         foreach possible value of $h$ from $?v$ to $?r$ along the shortest path do
6             $tmax_{?r} = \max\left(tmax_{?r}, \phi(v, h, e, t, p)\right)$
7         $max_{?r} = \min\left(max_{?r}, tmax_{?r}\right)$
8 return $\overline{scr(\theta)}$ computed by aggregating $max$ and actual values
```

The algorithm considers all unmapped variables (or edges – the computation is similar for this case). For each such variable, it computes multiple maximum values w.r.t. different mapped vertices and then takes the minimum (Line 7). The final upper bound is computed by aggregating these estimated values (for unmapped variables) and actual values (for mapped variables). Note that if multiple values of h are possible (this happens when length limits are greater than 1), the algorithm computes the maximum values for all possible values of h.

In our running example, suppose D1 is matched to $?d$. Two different values of h are possible (2 and 3) for $?m1$ along the path $?d \rightarrow ?m0 \rightarrow ?m1$. Thus, $max_{?m1} = \max_{h \in \{2,3\}} \big(\phi(D1, h, \xrightarrow{related}, movie, revenue)\big)$.

The following theorem establishes the correctness of this approach.

Theorem 4.3.1. *If θ is a partial substitution and θ' is a complete substitution that can be generated from θ, then $\overline{scr(\theta)} \geq scr(\theta')$ — recall that $\overline{scr(\theta)}$ is the value returned by Algorithm 4.*

Proof. Assume that $\theta = \{?v_1/v_1, ?v_2/v_2, \ldots\}$ and $?r$ is an unmapped variable. Let $spath(?v_i, ?r)$ be the shortest path between $?v_i$ and $?r$, ignoring any length limit, and $hvals(?v_i, ?r)$ be the set of all possible h values along $spath(?v_i, ?r)$. For a vertex r that will be mapped to $?r$, there must be at least one valid concatenation of h edges in $\mathcal{G}$ between v_i and r, where $h \in hvals(?v_i, ?r)$, for all i. Since $\phi(v_i, h, (d, l), t, p)$ captures all the candidate vertices for $?r$, and it also considers the maximum values that can derive from v_i, it is a valid upper bound. The same argument can be applied to unmapped edges, as the strength value of a path is bounded by the maximum of the strength values of the edges in the path. Thus, as a consequence of the monotonicity of scr, we have $\overline{scr(\theta)} \geq scr(\theta')$. □

4.3.2 The Vertex Step Index

The *Vertex Step Index* (VSI) helps identify suitable initial candidates for each query variable. It does this by storing information about connected vertices that meet some property constraints. In particular, it stores a value $\rho(v, h, (d, l), t, p \geq \delta)$ for each vertex v, number of hops h (upto some maximum), direction $d \in \{in, out\}$, label l, and property constraint $p \geq \delta$ where $\delta \in \mathit{dom}(p)$. $\rho(v, h, (d, l), t, p \geq \delta)$ is set to 1 if at least one vertex connected to v in a h hop path with (d, l) as the last edge satisfies the constraint $p \geq \delta$ — otherwise it is set to 0. By pre-computing and storing such information in the VSI, our AIQ-1 and AIQ-2 algorithms are able to obtain significant speedups.

The index is constructed as follows. For $h = 0$, we merely need to see if the vertex v itself satisfies the constraint $p \geq \delta$, i.e. we set

$$\rho(v, 0, \cdot, \cdot, p \geq \delta) = \begin{cases} 1, \text{if } \wp(v, p) \geq \delta; \\ 0, \text{otherwise,} \end{cases}$$

for every property of v and every 10th percentile of $\mathit{dom}(p)$. For $h = 1$, we look at the immediate l-neighbors of v and set

$$\rho(v, h, (d, l), t, p \geq \delta) = \bigvee_{u \in N(v,(d,l),t)} \rho(v, 0, \cdot, \cdot, p \geq \delta)$$

and for $h > 1$ we set

$$\rho(v, h, (d, l), t, p \geq \delta) = \bigvee_{u \in N(v)} \rho(u, h-1, (d, l), t, p \geq \delta).$$

A portion of the VSI for our running example is reported in Table 4.2.

Table 4.2: Portion of the VSI for the running example.

$\rho(M1, 1, \xleftarrow{made}, director, Num. \geq 5) = 1$
$\rho(M6, 2, \xleftarrow{made}, director, Num. \geq 7) = 0$
$\rho(M13, 1, \xrightarrow{related}, movie, Revenue \geq 10M) = 0$
$\rho(D1, 0, \cdot, \cdot, Num. \geq 5) = 1$

During indexing time, we create materalized views defined as

$View(h, (d, l), t, p \geq \delta) = \{v | v \in V \wedge \rho(v, h, (d, l), t, p \geq \delta) = 1\}$. In the implementation, such views are saved on disk and loaded as needed (we save each view in a separate file after sorting vertices in lexicographical order). We could build all views for the datasets we used in the experiments in a couple of hours.[7]

4.3.3 The AIQ-1 Algorithm

The AIQ-1 algorithm is reported in Algorithm 5 – for now, let us ignore the iterations on Lines 18 and 24 and the instruction on Line 23, which will be used in the AIQ-2 variation of the algorithm.

The algorithm starts by assigning a candidate set to each variable. On Line 1, it loads $View(h = 0, \cdot, \cdot, clo(\chi(?v)))$ for each variable $?v$, where $clo(\chi(?v))$ is the most "tightly" indexed property that subsumes $\chi(?v)$. For example, assume $p \geq x$ is not in the VSI, which instead indexes $p \geq x'$, with $x < x'$. If $\chi(?v) = p \geq x$, then $clo(\chi(?v)) = p \geq x'$. If there are multiple constraints for a variable, the algorithm uses the one that leads to the smallest view size. The initial candidate sets are then refined, for each variable $?r$, using edge directions and labels in the shortest path tree of Q. Finally, the algorithm performs the appropriate pairwise intersection operations[8] on Line 8.

For instance, in our running example, assume $?q = ?d$ and $?r = ?m2$. $IC_{?m2}$ is

[7]Due to space constraints, we do not develop algorithms in this paper to maintain these materialized views when the graph database is updated. When insertions and deletions are allowed to the graph database, both VPI and VSI can be updated in $O(h \times \delta)$ time. Simply put, the update method propagates changed values to h-hop neighbors and the indexed values associated with these neighbors will change if needed.

[8]Both intersections and unions (Line 7) of views can be performed using the merge-join method as the vertices are already sorted. On a RAID system with two SAS drives, performing these operations on views containing about 1M vertices takes less than a couple of seconds.

Algorithm 5: The AIQ-1 algorithm. Lines marked with E (resp., P) are added in the AIQ-2 variation to employ extensibility (resp., pull-requests).

```
   Input: Subquery $Q$
   /* Initialize views                                                    */
1  $IC_{?v} = View(0, \cdot, \cdot, clo(\chi(?v)))$ for all $?v \in V_Q$
   /* Refine views                                                        */
2  $T \leftarrow ShortestPathTree(Q)$
3  foreach $?r \in V_Q$ do
4      foreach $?q \in V_Q$ do
5          $TIC_{?r} \leftarrow \emptyset$
6          foreach possible value of $h$ from $?r$ to $?q$ in $T$ do
7              $TIC_{?r} = TIC_{?r} \cup View(h, e, t, clo(\chi(?q)))$
8          $IC_{?r} = IC_{?r} \cap TIC_{?r}$
   /* Start query processing                                              */
9  $?v \leftarrow arg\,min_{?v}|IC_{?v}|$
10 foreach $m \in IC_{?v}$ in descending order of the values of the property used by
   clo do
11     $\theta \leftarrow \{?v = m\}$
12     map$(Q, \theta)$
---------------------------------------------------------------------------
13 FUNCTION map
   Input: Subquery $Q$, partial substitution $\theta$
   Data: Local top-$k$ set $S$, priority queue $D$
14 foreach mapped edge $e = (?x, 1, ?y) \in E_Q$ do
15     if $e\theta$ is not in $\mathcal{G}$ then return
16 if all variables are mapped then
17     if $scr(\theta) \geq Min_S$ then add $\theta$ to $S$
18     E foreach $Q'$ allowed to be extended from $Q$ do
19         E map$(Q', \theta)$
20     return
21 else
       /* Try to prune                                                    */
22     if $\overline{scr}(\theta) < Min_S$ then
23         P Add $\theta$ to $D$
24         E foreach $Q'$ allowed to be extended from $Q$ do
25             E map$(Q', \theta)$
26         return
   /* Map one additional variable                                         */
27 $(u, ?v) \leftarrow arg\,min_{(u,?v)}|IC_{?v} \cap N(u, e, t)|$, where $e$ is the edge between $u$ and
   $?v$ in $Q\theta$ and $t = ?v.type$
28 foreach $c \in IC_{?v} \cap N(u, e, t)$ in descending order of the values of the property
   used by clo do
29     $\theta' = \theta \cup \{?v = c\}$
30     foreach $Q' \in$ spawn$(Q, (u, ?v), \theta')$ do
31         map$(Q', \theta')$
```

first initialized to $View(h = 0, \cdot, \cdot, revenue \geq \$10M)$. Possible values of h are 2 and 3 along the path $?d \rightarrow ?m0 \rightarrow ?m2$. Thus, the algorithm computes

$$TIC_{?m2} = \bigcup_{h \in \{2,3\}} View(h, \xleftarrow{made}, director, Num. \geq 5)$$

Lastly, $IC_{?m2} = TIC_{?m2} \bigcap View(h = 0, \cdot, \cdot, revenue \geq \$10M)$ because all movies matched to $?m2$ should have revenue greater than \$10M and also be 2 or 3 hops away from a director who has directed at least 5 movies.

After this initialization phase, for all variables $?v$, $IC_{?v}$ will contain all the vertices of $\mathcal{G}$ that can be matched to $?v$. This is ensured by the following result.

Proposition 4.3.1. *Given a subquery Q and a variable $?v$ in Q, none of the vertices in $V \setminus IC_{?v}$ can be matched to $?v$.*

AIQ-1 then calls the `map` subroutine (Line 12) to find the top-k substitutions for each starting point m, in descending order of the values of the property used by clo. `map` performs a depth-first search. On Line 15, it prunes a partial substitution θ if it violates subgraph isomorphism. It then checks whether all of the variables are mapped by θ (Line 16). If they are, it computes the corresponding score and adds the substitution to the local top-k set S. Unmapped variables can be pruned after comparing $\overline{scr(\theta)}$ with the minimum score Min_S of the substitutions in S (Line 22). On Line 27, `map` chooses the variable to map next. The chosen variable $?v$ must have at least one mapped neighbor and must minimize the number of mapping candidates $IC_{?v} \cap N(u, e, t)$. AIQ-1 then continues the search for each of the "spawned" queries (Line 31).

The key idea behind the `spawn()` procedure is that, when processing an edge

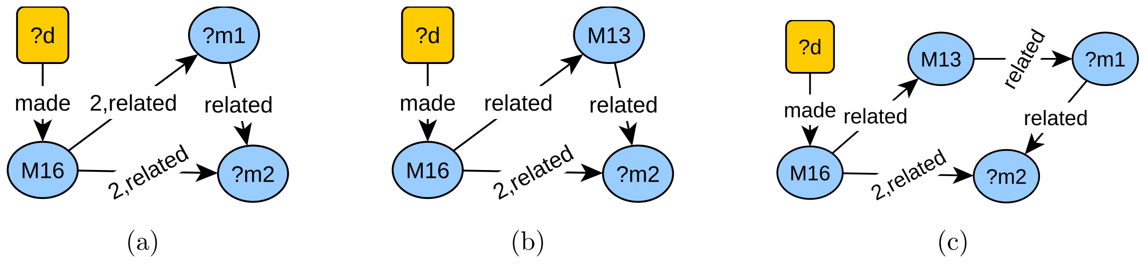


Figure 4.5: Queries (b) and (c) are spawned when M13 is matched to $?m1$ in (a).

$e = (u, \ell, ?v)$ with $?v$ mapped to v, the procedure creates two queries: in the first one ℓ is set to 1, and in the second one e is divided into $e' = (u, 1, v)$ and $e'' = (v, \ell - 1, ?v)$. Observe that e'' will be spawned again in the next step of the depth-first search – this way, we ensure that all length values less than or equal to ℓ are considered for e. For instance, consider the case in Figure 4.5 and assume $u = M16$, $?v = ?m1$, and $v = M13$. In this case, the mapped edge $(M16, 2, M13)$ generates two spawned queries: one with $(M16, 1, M13)$ and another with $(M16, 1, M13)$ and $(M13, 1, ?m1)$.

4.3.4 The AIQ-2 Algorithm

The AIQ-2 algorithm is an extension of AIQ-1 that makes use of *extensibility* and *pull-requests*.

Extensibility

Let sq_i and sq_j be two subqueries that originated from a top-k approximate query AQ. We say that sq_i *subsumes* sq_j (denoted $sq_j \sqsubset sq_i$) if and only if sq_i has all of the edges in sq_j plus an additional one. In this case, substitutions for sq_j can be reused when finding substitutions of sq_i. This extensibility property helps prune redundant subgraph matching tasks. Using the subsumption relationship, we can draw an extensibility tree such as the one shown in Figure 4.6 – in the diagram, there is an edge from sq_j to sq_i when the substitutions for sq_j can be extended to extensions for sq_i.

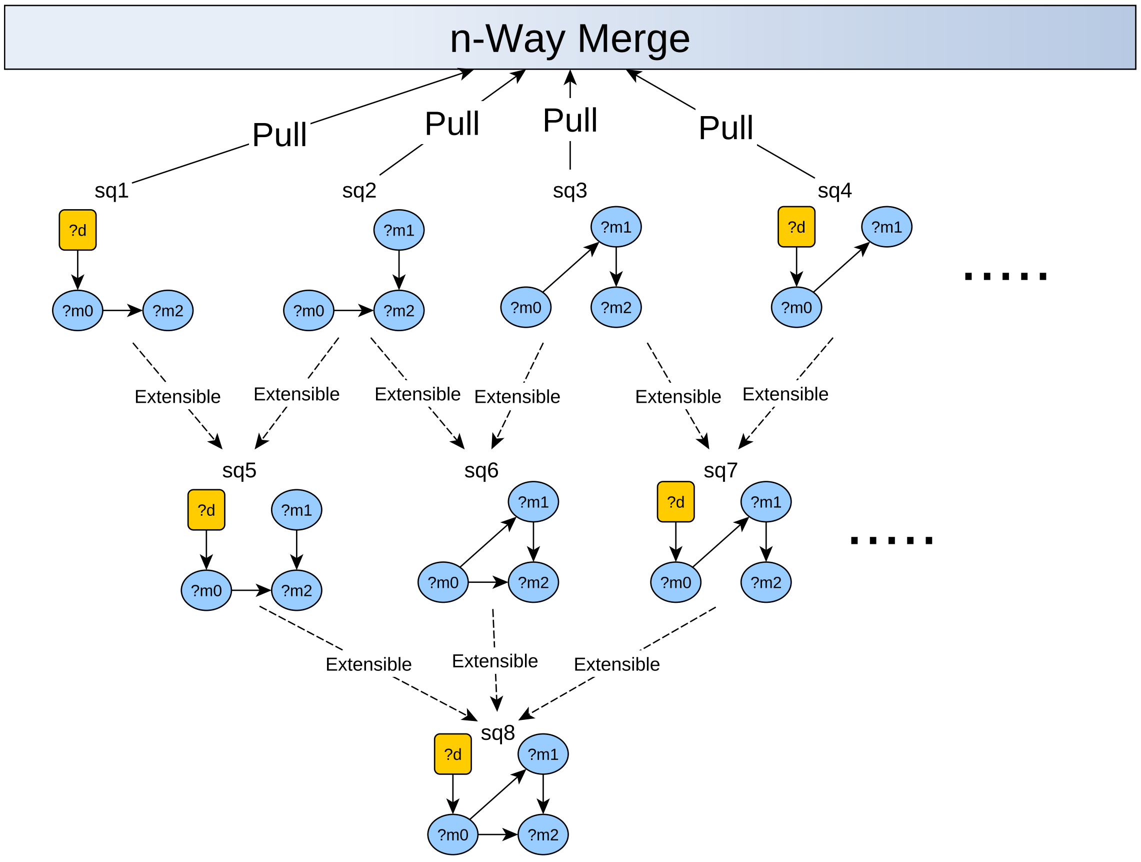


Figure 4.6: Query processing diagram for AIQ-2.

In AIQ-2, we selectively allow extensions in order to prevent duplicate substitutions. For instance, in Figure 4.6, if substitutions for sq_1 and sq_2 are allowed to be extended into substitutions for sq_5, duplicate substitutions will have to be handled. We choose the single subquery to be extended using the following dynamic programming formulation:

$$F(sq_i) = \begin{cases} \min\limits_{sq_j}\Big(F(sq_j) + w(v)\Big) & \text{if } sq_i \text{ has children} \\ w(sq_i) & \text{otherwise} \end{cases}$$

where $w(sq_i) = \sum\limits_{?v \in V_{sq_i}} \ln |IC_{?v}|$ and $sq_j \sqsubset sq_i$.

There are multiple ways to extend each sq_i. For example, Figure 4.6 shows multiple different paths through which sq_1, sq_2, sq_3, and sq_4 can be extended to get sq_8. Of course, the exact query sq_8 only needs to be generated with one of these paths so as to avoid duplication. Our dynamic programming formulation finds the shortest path from one of the smallest subqueries such as sq_1, sq_2, sq_3, and sq_4 to sq_i on the vertex-weighted extensibility tree. The vertex weight $w(sq_i)$ is based on the size of candidate set of variables in sq_i. Thus, it represents an upper bound on the number of possible answers to the exact subquery sq_i. The shortest path that minimizes the sum of vertex weights provides an optimal extension strategy to generte sq_i.

The iterations on Lines 18 and 24 are added to Algorithm 5 in order to make use of extensibility – this way, the algorithm can reuse pruned partial substitutions when some of them may be extended to substitutions for larger subqueries.

Pull-Requests

The AIQ-2 algorithm processes top-k approximate queries through a sequence of *pull-request* operations. In particular, it first invokes AIQ-1 to retrieve the top-1

Algorithm 6: Pull-request operation.

```
1  FUNCTION pull-request
   Input: Subquery $Q$
   Data: Priority queue $D$, local top-$k$ set $S$
   Output: Pull response
2  $\theta \leftarrow getTop(S)$
3  foreach $\theta' \in D$ do
4      if $\overline{scr(\theta')} \geq scr(\theta)$ then
5          remove $\theta'$ from $D$
6          map $(Q, \theta')$
7          $\theta \leftarrow getTop(S)$
8      else
9          break
10 return $get\&removeTop(S)$
```

substitution and then it incrementally retrieves the top-k' substitutions with $k' \in [2..k]$.

In order to support pull-requests, the pruned partial substitutions must be saved in the priority queue D for later use, so Line 23 is added to Algorithm 5. The substitutions in D are kept sorted in descending order of upper bound.

Algorithm 6, which executes pull-requests, retrieves (but does not remove) the top substitution θ from S (Line 2), and compares θ with all substitutions in D. If the upper bound of a substitution θ' in D is greater than $scr(\theta)$, it invokes `map` to process θ' (Line 6) and update the top substitution in S (Line 7), as it may have changed while processing θ'. The `map` procedure, when invoked on Line 6, is set to not produce extensions, as Algorithm 5 already computed them (on Lines 18 and 24). At this point, if $\overline{scr(\theta')} < scr(\theta)$, the procedure can be stopped (Line 9).

The following example illustrates how AIQ-2 works.

Example 4.3.2. *Consider the extension path shown in Figure 4.7 that was calculated by our dynamic programming algorithm. AIQ-2 sends pull-requests to the smallest subqueries and obtains pull-responses based on the selected extension paths.*

For example, sq_1 considers all its extended subqueries (sq_5 and sq_8) when answering a pull-request. Figure 4.8 (a) shows a substitution for sq_1 that can be extended to sq_5 as in Figure 4.8 (b). Figure 4.8 (b) can be further extended to Figure 4.8 (c) for sq_8. sq_1 then returns the best of these substitutions as a pull-response.

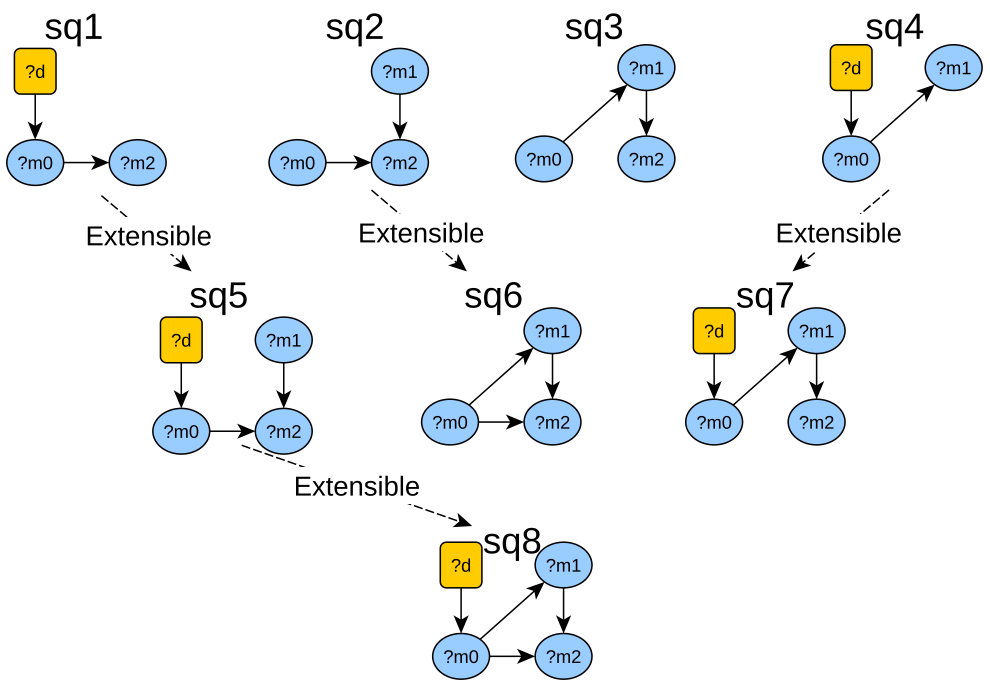
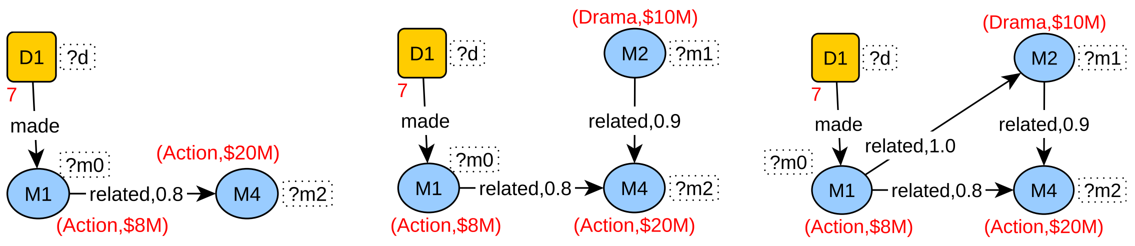


Figure 4.7: An example of selected extension paths



(a) sq_1 (b) sq_5 (c) sq_8

Figure 4.8: Extension steps

Using n-way merging in this process allows us to perform much less than $|SQ| \times k$ pull-requests when computing the top-k answer to an approximate query

AQ (where SQ is the set of subqueries corresponding to AQ). This is ensured by the following result.

Theorem 4.3.3. *Let AQ be a top-k approximate query and SQ be the set of subqueries generated from AQ. The top-k substitutions for AQ can be computed after $|SQ| - 1 + k$ pull-requests.*

Proof. The global top-1 result is the local top-1 result of one of the subqueries. Therefore, finding the global top-1 requires $|SQ|$ pull-requests. The global top-2 result is either the top-1 result of one of the other subqueries, or the top-2 result of the subquery that returns the global top-1 result. Retrieving each of the global top-k' results, with $k' \in [3..k]$, requires one additional pull-request. Therefore, computing the top-k results needs $|SQ| - 1 + k$ pull-requests. □

4.4 Experimental Evaluation

We conducted our experimental assessment using three real graph databases: CiteSeerX [27] (which contains 16.2M edges), IMDb [37] (31.4M edges), and YouTube [25] (56.4M edges). We added edge strength values to the datasets using two methods: *fractional*, where the strength of an edge from v to u is calculated as $\frac{|N(v) \cap N(u)|}{|N(v)|}$ (for example, in CiteSeerX, the more common co-authors, the higher the strength of the link between two authors) and *randomized*, which generates a random strength value between 0.5 and 1.0.

We used five different types of queries, summarized in Table 4.3. Each query type consists of s vertices and t edges, and r out of the t edges have $\ell = 2$ (i.e. r edges allow for 2-hop substitutions). For each query type, we generated numerous queries with all possible valid edge label combinations.[9] We added a property constraint $p \geq x$ to each variable, where x is a random number between the 50th and 90th percentile of $dom(p)$.

[9] In IMDb, for example, $?p0 \xrightarrow{cast} ?m1 \xrightarrow{direct} ?m2$ is not a valid query because it says $?p0$ was in $?m1$ and $?m1$ directed another movie $?m2$ which, of course, does not make sense.

Table 4.3: Query types. s is the number of variables. t is the number of edges. r is the number of edges whose $\ell > 1$. q is the number of generated queries.

	Q1	Q2	Q3	Q3s	Q4	Q4s	Q5
s	3	3	4	4	4	4	5
t	2	3	4	4	5	5	6
r	0	0	1	0	1	0	2

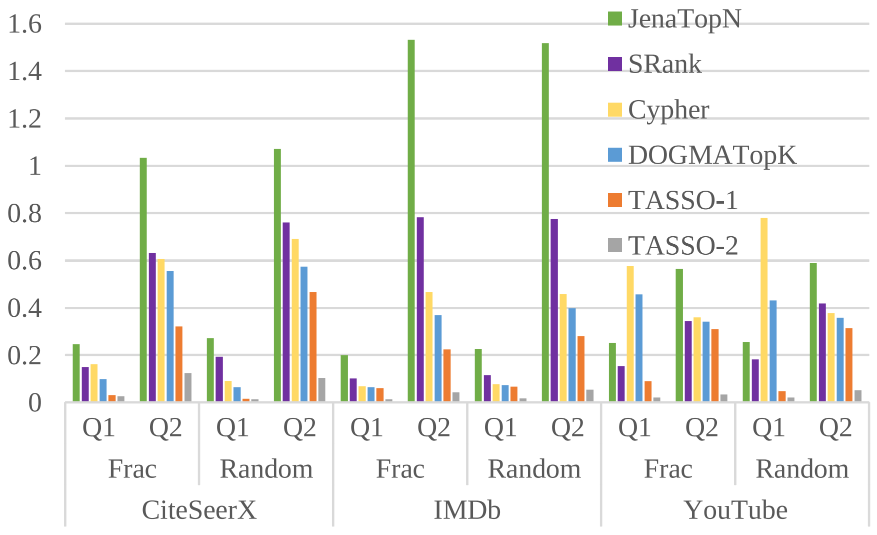


Figure 4.9: Processing time (in hours) of Q1 and Q2 when $\ell = 1$ with $k = 100$.

All experiments were carried out on a cluster of 64 nodes, each with an Intel Xeon CPU clocked at 2.40GHz with 24GB RAM and two SAS hard drives. We assigned a dedicated machine to process each type of queries and dropped all I/O caches between queries for fair comparison.

1-Hop Queries. Figure 4.9 and Figure 4.10 show the query processing times for query types Q1, Q2, Q3s, and Q4s, all of which have $\ell = 1$ for all query edges. These are relatively simple queries.

JenaTopN showed the poorest performance, except for Q1 on YouTube. Triple-stores generally suffer from massive table-joins [9, 21, 55], and our results show that this is the case even for top-k queries with dynamic FILTER statements. In most cases, Cypher outperformed JenaTopN (for Q2 on IMDb it was about 3 times faster) and showed similar performance to SRank. DOGMATopK could answer queries in 50% $\sim$ 90% of the time needed by Cypher. AIQ-1 was faster than DOGMATopK in many cases (about 15 times for Q1 on YouTube). AIQ-2 showed very stable performance and was around one order of magnitude faster than all other algorithms. The average relative processing times are summarized in Table 4.4.

For Q3s and Q4s, Cypher's performance was the poorest, and its performance quickly degraded as query size increased. AIQ-2's performance was very stable. In all cases, AIQ-2 could answer queries in an hour. In comparison with other algorithms, AIQ-2 was 7.2 $\sim$ 82 times faster than past work, as shown in Table 4.4. Moreoever, AIQ-2 was 2.3$\sim$ 3.3 times faster than AIQ-1.

Table 4.4: Average relative processing time (AIQ-2 = 1)

Query Type	JenaTopN	SRank	Cypher	DOGMATopK	AIQ-1
Q1/Q2	18.65	9.51	11.91	8.46	3.33
Q3s/Q4s	16.83	7.27	81.51	6.57	2.39
Q3/Q4/Q5	-	-	-	5.09	2.27

Multi-hop queries. The query processing times for query types Q3, Q4, and Q5, which allow at least one edge to support multiple hops, are shown in Figure 4.10. Note that in this figure, there is a gap between 18 hours and 20 hours on the y-axis to show the very poor performance of some algorithms. We ran these experiments only against DOGMATopK as it provided the best performances in the previous experiments, and the difference w.r.t. JenaTopN, SRank and Cypher t]

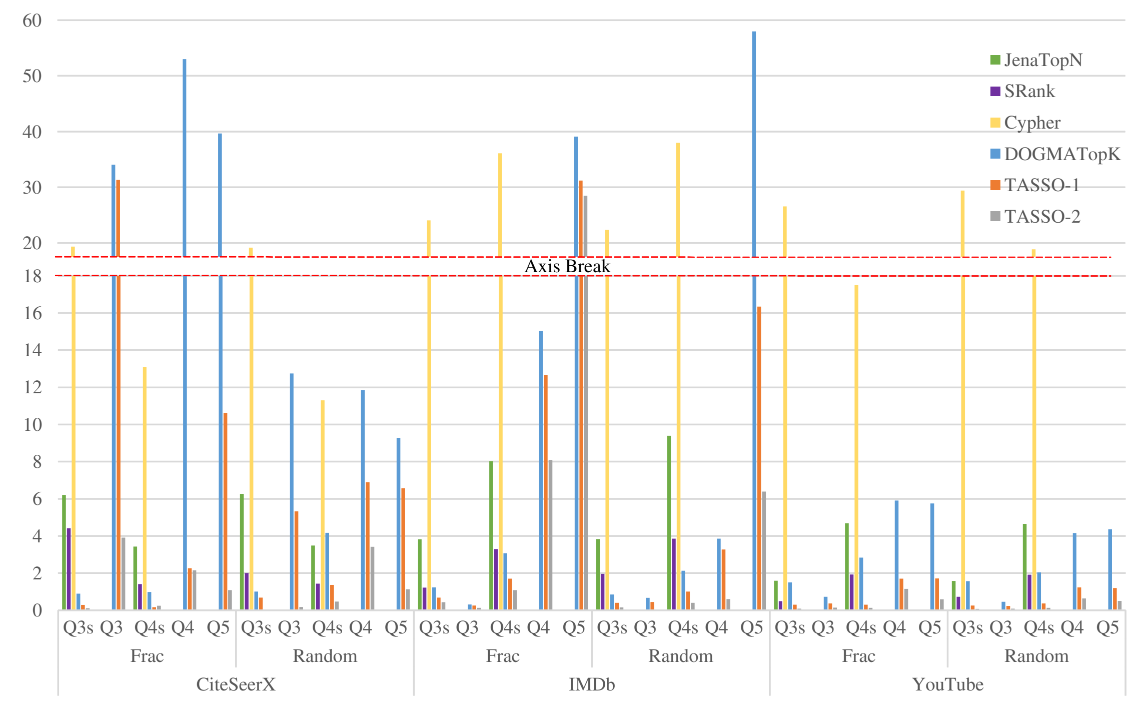


Figure 4.10: Processing time (in hours) for Q3, Q4, and Q5 with $k = 100$.

is expected to increase when dealing with edge limits greater than 1. Moreover, the syntax of SPARQL does not allow using intermediate vertices[10] for scoring terms when $\ell > 1$. Thus, there is no way to test those queries with triplestores.

We see from this figure that AIQ-2 is always significantly better than DOGMATopK, usually by one order of magnitude. The biggest savings occured with Q4 and Q5 on CiteSeerX with the fractional method, where AIQ-2 was $25 \sim 40$ times faster than DOGMATopK. For Q3 and Q5 on CiteSeerX, AIQ-2 was $8 \sim 10$ times faster than AIQ-1. *In general, our fastest algorithm is AIQ-2 which is significantly faster than all other algorithms that we tested, with the speedup improving with the complexity of the query.*

[10]Remember that edge strength values are saved in dummy vertices.

t]

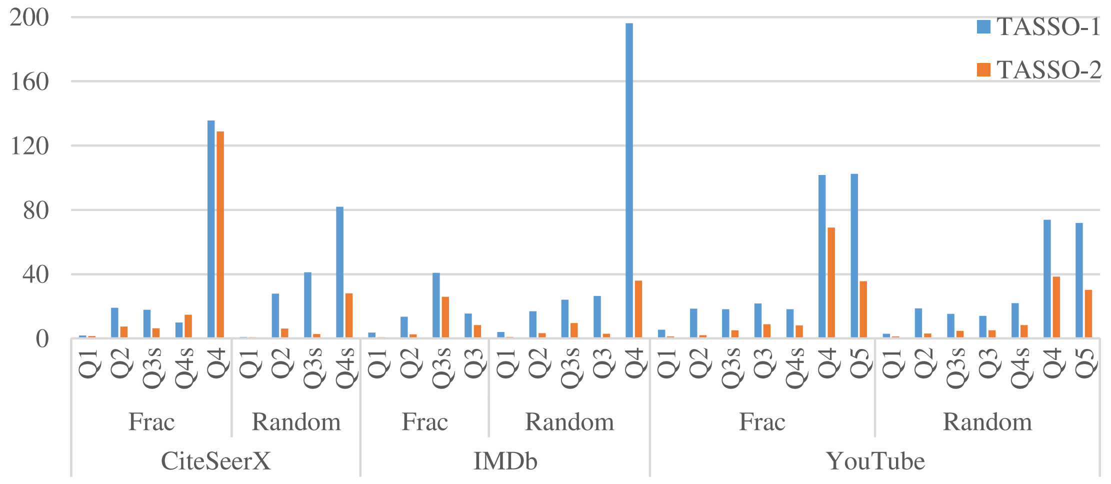


Figure 4.11: Processing time (in minutes) comparing AIQ-1 with AIQ-2 with $k = 100$. Queries whose response time is greater than 200 mins are omitted in this chart. Please refer to Figure 4.10 for them.

More detailed comparison of AIQ-1 and AIQ-2. Because previous figures show the y-axis in hours (because of the slow performance of some of the baseline algorithms), we show Figure 4.11 below which provides a y-axis in minutes.

Pruning Efficiency. For every pruned partial substitution, we also considered the ratio of unmapped variables to all variables in the query at the time pruning occurred, as a measure of how early the substitution was pruned. The greater this ratio, the more effective the pruning because it means that the pruning did not try to map many variables. Figure 4.12 shows the results in the cases where half variables at most were mapped. AIQ-1 and AIQ-2 showed very high ratios. This confirms that our VPI structure actually provides tighter upper bounds than DOGMATopK. In addition, AIQ-2 proved it was capable of pruning (i) more partial substitutions (ii) earlier than the other algorithms.

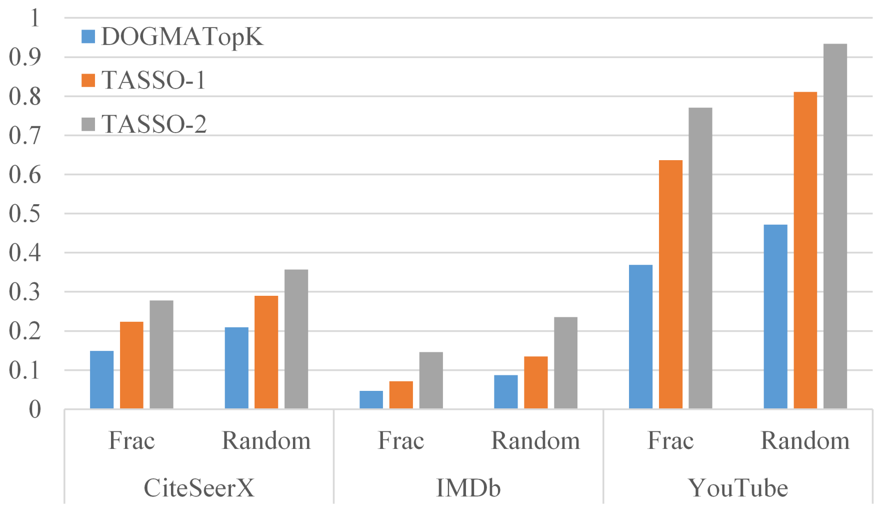


Figure 4.12: Ratio of partial substitutions where at most half variables are mapped at the time of pruning.

Run-time with Varying k. We ran experiments to measure runtimes when the value of k is varied (Figure 4.13). DOGMATopK and AIQ-1 required progressively longer than AIQ-2 as k increased. There was no significant difference between the cases under AIQ-2, confirming the claim in Theorem 4.3.3.

Number of Complete Substitutions Generated. Some complete substitutions lead to an answer in the top-k, others do not. So one measure of the effectiveness of a pruning strategy for top-k queries is the number of complete substitutions generated by an algorithm,. Figure 4.14 shows this number for the three algorithms (DOGMATopK,AIQ-1,AIQ-2) that performed well in the preceding experiments. We see that AIQ-2 performs the best, always generating the smallest number of complete substitutions, suggesting that its pruning strategy is working well. For instance, on CiteSeerX, this number is $40 \sim 50$ times smaller than the others.

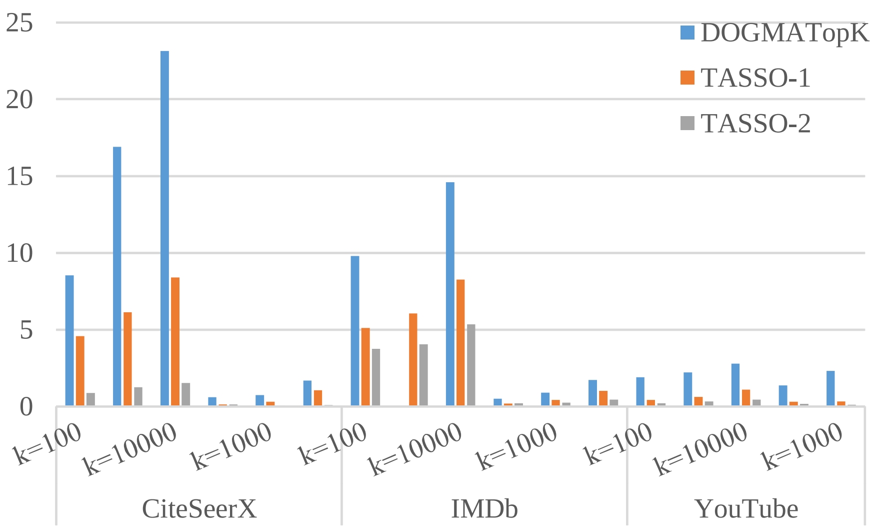


Figure 4.13: Processing time (in hours, averaged for all query types) for different values of k.

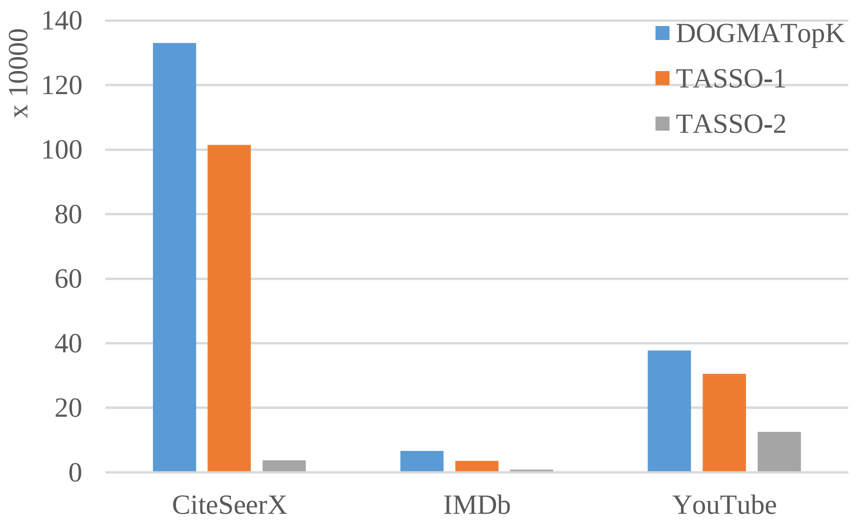


Figure 4.14: Number of complete substitutions generated (averaged for all cases).

Chapter 5: Probabilistic Importance Queries

In the previous chapters, we introduced importance queries based on exact subgraph matching. However, there are many applications where a user wishes to express uncertainty or probability of answers. For this, he or she may specify subgraph queries structured in "blocks", some of which are "fixed" in the sense that they must be completely matched by every query answer. Some of query blocks are such certain that they must be matched in answers, but other blocks are uncertain or conditional. This structure is suitable to express uncertain queries. In each query block, a base score is defined from mapped vertices' properties on it and, we aggregate them to calculate the final probability. Thus, more query blocks are matched, it is likely for an answer to have a higher probability.

5.1 Formal Definition of Probabilistic Importance Queries

We are now ready to introduce the notion of query block. Let $*$ be a distinguished "wildcard" label not occurring in $\mathcal{L}$.

Definition 5.1.1 (Query Block)**.** *A* query block *is a triple $QB = \langle V_{QB}, E_{QB}, \chi_{QB}\rangle$, where $V_{QB} \subseteq V \cup \mathsf{VAR}$, $E_{QB} \subseteq V_{QB} \times (2^{\mathcal{L}} \cup \{*\}) \times V_{QB}$, and χ_{QB} is a finite set of constraints over variables in V_{QB}.*

Thus, a query block is a directed edge-labeled graph where (i) each vertex is either a vertex name or a variable symbol, and (ii) each edge is labeled with a set of edge labels (or $*$). Intuitively, the user uses this graph to specify the structure of the sub-graphs of $\mathcal{G}$ he is looking for ($*$ matches any label). Finally, the constraints in χ_{QB} express requirements about variables or their properties. A vertex in $\mathcal{G}$ can only "match" a variable if it satisfies the constraints associated with that variable. We use VAR_{QB} to denote the set of all variables in a query block QB, i.e. $\mathsf{VAR}_{QB} = \mathsf{VAR} \cap V_{QB}$.

Example 5.1.2. *Figure 5.1 shows a graph database we will use as our running example. This figure shows a set of users and companies with edge labeled links. Each vertex has two properties,* postsAboutABC *and* endorsedDEF *telling us the number of times the vertex posted about* ABC *(we can think of this as "mentions" in Twitter or Facebook) and the number of endorsements of* DEF *(you can think of this as say the number of times the person said something positive about* DEF *according to a sentiment analysis program [59]). For instance, we see that the vertex "Bob" posted 50 posts mentioned* ABC *and 2 posts in which he expressed positive sentiment about* DEF. *Of course, these are just example properties. We can imagine*

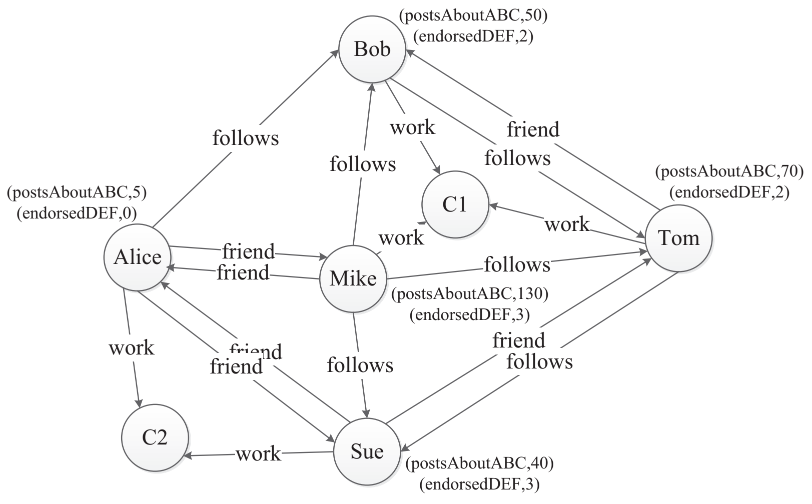


Figure 5.1: Example graph database.

many more properties about Bob. We may have a postsAboutX *attribute for many different topics X. And likewise, we may track his expressed sentiments on many different topics T via a set of* endorsedT *properties.*

Example 5.1.3. *Figure 5.2 shows the four query blocks we use in our running example.*

Given two query blocks $QB = \langle V_{QB}, E_{QB}, \chi_{QB} \rangle$ and $QB' = \langle V_{QB'}, E_{QB'}, \chi_{QB'} \rangle$,

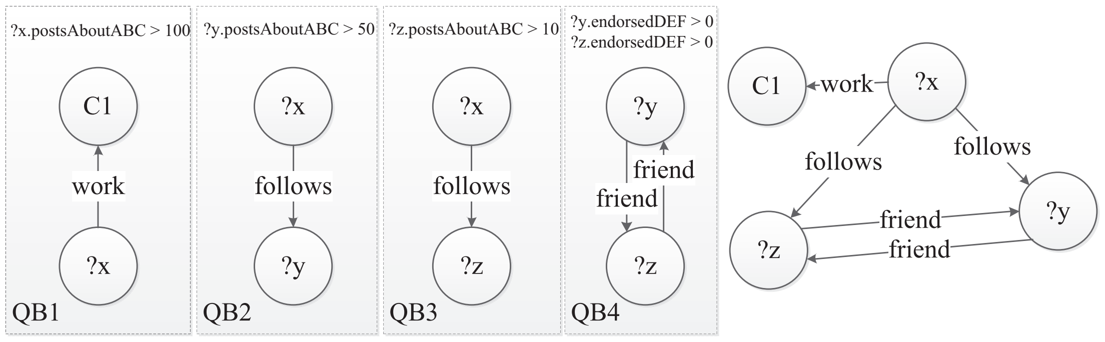


Figure 5.2: Example query blocks (left) and corresponding query graph (right).

we define the union of QB and QB' as the query block $\langle V_{QB} \cup V_{QB'}, E_{QB} \cup E_{QB'}, \chi_{QB} \cup \chi_{QB'} \rangle$.

A (partial) *substitution* θ is a (partial) mapping $\theta : \mathsf{VAR} \to V$ where $dom(\theta) \subseteq \mathsf{VAR}$ denotes the set of variables for which θ is defined. Applying θ to an expression e (vertex, term, or constraint) means replacing every variable $?x \in dom(\theta)$ occurring in it with vertex name $\theta(?x)$, and the expression resulting from such an application is denoted as $e\theta$. The composition of two substitutions θ and θ' with disjoint domains, denoted $\theta\theta'$, is the substitution that maps each variable $?x \in dom(\theta)$ to $?x\theta$ and each variable $?x \in dom(\theta')$ to $?x\theta'$. A ground expression is an expression with no variables — ground terms and ground constraints are evaluated in the obvious way.

Definition 5.1.4 (Answer Substitution for a Query Block)**.** *Suppose $QB = \langle V_{QB}, E_{QB}, \chi_{QB} \rangle$ is a query block. An* answer substitution *θ for QB is a substitution such that (i) $dom(\theta) = \mathsf{VAR}_{QB}$, (ii) for each $(\alpha, SL, \beta) \in E_{QB}$, there is an edge (v_1, ℓ, v_2) in $\mathcal{G}$ such that $v_1 = \alpha\theta$, $v_2 = \beta\theta$, and $\ell \in SL$ if $SL \neq \{*\}$; (iii) for each constraint $C \in \chi_{QB}$, it is the case that $C\theta$ evaluates to true.*

Example 5.1.5. *Consider query block QB_3 of our running example. The answer substitutions for QB_3 are shown below.*

	θ_1	θ_2	θ_3	θ_4	θ_5
$?x$	*Bob*	*Mike*	*Mike*	*Mike*	*Tom*
$?z$	*Tom*	*Bob*	*Tom*	*Sue*	*Sue*

Note that $\{?x/Bob, ?z/Alice\}$ is not an answer substitution for QB_3 because the constraint $?z.postsAboutABC > 10$ is violated, as $\wp(Alice, postsAboutABC) = 5$.

The existence of an answer substitution θ for a query block QB guarantees that there is a way of mapping all the variables in QB to vertices of the graph database so that it satisfies all other requirements specified by QB: presence of vertex names specified in the query and satisfaction of the constraints. We use $\mathsf{AS}(QB)$ to denote the set of all answer substitutions for QB. A substitution θ is said *partial for QB* if $dom(\theta) \neq \mathsf{VAR}_{QB}$.

Definition 5.1.6 (Scoring Function)**.** *Let $QB = \langle V_{QB}, E_{QB}, \chi_{QB} \rangle$ be a query block and θ a substitution in $\mathsf{AS}(QB)$. A* scoring function *SF is a function that assigns a value $SF(QB, \theta) \in \mathbb{R}$ to the pair (QB, θ).*

We assume that scoring functions are *monotonic*. If we express $SF(QB, \theta)$ as a function from $\mathbb{R}^h$ to $\mathbb{R}$ whose arguments are obtained from vertices/edges in the query block, we say that $SF(QB, \theta)$ is monotonic if there exists a partial order $\leq_m$ over $\mathbb{R}^h$ such that $\forall X, Y \in \mathbb{R}^h$, it holds that $X \leq_m Y \Rightarrow SF(X) \leq SF(Y)$.

Example 5.1.7. *In our running example, a possible scoring function is $SF(QB_3, \theta) = e^{-10^3/p}$ with $p = \wp(?z\theta, postsAboutABC) + \wp(?y\theta, postsAboutABC)$. This function is monotonic because if we write p as $a + b$ then the needed partial order $\leq_m$ is $[a, b] \leq_m [a', b']$ if $a \leq a'$ and $b \leq b'$.*

Definition 5.1.8 (Scoring Query Block)**.** *A* scoring query block *is a triple $SQB = \langle QB, SF, V^a \rangle$ where QB is a query block, SF is a scoring function, and $V^a \subseteq \mathsf{VAR}_{QB}$.*

Example 5.1.9. *In our running example, two possible scoring query blocks are $SQB_{3,1} = \langle QB_3, SF_{3,1}, \emptyset \rangle$ and $SQB_{3,2} = \langle QB_3, SF_{3,2}, \{?z\} \rangle$ where $SF_{3,1}(QB_3, \theta)$ is the scoring function of Example 5.1.7 and $SF_{3,2}(QB_3, \theta) = \wp(?z\theta, postsAboutABC)$.*

Our notion of scoring query blocks is rich enough to express normalized "preference" scoring, capturing the preference expressed by the user over a given subgraph of the graph database.

Definition 5.1.10 (Preference Scoring)**.** *Given a scoring query block $SQB = \langle QB, SF, V^a \rangle$ and a substitution $\theta \in \mathsf{AS}(QB)$, the* preference scoring *of θ in SQB*

is

$$SF(SQB,\theta) = \begin{cases} SF(QB,\theta), & \textit{if } V^a = \emptyset \\ \frac{SF(QB,\theta)}{\sum_{\theta' \in \mathsf{AS}(QB\theta^{V^a})} SF(QB,\theta^{V^a}\theta')}, & \textit{otherwise,} \end{cases}$$

where θ^{V^a} is a partial substitution for QB such that $dom(\theta^{V^a}) = dom(\theta) \setminus V^a$ and $\forall ?x \in dom(\theta^{V^a})$, $\theta^{V^a}(?x) = \theta(?x)$.

Intuitively, if the set V^a is not empty, then we normalize the scores of all substitutions for the query block $QB\theta^{V^a}$ in order to get a preference score. We refer to this as *normalized score*.

Example 5.1.11. *Continuing our running example, suppose we want to compute the preference scoring of substitution $\theta_2 = \{?x/Mike, ?z/Bob\}$ in $SQB_{3,1}$ and $SQB_{3,2}$. We have $SF(SQB_{3,1},\theta_2) = e^{-10^3/p} = 0.0038$ with $p = \wp(?z\theta, postsAboutABC) + \wp(?y\theta, postsAboutABC) = 180$. In order to compute $SF(SQB_{3,2},\theta_2)$ we need to look at the answer substitutions in $\mathsf{AS}(QB_3\theta^{V^a})$ where θ^{V^a} is $\{?x/Mike\}$ — therefore, $QB_3\theta^{V^a}$ is the third query block of Figure 5.2 after replacing $?x$ with $Mike$. The set $\mathsf{AS}(QB_3\theta^{V^a})$ contains the substitutions $\theta' = \{?z/Bob\}$, $\theta'' = \{?z/Tom\}$, and $\theta''' = \{?z/Sue\}$. Thus, we have*

$$\begin{aligned} SF(SQB_{3,2},\theta_2) &= \frac{SF_{3,2}(QB_3,\theta_2)}{SF_{3,2}(QB_3,\theta^{V^a}\theta') + SF_{3,2}(QB_3,\theta^{V^a}\theta'') + SF_{3,2}(QB_3,\theta^{V^a}\theta''')} \\ &= \frac{\wp(Bob, postsAboutABC)}{\sum_{v\in\{Bob,Tom,Sue\}} \wp(v, postsAboutABC)} = 0.3333 \end{aligned}$$

Note that $\theta^{V^a}\theta' = \theta_2$, $\theta^{V^a}\theta'' = \theta_3$, and $\theta^{V^a}\theta''' = \theta_4$.

We now define scoring queries.

Definition 5.1.12 (Scoring Query). *A* scoring query *is a tuple $SQ = \langle S, F, \chi^g, SF^g\rangle$ where:*

- *$S = \{SQB_1, \ldots, SQB_m\}$ with $SQB_i = \langle QB_i = \langle V_{QB_i}, E_{QB_i}, \chi_{QB_i}\rangle, SF_i, V_i^a\rangle$.*
- *$F \subseteq S$ and for each $\langle V_{QB_i}, E_{QB_i}, \chi_{QB_i}\rangle \in F$, $V_{QB_i} \cap V \neq \emptyset$.*
- *χ^g is a set of global constraints on $\bigcup_{i=1}^m V_{QB_i}$.*
- *$SF^g : \{0,1\}^m \times [0,1]^m \to [0,1]$ is a "global" scoring function that combines the scores from the scoring query blocks.*

Intuitively, this query defines a set S of scoring query blocks, a subset F of which must be matched. It includes constraints χ^g on the variables in S and a global scoring function that merges the scores from each scoring query block. We now define the concept of a "valid" subset of a scoring query.

Definition 5.1.13 (Valid Subset of a Scoring Query)**.** *Given a scoring query* $SQ = \langle S, F, \chi^g, SF^g \rangle$*, a* valid subset of SQ *is a set* $Sub^{SQ} \subseteq S$ *such that:*

- $F \subseteq Sub^{SQ}$*;*
- *For each* $SQB_h \in Sub^{SQ} \setminus F$*, there exists a sequence* $< SQB_1, \ldots, SQB_r >$ *with* $SQB_i \in Sub^{SQ} \setminus (F \cup \{SQB_h\})$ *s.t. (i)* $\exists\ SQB_l \in F$ *s.t.* $\mathsf{VAR}_{QB_l} \cap \mathsf{VAR}_{QB_1} \neq \emptyset$*, (ii)* $\forall i \in \{1, \ldots, r-1\}$*,* $\mathsf{VAR}_{QB_i} \cap \mathsf{VAR}_{QB_{i+1}} \neq \emptyset$*, and (iii)* $\mathsf{VAR}_{QB_r} \cap \mathsf{VAR}_{QB_h} \neq \emptyset$*.*

The second condition in the above definition says that for each scoring query block $SQB_h \in Sub^{SQ} \setminus F$ there exists a sequence of scoring query blocks in $Sub^{SQ} \setminus (F \cup \{SQB_h\})$ that connects the variables in SQB_h with the variables in the query blocks in F. The set of all valid subsets of SQ is denoted by $USub^{SQ}$.

Example 5.1.14. *Consider the scoring query* $SQ = \langle \{SQB_1, SQB_2, SQB_3, SQB_4\}, \{SQB_1\}, \emptyset, SF^g \rangle$ *where* SQB_1*,* SQB_2*,* SQB_3*, and* SQB_4 *are associated with* QB_1*,* QB_2*,* QB_3*, and* QB_4 *in Figure 5.2, respectively. We have* $USub^{SQ} = \{\{SQB_1\}, \{SQB_1, SQB_2\}, \{SQB_1, SQB_3\}, \{SQB_1, SQB_2, SQB_4\}, \{SQB_1, SQB_3, SQB_4\}, \{SQB_1, SQB_2, SQB_3, SQB_4\}\}$*. Note that the subset* $\{SQB_1, SQB_4\}$ *is not valid because there is no connection between the variables in* SQB_1 *and* SQB_4*.*

Definition 5.1.15 (Substitution for a Scoring Query)**.** *Suppose* $SQ = \langle S, F, \chi^g, SF^g \rangle$ *is a scoring query and* $Sub^{SQ} \in USub^{SQ}$*. Let* $mod(SQ, Sub^{SQ})$ *be the query block obtained as* $\{\langle \emptyset, \emptyset, \chi^g \rangle\} \cup \bigcup_{\langle QB, \cdot, \cdot \rangle \in Sub^{SQ}} QB$*. A* substitution for SQ w.r.t. Sub^{SQ} *is a substitution in* $\mathsf{AS}(mod(SQ, Sub^{SQ}))$*.*

Definition 5.1.16 (Score of a Substitution)**.** *Suppose we are given a scoring query* $SQ = \langle S, F, \chi^g, SF^g \rangle$*, a valid subset* Sub^{SQ} *of* SQ*, and a substitution* θ *for* SQ *w.r.t.* Sub^{SQ}*. The score of* Sub^{SQ} *and* θ *is* $P(Sub^{SQ}, \theta) = SF^g(D, M)$ *where* D *and* M *are two vectors (in* $\{0,1\}^m$ *and* $[0,1]^m$*, respectively) such that if* $SQB_i \notin Sub^{SQ}$*, then* $d_i = 0$ *and* $m_i = 0$*; otherwise,* $d_i = 1$ *and* $m_i = SF(\langle mod(SQ, Sub^{SQ}), SF_i, V_i^a \rangle, \theta)$*.*

We assume SF^g is monotonic w.r.t. D and M, i.e. $SF^g(D, M) \geq SF^g(D', M')$ whenever $d_i \geq d'_i$ and $m_i \geq m'_i$ for all i, where $d_i \in D$, $d'_i \in D'$, $m_i \in M$, and $m'_i \in M'$.

We are now ready to define the top-k preferred answers to a scoring query.

Definition 5.1.17 (Top-k Answers to a Scoring Query). *Given a scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$ and a non-negative integer k, we define the* top-k answers to *SQ as the set $Top\text{-}k = \{(Sub_1^{SQ}, \theta_1), \ldots, (Sub_k^{SQ}, \theta_k)\}$ such that:*

- *$Sub_i^{SQ} \in USub^{SQ}$ and θ_i is a substitution for SQ w.r.t. Sub_i^{SQ} (for each $i \in \{1, \ldots, k\}$);*
- *for each pair (Sub^{SQ}, θ) were θ is a substitution for SQ w.r.t. Sub^{SQ} and $(Sub^{SQ}, \theta) \notin Top\text{-}k$, we have that $\forall (Sub_i^{SQ}, \theta_i) \in Top\text{-}k$, $P(Sub_i^{SQ}, \theta_i) \geq P(Sub^{SQ}, \theta)$.*

Example 5.1.18. *Assume that, for all query blocks of our running example, we want to compute a score using the sigmoid function $sigmoid(t) = \frac{1}{1+e^{-t}}$ over the sum of variables denoting* postsAboutABC *property values. As our global scoring function, we want to use the sigmoid function over the average of D, multiplied by the average of M. In this case we have:*

- $SQB_1 = \langle QB_1, SF_1 = sigmoid(\frac{?x.postsAboutABC}{1K}), V_i^a = \{?x\} \rangle$
- $SQB_2 = \langle QB_2, SF_2 = sigmoid(\frac{?x.postsAboutABC + ?y.postsAboutABC}{1K}), V_i^a = \emptyset \rangle$
- $SQB_3 = \langle QB_3, SF_3 = sigmoid(\frac{?x.postsAboutABC + ?z.postsAboutABC}{10K}), V_i^a = \emptyset \rangle$
- $SQB_4 = \langle QB_4, SF_4 = sigmoid(\frac{?z.postsAboutABC + ?y.postsAboutABC}{10K}), V_i^a = \emptyset \rangle$
- $SQ = \langle \{SQB_1, SQB_2, SQB_3, SQB_4\}, F = \{SQB_1\}, \chi^{\mathcal{G}}, SF^{\mathcal{G}} = sigmoid(\frac{\sum_{d_i \in D} d_i}{4} \times \frac{\sum_{m_i \in M} m_i}{4})$.

5.2 Baseline Top-k Answering Algorithm

A straightforward algorithm to compute the top-k answers to a scoring query first computes all answer substitutions and then computes the score of each answer substitution in order to find the best k. Any subgraph matching algorithm could be used for this purpose. In this section, we describe our baseline algorithm, which builds upon the DOGMA framework [13] that considers our specific problem situation (queries with constants and large disk-residing graphs).

The following definition merges together a set of scoring query blocks.

Definition 5.2.1 (Unified Query). *Given a scoring query SQ and a set $Sub^{SQ} = \{SQB_1, SQB_2, \ldots, SQB_n\} \in USub^{SQ}$, with $SQB_i = \langle QB_i, SF_i, V_i^a \rangle$ and $QB_i = \langle V_{QB_i}, E_{QB_i}, \chi_{QB_i} \rangle$, the* unified query *of Sub^{SQ} is $unified(Sub^{SQ}) = \langle \bigcup_{i=1}^n V_{QB_i}, \bigcup_{i=1}^n E_{QB_i}, \bigcup_{i=1}^n \chi_{QB_i} \rangle$.*

Algorithm 7 is our Base algorithm. For each valid subset Sub^{SQ} of SQ, answer substitutions are found by applying findSubstitutions to the unified query. A score for each answer substitution is computed, and finally the top-k are returned. In the algorithm, $dom(\theta_\emptyset) = \emptyset$.

Algorithm 7: Base

```
1  FUNCTION map ($SQ$)
   Input: Scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$
   Output: Top-$k$ answers and their scores
2  $S \leftarrow \emptyset$
3  foreach $Sub^{SQ} \in USub^{SQ}$ do
4      foreach $\theta \in$ findSubstitutions($unified(Sub^{SQ}), \theta_\emptyset$) do
5          $S \leftarrow S \cup \{$ ($\theta$, computeScore($Sub^{SQ}, \theta$))$\}$
6  return top-$k$ of $S$
7  FUNCTION findSubstitutions ($Q, \theta$)
   Input: Unified query $Q = \langle V_Q, E_Q, \chi_Q \rangle$, partial substitution $\theta$
   Output: Set $AS$ of answer substitutions obtained by extending $\theta$
8  if $\theta$ maps every variable in $Q$ then
9      return $\theta$
10 $NV \leftarrow \{(c, ?v) | (?v, c)$ or $(c, ?v) \in E_Q\}$
11 foreach $(c, ?v) \in NV$ do
12     $R_{c,?v} \leftarrow$ getNeighborNum($c$, getEdgeLabel($c$, $?v$))
13     if $R_{c,?v} = 0$ then return $\emptyset$
14 $(c, ?w) \leftarrow (c, ?v) \in NV$ with minimum $R_{c,?v}$
15 $N_{?w} \leftarrow$GetValidNeighbors($Q$, $c$, $?w$)
16 $AS \leftarrow \emptyset$
17 foreach $m \in N_{?w}$ do
18     $\theta' \leftarrow \theta \cup \{?w/m\}$
19     $AS \leftarrow AS\cup$ map($Q\theta'$,$\theta'$)
20 return $AS$
```

The findSubstitutions function uses function getNeighborNum that returns the number of c's neighbors which are connected through an edge with a given label.

Moreover, function `GetValidNeighbors`, when applied to a unified query Q, a vertex c, and a variable $?w$, returns the neighbors of c that can be mapped to $?w$ given E_Q and χ_Q. At this point, we access the disk index and also use DOGMA's pruning technique based on IPD values to filter neighbors that cannot be part of a valid answer (see Section 5.3.1). .

Instead of doing massive table-joins [9, 21, 55], findSubstitutions uses DOGMA to perform a depth-first graph traversal search from constants. Line 8 checks whether a complete substitution for the unified query has been generated. Line 10 inspects every edge of the having one end mapped to a vertex of the graph and not the other. $R_{c,?v}$ is the number of c's neighbors in the graph that are connected through an edge having the same label as the one between c and $?v$ — therefore, $R_{c,?v}$ is the number of candidates for $?v$. In line 14 we select the edge in the subgraph query with the lowest number of candidates. In line 17, we substitute $?w$ with each candidate m and recursively continue the assembly of answers.

5.3 ATK Algorithm

This section describes the proposed advanced pruning ATK algorithm. We start by briefly discussing the index structure used by the algorithm for both subgraph matching and the computation of upper bounds on scores. We then move on to describe the overall structure of the algorithm and its main components.

5.3.1 ATK Graph Database Index

ATK employs a disk-based index that leverages the DOGMA index and extends it to store additional information. DOGMA decomposes the data graph into a large number of small, densely connected subgraphs and stores them in a binary tree structured index. Initially, it iteratively halves the number of vertices by merging randomly selected vertices with all of their neighbors. This process is repeated till a graph with less than or equal to Z vertices is reached. The graph DBMS administrator can choose Z so that graphs will fit into a disk page and their partitioning can be done in reasonable time (in our experiments, we set $Z = 100$). After reaching a single graph, DOGMA builds a binary tree by iteratively partitioning each graph into two in order to minimize edge cut. Each node of the index tree thus represents a subgraph of the original graph database. Leaves of the tree correspond to disk pages.

DOGMA has been shown to be able to increase the I/O-efficiency by exploiting data locality. Moreover, for every vertex, it stores the *internal partition distance* (IPD), i.e. the number of hops from the vertex to the nearest other vertex outside the disk page. Using IPD, a minimum distance between vertices can quickly be computed, and candidates can be pruned if their distance is higher than the distance between their respective query graph variables. For example, assume a partial substitution θ where $?v$ is mapped to c_1 and $?u$ to c_2 with $dist(?v, ?u) = 4$ (where $dist(x, y)$ denotes the length of the shortest path between x and y), $IPD(c_1) = 2$ and $IPD(c_2) = 3$, and c_1 and c_2 reside on different disk pages. This partial substitution cannot create a complete substitution because $dist(?v, ?u) < IPD(c_1) + IPD(c_2)$, so it can be pruned right away.

We make the following extension to DOGMA in order to support scoring queries. For each leaf node N, we store the edges that connect the node to other leaf nodes, along with aggregates (maximum, minimum, and average) of the properties of the vertices represented by N or adjacent to a vertex in N (called "boundary" vertices). As it will be clearer in the following, this additional information allows the ATK algorithm to compute upper bounds much more efficiently.

5.3.2 Overall Structure of ATK

The ATK algorithm, given a scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$, works by recursively extending pairs of the form (Sub^{SQ}, θ), where $Sub^{SQ} \in USub^{SQ}$ and θ is a (partial) substitution for $unified(Sub^{SQ})$. The algorithm is based on two main operations: *block extension* and *variable extension.*

Definition 5.3.1 (Block Extension, Variable Extension)**.** *Consider a pair (Sub^{SQ}, θ) where $Sub^{SQ} \in USub^{SQ}$ and θ is a (partial) substitution for $unified(Sub^{SQ})$. Suppose $SQB \in S \setminus Sub^{SQ}$ is a scoring query block such that $Sub^{SQ} \cup \{SQB\} \in USub^{SQ}$. The result of block extension is a pair $(Sub^{SQ} \cup \{SQB\}, \theta)$. Now assume θ is partial for $unified(Sub^{SQ})$. The result of variable extension is a pair (Sub^{SQ}, θ') where $dom(\theta') = dom(\theta) \cup \{?x\}$ and $?x \in \mathsf{VAR}_{unified(Sub^{SQ})} \setminus dom(\theta)$.*

Intuitively, block extension adds a new scoring query block to Sub^{SQ} without changing θ. Variable extension maps a variable that is not already mapped by θ.

The ATK algorithm starts by calling extendVariable $(SQ, F, \theta_\emptyset)$ and proceeds by alternately performing block and variable extensions (Algorithm 8). For each kind of extension, ATK needs to choose which block to extend, or which variable and associated vertex in the graph database to match the variable. The algorithm continuously maintains the top-k answers computed so far along with their associated scores — this is data structure T in Algorithm 8. In order to prune, ATK compares an upper bound of the score of (Sub^{SQ}, θ) with the minimum score of the answers in T. The latter is updated whenever a new answer substitution is found.

The next few sections discuss the block and variable extension operations in detail.

5.3.3 Block Extension

Block extension is implemented by the extendBlock function. The set EB contains scoring query blocks that can be used to extend (Sub^{SQ}, θ). When we extend (Sub^{SQ}, θ) to $(A = Sub^{SQ} \cup \{SQB\}, \theta)$, we have to consider three cases. Let us first assume $|T| = k$, which means T already stores k answers. In the first case, biggestUpperBound $(A, \theta) < \min_{(\cdot,\cdot,p)\in T} p$. This means that it is not possible to find *any valid superset* of Sub^{SQ} that can be used to extend θ whose score is greater than at least one substitution in T. In this case, we completely prune away this search path. In the second case, we have that upperBound $(A, \theta) < \min_{(\cdot,\cdot,p)\in T} p$, meaning that no substitution extending θ w.r.t. A has a score greater than at least one substitution in T. In this case, we further extend A by adding other blocks and without changing θ. This operation is done by recursively calling extendBlock (SQ, A, θ). If neither the first nor the second case above applies, or if $|T| < k$, we extend θ w.r.t. A by calling extendVariable (SQ, A, θ).

The upperBound function is used to compute an upper bound on the scores of

Algorithm 8: ATK block and variable extension functions

```
Data: $T$: global data structure that maintains the top-$k$ preferred answers,
      as triples of the form $(Sub^{SQ}, \theta, p)$. $R$: global data structure that
      maintains the already examined valid subsets of $SQ$

1  FUNCTION extendBlock $(SQ, Sub^{SQ}, \theta)$
   Input: Scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$, set $Sub^{SQ} \in USub^{SQ}$, partial
          substitution $\theta$
2  $EB \leftarrow \{SQB | SQB \in S \setminus Sub^{SQ}, Sub^{SQ} \cup \{SQB\} \in USub^{SQ}\}$
3  foreach $SQB \in EB$ do
4      $A \leftarrow Sub^{SQ} \cup \{SQB\}$
5      if $A \notin R$ then
6          if $|T| = k$ and biggestUpperBound$(A, \theta) < \min_{(\cdot,\cdot,p) \in T} p$ then
7              return
8          else if $|T| = k$ and upperBound $(A, \theta) < \min_{(\cdot,\cdot,p) \in T} p$ then
9              extendBlock$(SQ, A, \theta)$
10         else
11             extendVariable$(SQ, A, \theta)$
12         $R \leftarrow R \cup \{A\}$

13 FUNCTION extendVariable $(SQ, Sub^{SQ}, \theta)$
   Input: Scoring query $SQ = \langle S, F, \chi^g, SF^g \rangle$, set $Sub^{SQ} \in USub^{SQ}$, partial
          substitution $\theta$
14 if $dom(\theta) = \mathsf{VAR}_{unified(Sub^{SQ})}$ then
15     updateT $(Sub^{SQ}, \theta,$computeScoreAdvanced$(Sub^{SQ}, \theta))$
16     if $(Sub^{SQ} = F)$ then $R \leftarrow \emptyset$
17     extendBlock$(SQ, Sub^{SQ}, \theta)$
18     return
19 $NV \leftarrow \{(c, ?v) | c$ and $?v$ are neighbors in $Sub^{SQ}$ and $?v \notin dom(\theta)\}$
20 $NV \leftarrow NV \cup \{(\theta(?z), ?v) | ?z$ and $?v$ are neighbors in $Sub^{SQ}$, $?z \in dom(\theta)$,
   and $?v \notin dom(\theta)\}$
21 foreach $(c, ?v) \in NV$ do
22     $R_{c,?v} \leftarrow$ getNeighborNum($c$, getEdgeLabel($c$,$?v$))
23     if $R_{c,?v} = 0$ then return
24 choose $(c, ?w) \in arg\,max_{(c,?v) \in NV} R_{c,?v}$
25 $N_{?w} \leftarrow$GetValidNeighbors($Sub^{SQ}$, $c$, $?w$)
26 if $(|T| = k)$ then
27     $N_{?w} \leftarrow \{m | m \in N_{?w}$ and biggestUpperBound$(Sub^{SQ}, \theta \cup \{?w/m\})$
       $> \min_{(\cdot,\cdot,p) \in T} p\}$
28 Sort $N_{?w}$ by descending order of biggestUpperBound$(Sub^{SQ}, \theta \cup \{?w/m\})$ for
   all $m \in N_{?w}$
29 foreach $m \in N_{?w}$ do
30     $\theta' \leftarrow \theta \cup \{?w/m\}$
31     extendVariable($SQ$,$Sub^{SQ}$, $\theta'$)
```

the extensions of θ w.r.t. any given Sub^{SQ} as follows. To build the D and M vectors to be used as in Definition 5.1.16, we set $m_i = d_i = 0$ for $SQB_i \notin Sub^{SQ}$ and $d_i = 1$ for $SQB_i \in Sub^{SQ}$. When computing m_i for a $SQB_i \in Sub^{SQ}$, the upper bound is simply set to 1 if $V_i^a \neq \emptyset$. Otherwise, we compute an upper bound for the extensions of θ by deriving an upper bound for each variable $?v$'s property. If $?v \in dom(\theta)$, then we take the exact value by looking at $?v\theta$. For the candidate vertex set of a variable $?v \notin dom(\theta)$,[1] we could in principle retrieve all their properties and take the maximum value — however, this would require a prohibitively expensive retrieval. We find good upper bounds more efficiently by using the maximum values stored by the ATK index along with disk pages. For a variable $?v \notin dom(\theta)$, we look at its distance in $unified(Sub^{SQ})$ to all vertices $c \in dom(\theta)$. If $dist(c, ?v) < IPD(c)$ for some c, then we know that $?v$ has to be mapped in the same disk page as c. We can then use the maximum values stored along with the disk pages where c resides. If multiple maximum values are possible, we take the minimum. If $dist(c, ?v) < IPD(c) + 1$, we do the same but use the maximum values of the disk page together with its boundary vertices. If instead $dist(c, ?v) > IPD(c) + 1$ for all c, we have no local information so we use global maximum values. We finally obtain the upper bound using these exact and maximum values.

Example 5.3.2. *Suppose we have three query blocks* $SQB_1 = \langle p \rightarrow ?x, SF_1, \emptyset\rangle$, $SQB_2 = \langle ?x \rightarrow ?a \rightarrow ?b, SF_2, \emptyset\rangle$ *and* $SQB_3 = \langle ?x \rightarrow ?b, SF_3, \emptyset\rangle$, *and suppose* $IPD(c) = 2$. *Let* $\theta = \{?x/c\}$ *be an answer substitution for* SQB_1. *Suppose we want to compute an upper bound for the score of extensions of* θ *w.r.t.* $Sub^{SQ} = \{SQB_1, SQB_2\}$. *We have that* $dist(c, ?a) = 1$ *and* $dist(c, ?b) = 2$ *in* $unified(Sub^{SQ})$. *In this case,* $D = (1, 1, 0)$ *and* $M = (m_1, m_2, 0)$, *where* m_1 *is the exact score computed from c's property values and* m_2 *is the maxi-*

[1] We can get all candidates by checking the distance. For example, let c be a constant and $h = dist(c, ?v)$. We retrieve all neighbors of c whose distance to c is less than h in the graph.

mum score computed from upper bounds on ?a's and ?b's property values. Since $dist(c, ?a) < IPD(c)$, we know that ?a is mapped to the same disk page as c, so we can retrieve the maximum values of its properties from the index. Moreover, since $dist(c, ?b) < IPD(c) + 1$, variable ?b is mapped to the same disk page plus its boundary vertices, so again we retrieve the upper bound directly from the index.

Function biggestUpperBound, given a valid subset Sub^{SQ} and a partial substitution θ, returns the biggest possible upper bound of θ, i.e. an upper bound that is greater than or equal to the upper bounds of any extension of θ w.r.t. Sub'^{SQ}, where $Sub^{SQ} \subset Sub'^{SQ} \subseteq S$. This computation is performed by defining a distance $dist'$ from c to $?v$ for every valid subset $Sub^{SQ} \in USub^{SQ}$ as $dist'(c, ?v) = \max\{dist(c, ?v) \mid dist(c, ?v)$ is computed w.r.t. $unified(Sub^{SQ})\}$ and using its values for computing m_i. Note that we can pre-compute $dist'(c, ?v)$ between every pair of vertices/variables in $unified(Sub^{SQ})$. The unified query usually has relatively few vertices — even very complex unified queries will likely have 20-30 vertices, with 100 being a generous estimate of the maximum size. So the quadratic computation is not prohibitive.

The following result ensures that our computation is correct.

Proposition 5.3.1. *Given a scoring query SQ, a set $Sub^{SQ} \in USub^{SQ}$, and a substitution θ for $unified(Sub^{SQ})$,* biggestUpperBound (Sub^{SQ}, θ) *is greater than or equal to the upper bound of any extension of θ w.r.t. any Sub'^{SQ} such that $Sub^{SQ} \subset Sub'^{SQ} \in USub^{SQ}$.*

Proof. Let $D' = (d'_0, d'_1, \ldots)$ and $M' = (m'_0, m'_1, \ldots)$ be the two vectors computed by upperBound (S, θ) using $dist'$. Now assume $P^{\mathcal{G}}(D', M')$ is not the biggest upper bound, which means there exists a valid subset $Sub'^{SQ} \supset Sub^{SQ}$ such that an extension of θ w.r.t. Sub'^{SQ} has a higher score. Let $M = (m_0, m_1, \ldots)$ be the vector used to compute the upper bound for this extension of θ. We have that $m_i > m'_i$ for some i (since in this case $d'_i = 1$ for all i). However, this implies that there exist a vertex c and a variable $?v$ such that $dist'(c, ?v) < dist(c, ?v)$, which contradicts the definition of $dist'$. □

Finally, function updateT adds triples of the form (Sub^{SQ}, θ, p) to T, also making sure that (i) T always contains k triples at most, and (ii) the triples in T are those with the higher values of p. To this aim, if $|T| = k$, it adds a triple (Sub^{SQ}, θ, p) to T only if $\min_{(\cdot,\cdot,p')\in T} p' < p$ — in this case, it also removes a triple with minimum p'.

5.3.4 Variable Extension

The extendVariable function is similar to findSubstitutions of the Base algorithm, but we add two key parts. First, we call extendBlock if we generate an answer substitution for $unified(Sub^{SQ})$ (line 17). Second, we compute a biggest upper bound when $?w$ is mapped to m and prune the partial substitution if this is smaller than the minimum score in T (line 27) .

In order to compute a normalized score for a scoring query block SQB_i w.r.t. an answer substitution θ (function computeScoreAdvanced) we construct a subquery $QB_i\theta^{V^a}$ and find all its answer substitutions $\mathsf{AS}(QB_i\theta^{V^a})$. While processing $QB_i\theta^{V^a}$, we can ignore all constants that are not directly connected to any variables. By Definition 5.1.10, $\sum_{\theta'\in\mathsf{AS}(QB_i\theta^{V^a})} SF_i(QB_i, \theta^{V^a}\theta')$ will be the denominator. This creates another overhead, but in general, the number of variables in $QB_i\theta^{V^a}$ is small and we can reuse results once $\mathsf{AS}(QB_i\theta^{V^a})$ is computed. In particular, many partial substitutions will be pruned during the depth-first search and only a small number of substitutions will be left to handle in line 15.

Example 5.3.3. *In the case of Example 5.1.18, ATK works as follows. It starts with the call extendVariable $(SQ, \{PQB_1\}, \theta_\emptyset)$, which is depicted as "Start" in Figure 5.3. There are three candidates for variable $?x$ of SQB_1, namely Mike, Tom, and Bob. Only Mike meets the constraint $?x.postsAboutABC > 100$. $\theta_1 = \{?x/Mike\}$ is a complete substitution for $\{SQB_1\}$, so its score is computed and the top-1 is now θ_1. θ_1 is extended to $\{SQB_1, SQB_2\}$, and along the bold path, the following answer* *substitutions are found:*

- $\theta_1 = \{?x/Mike\}$ *for* $\{SQB_1\}$*;*
- $\theta_2 = \{?x/Mike, ?y/Tom\}$ *for* $\{SQB_1, SQB_2\}$*;*
- $\theta_3 = \{?x/Mike, ?y/Tom, ?z/Bob\}$ *for* $\{SQB_1, SQB_2, SQB_4\}$*;*
- $\theta_4 = \theta_3$ *for* $\{SQB_1, SQB_2, SQB_3, SQB_4\}$*.*

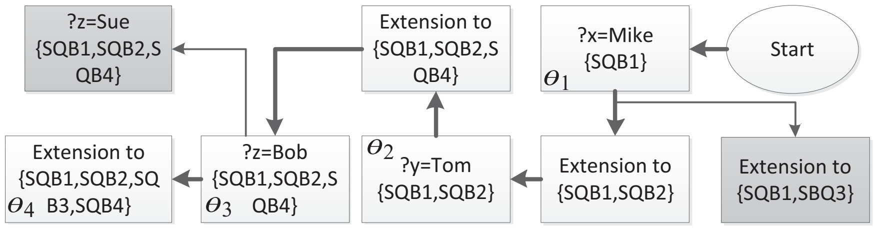


Figure 5.3: Example run of the ATK algorithm. Gray boxes represent pruned search.

Thus, θ_4 *is now the top-1 with a score of 46.7%. The depth-first search now backtracks to the mapping which sets ?z to Sue, but its upper bound score of 46.6% is too low to beat* θ_4 *(because Sue's property value is too low). Thus, this mapping attempt is pruned. The extension of* θ_1 *to* $\{SQB_1, SQB_3\}$ *is also pruned because its upper bound score with* $d_2 = 0$ *and* $d_4 = 0$ *is less than the current top-1's score. Note that we only show the steps that find* θ_4 *and two pruning cases in Figure 5.3 — all other depth-first search cases are omitted in the figure.*

5.4 Experiments

We tested our framework using the 4 graph databases in Table 5.1. For each database, we generated 3000 scoring queries consisting of h query blocks each. We started by randomly retrieving h subgraphs with e edges each, ensuring that each subgraph partially overlapped with at least one of the others. Then we selected a fixed portion of vertices (60% to 80%, depending on the query size and graph database used) from the set of retrieved ones and converted them to variables in all subgraphs. This ensured that the generated queries had at least one answer.

For generating constraints, we selected 30% of variables. For each selected variable $?v$ that replaced a vertex u with property $u.p = z$, we wrote a constraint

	$\lvert V\rvert$	$\lvert E\rvert$
CiteSeerX [27]	0.93M	2.9M
IMDb [37]	2.1M	7.7M
YouTube [25]	4.6M	14.9M
Flickr [19]	6.2M	15.2M

Table 5.1: Graph databases used in the experiments.

whose form is either $?v.p \geq z$ or $?v.p \leq z$ (if u had multiple numerical properties, we chose one randomly). This way, we also guaranteed that there existed at least one substitution for each query block (we actually obtained upto tens of millions of substitutions).

In r out of the h scoring query blocks, we used normalized preference scoring, i.e. we chose a set of $|\mathsf{VAR}_{QB_i}|/2$ variables and designated them as V_i^a. Then we defined:

$$SF_i(SQB_i, \theta) = \frac{1}{1+e^{-t}}, \text{ where } t = \begin{cases} \frac{\sum_{?v \in \mathsf{VAR}_{QB_i}} \wp(?v\theta, property)}{1K}, & \text{if } V_i^a = \emptyset; \\ \frac{\sum_{?v \in V_i^a} \wp(?v\theta, property)}{1K}, & \text{otherwise.} \end{cases}$$

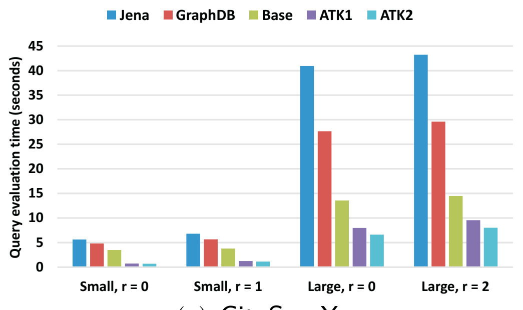
Moreover, we defined $SF^{\mathcal{G}}(D, M) = \frac{1}{1+e^{-t}} \times \frac{\sum_i m_i}{|M|}$, where $t = \sum_i d_i$.

We generated two groups of "small" queries with $h = 2$, $e = 2$, and $r \in \{0, 1\}$, and two groups of "large" queries with $h = 5$, $e = 3$, and $r \in \{0, 2\}$. We compared the performance of the Base algorithm with the ATK algorithm and distinguished two variants of the latter — we call ATK1 the variant where we only prune partial substitutions, and ATK2 the one where we prune both partial substitutions and block extensions. This allowed us to assess whether the effort devoted to pruning block extensions along with partial substitutions pays off in terms of overall query evaluation performance.

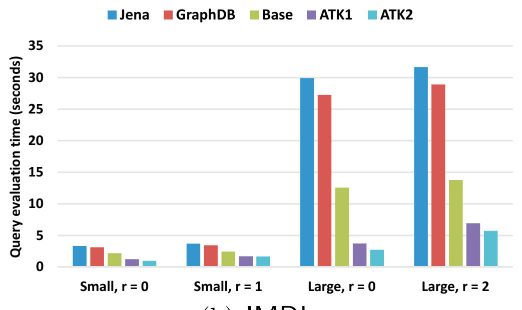
Moreover, we compared ATK with two of the most popular RDF query engines, Jena [1] and GraphDB [51], used as function findSubstitutions in the Base algorithm.

5.4.1 Experimental Results

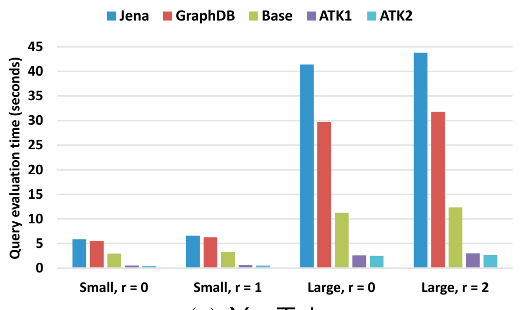
We started by measuring average query evaluation times for different query groups and algorithms, with $k = 5$. The results are reported in Figure 5.4.



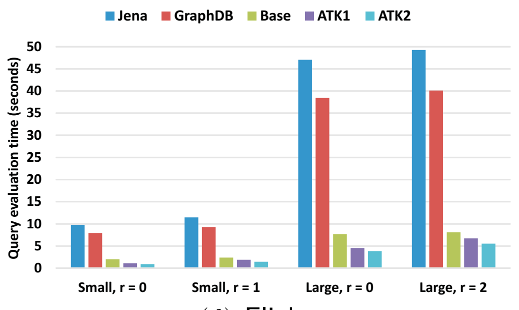
(a) CiteSeerX



(b) IMDb



(c) YouTube



(d) Flickr

Figure 5.4: Query evaluation times for different query groups and algorithms.

In general, the performances obtained by the ATK algorithm appear very satisfactory. ATK2 always outperforms ATK1 (with an average performance improvement around 15%) which confirms that pruning block extensions as well as partial substitutions is always beneficial. In comparison with Base, ATK2 saved 32% to 85% of query evaluation time for small queries, whith an average of 63%. For large queries, it saved 32% to 78% with an average of 59%. In the majority of cases, the performance advantage of ATK2 over Base when $r = 0$ is higher than the one obtained when $r > 0$ — for instance, with large queries over IMDb, the advantage is 78% when $r = 0$ and 58% when $r = 2$. This is natural because the components of the M vector used in function upperBound are simply set to 1 if V^a is not empty. An interesting good result is that ATK2 still shows a good performance advantage over ATK1 when $r > 0$.

Jena and GraphDB need a longer time to answer queries than Base. As indicated in [13, 9, 21, 55], traditional relational storage techniques such as triplestores have lower performance than graph database systems when facing these kinds of queries, due to inefficient table-joins.

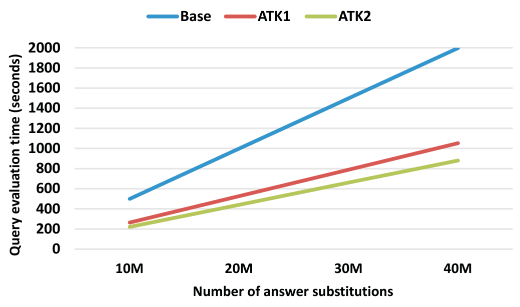
Table 5.2 shows p-values that demonstrate that these results on query evaluation time reduction are statistically significant. The highest one is p_2 on YouTube, which is still lower than the usual significance threshold of 0.05. For all other cases, p-values are much lower than the threshold. This shows that all of our claims of improved efficiency via our two pruning methods are statistically significant as is the assertion that ATK2 is better than ATK1.

	p_1	p_2
CiteSeerX	3×10^{-3}	1.7×10^{-3}
IMDb	5.8×10^{-4}	2.5×10^{-3}
YouTube	1.7×10^{-4}	2.2×10^{-2}
Flickr	2×10^{-3}	7.19×10^{-9}

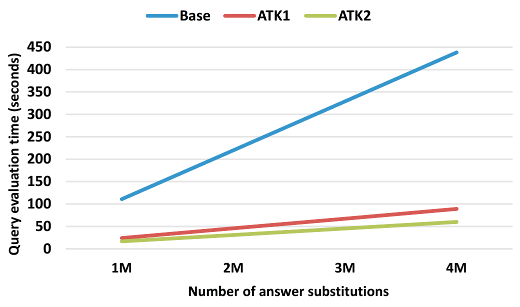
Table 5.2: p-values. p_1 is obtained using the paired t-test between Base and ATK1 for all query groups — p_2 compares ATK1 and ATK2 in the same way.

We also retrieved the number of answer substitutions computed for all queries with $k = 5$ and drew a trend line (linear line with least square error) showing how

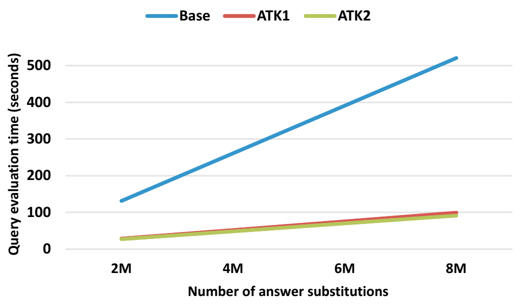
Base, ATK1, and ATK2 scale with the number of answer substitutions. As expected (Figure 5.5) both ATK1 and ATK2 scale much better than Base, showing the best performance on the IMDb and YouTube datasets. Again, ATK2 outperforms ATK1 in all cases.



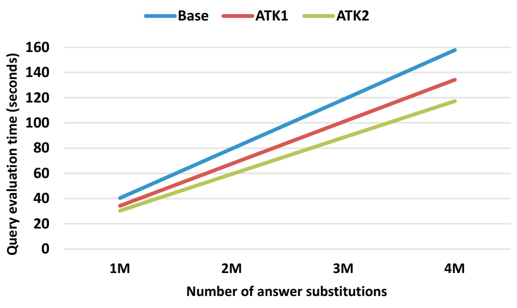
(a) CiteSeerX



(b) IMDb



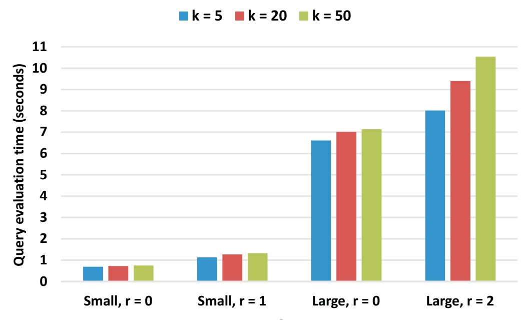
(c) YouTube



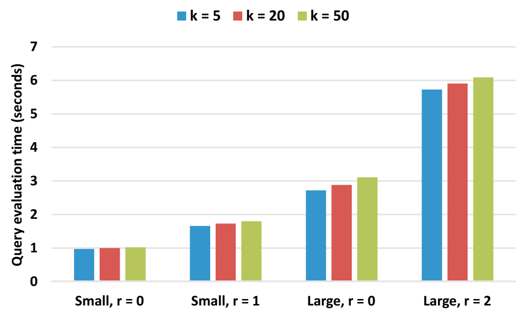
(d) Flickr

Figure 5.5: Trend lines for query evaluation times vs. number of answer substitutions.

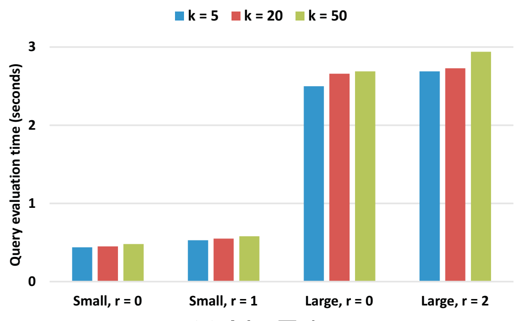
Finally, we assessed how the query evaluation times of ATK2 scale with the value of k. The results are shown in Figure 5.6. The algorithm appears to scale gracefully — the increment in evaluation time is over 10% in just 5 out of 48 cases. On average, the increment was less than 6% when moving from $k = 5$ to $k = 20$ and less that 5% when moving from $k = 20$ to $k = 50$. Interestingly, the average increment was just 11.03% even when moving from $k = 5$ to $k = 50$.



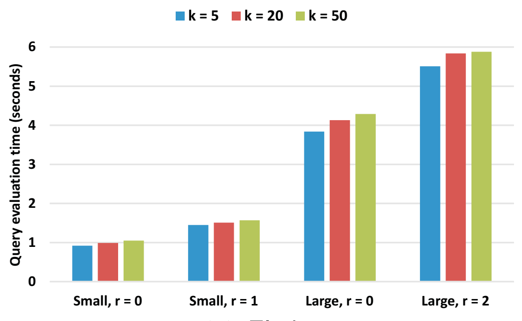
(a) CiteSeerX



(b) IMDb



(c) YouTube



(d) Flickr

Figure 5.6: Query evaluation times of ATK2 for different query groups and values of k.

Chapter 5: Conclusion

Edge-labeled graphs are occurring increasingly frequently. In many applications, it is appropriate to allow a user to specify functions that capture what is important for him, as opposed to ramming a top-down definition of importance down the user's throat. In this thesis research, we argue that the user may express his notion of importance via a set of subgraph matching queries and scorring/aggregating functions. To this end, we propose four different top-k importance computation models, which spans even aggregated, approximate and probabilistic top-k queries. We carried out detailed experiments with real-world data from CiteSeerX, IMDb, YouTube, Flickr, GovTrack, SPINN, BSBM, and Orkut in order to evaluate performance. The datasets involved graph databases with upto 234M edges. Our experiments show that answers on these real-world data sets can be computed in very good time frames.

Bibliography

[1] Apache Jena. https://jena.apache.org.

[2] Apache Jena TopN Optimization. https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/sparql/algebra/optimize/TransformTopN.html.

[3] Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm.

[4] Neo4j. http://neo4j.com.

[5] Social Network Intelligence Benchmark (SIB). http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark.

[6] The SP^{2}Bench SPARQL Performance Benchmark. http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B.

[7] 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.

[8] Ian A. Andrews, Srijan Kumar, Francesca Spezzano, and V. S. Subrahmanian. SPINN: suspicion prediction in nuclear networks. In *ISI*, 2015.

[9] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629–638, 2012.

[10] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[11] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[12] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. Probabilistic Subgraph Matching on Huge Social Networks. In *2011 International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 271–278, 2011.

[13] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases. In *The Semantic Web – ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 97–113. 2009.

[14] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In *ASONAM*, pages 248–255, 2010.

[15] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *ICDE Workshops*, pages 94–99, 2011.

[16] 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.

[17] Nicolas Bruno and Hui (Wendy) Wang. The threshold algorithm: From middleware systems to the relational engine. *IEEE Trans. on Knowl. and Data Eng.*, 19(4):523–537, April 2007.

[18] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *WWW*, pages 74–83, 2004.

[19] Meeyoung Cha, Alan Mislove, and P. Krishna Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *WWW*, pages 721–730, 2009.

[20] James Cheng, Yiping Ke, and Wilfred Ng. Efficient query processing on graph databases. *ACM Trans. Database Syst.*, 34(1), 2009.

[21] Jiefeng Cheng and Jeffrey Xu Yu. A Survey of Relational Approaches for Graph Pattern Matching over Large Graphs. In *Graph Data Management*, pages 112–141. 2011.

[22] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast Graph Pattern Matching. In *ICDE Conf.*, pages 913–922, 2008.

[23] Jiefeng Cheng, Jeffrey Xu Yu, and Philip S. Yu. Graph Pattern Matching: A Join/Semijoin Approach. *IEEE Trans. on Knowl. and Data Eng.*, 23(7):1006–1021, July 2011.

[24] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. Top-k graph pattern matching over large graphs. In *ICDE*, pages 1033–1044, 2013.

[25] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Statistics and Social Network of YouTube Videos. In *IWQoS*, pages 229–238, 2008.

[26] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Who is tweeting on Twitter: human, bot, or cyborg? In *ACSAC*, pages 21–30, 2010.

[27] CiteSeerX. Public Dataset, 2011. http://csxstatic.ist.psu.edu/about/data.

[28] 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.

[29] John P Dickerson, Vadim Kagan, and VS Subrahmanian. Using sentiment to detect bots on Twitter: Are humans more opinionated than bots? In *ASONAM*, pages 620–627, 2014.

[30] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified Top-k Graph Pattern Matching. *PVLDB*, 6(13):1510–1521, 2013.

[31] V. Kumar G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[32] Rosalba Giugno and Dennis Shasha. GraphGrep: A Fast and Universal Method for Querying Graphs. In *ICPR Conf.*, pages 112–115, 2002.

[33] Sudipto Guha and Samir Khuller. Improved Methods for Approximating Node Weighted Steiner Trees and Connected Dominating Sets. *Inf. Comput.*, 150(1):57–74, 1999.

[34] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query.

[35] Bruce Hendrickson and Robert Leland. A Multi-Level Algorithm For Partitioning Graphs. *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 28, 1995.

[36] M. Huisman and M. A.J Van Duijn. Software for social network analysis. *Models and methods in social network analysis*, pages 270–316, 2005.

[37] IMDb. IMDb Dataset, 2012. http://www.imdb.com/interfaces.

[38] Yiping Ke, James Cheng, and Jeffrey Xu Yu. Querying Large Graph Databases. In *DASFAA Conf.*, pages 487–488, 2010.

[39] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - A pragmatic semantic repository for OWL. In *WISE Workshops*, pages 182–192. Springer Berlin Heidelberg, 2005.

[40] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. Efficient Execution of Top-K SPARQL Queries. In *ISWC*, pages 344–360, 2012.

[41] Hooran MahmoudiNasab and Sherif Sakr. An Experimental Evaluation of Relational RDF Storage and Querying Techniques. In *DASFAA Works.*, pages 215–226, 2010.

[42] Mauro San Martín and Claudio Gutierrez. Representing, Querying and Transforming Social Networks with RDF/SPARQL. In *ESWC Conf.*, pages 293–307, 2009.

[43] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM*, pages 29–42, 2007.

[44] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In *ISWC*, pages 454–469, 2011.

[45] Richard Myers, Richard C. Wilson, and Edwin R. Hancock. Bayesian Graph Edit Distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(6):628–635, 2000.

[46] 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.

[47] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, August 2008.

[48] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD Conf.*, pages 627–640, 2009.

[49] N.J. Nilsson. *Principles of artificial intelligence*. Tioga, Palo Alto CA, 1980.

[50] W. Nooy, A. Mrvar, and V. Batagelj. Exploratory social network analysis with Pajek. *Structural analysis in the social sciences*, 27, 2005.

[51] Ontotext. GraphDB, 2014. http://www.ontotext.com.

[52] Michael Ovelgönne, Noseong Park, V. S. Subrahmanian, Elizabeth K. Bowman, and Kirk Ogaard. Personalized Best Answer Computation in Graph Databases. In *International Semantic Web Conference*, volume 8218 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2013.

[53] Yan Qi, K. Selçuk Candan, and Maria Luisa Sapino. Sum-Max Monotonic Ranked Joins for Evaluating Top-K Twig Queries on Weighted Data Graphs. In *VLDB*, pages 507–518, 2007.

[54] Sherif Sakr. GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries. In *DASFAA Conf.*, pages 123–137, 2009.

[55] Sherif Sakr and Ghazi Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Record*, 38(4):23–28, 2009.

[56] 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.

[57] Michael Sintek and Malte Kiesel. RDFBroker: A Signature-Based High-Performance RDF Store. In *ESWC*, pages 363–377, 2006.

[58] 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, Beijing, China, 2008. ACM.

[59] V. S. Subrahmanian and Diego Reforgiato Recupero. AVA: Adjective-Verb-Adverb Combinations for Sentiment Analysis. *IEEE Intelligent Systems*, 23(4):43–50, 2008.

[60] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[61] Yuanyuan Tian and Jignesh M. Patel. TALE: A Tool for Approximate Large Graph Matching. In *ICDE*, pages 963–972, 2008.

[62] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Exploration Newsletter*, 5(1):59–68, 2003.

[63] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[64] R. Wilson and E. Hancock. Bayesian compatibility model for graph matching. *Pattern Recognition Letters*, 17:263–276, 1996.

[65] Xifeng Yan, Bin He, Feida Zhu, and Jiawei Han. Top-K aggregation queries over large networks. In *ICDE*, pages 377–380, 2010.

[66] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure Similarity Search in Graph Databases. In *SIGMOD Conf.*, pages 766–777, 2005.

[67] Kyung Hyan Yoo and Ulrike Gretzel. Comparison of Deceptive and Truthful Travel Reviews. In *ENTER*, pages 37–47. Springer, 2009.

[68] Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT Conf.*, pages 192–203, 2009.

[69] Shijie Zhang, Shirong Li, and Jiong Yang. SUMMA: subgraph matching in massive graphs. In *CIKM Conf.*, pages 1285–1288, 2010.

[70] 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.

[71] Yuanyuan Zhu, Lu Qin, Jeffrey Xu Yu, and Hong Cheng. Finding top-k similar graphs in graph databases. In *EDBT*, pages 456–467, 2012.

[72] Lei Zou, Lei Chen, and M. Tamer Özsu. DistanceJoin: Pattern Match Query In a Large Graph Database. *PVLDB*, 2(1):886–897, 2009.