UNIVERSITY OF CALIFORNIA
Santa Barbara

Querying Large-scale Knowledge Graphs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Shengqi Yang

Committee in Charge:

Professor Xifeng Yan, Chair

Professor Divyakant Agrawal

Professor Tao Yang

March 2015

The Dissertation of
Shengqi Yang is approved:

Professor Divyakant Agrawal

Professor Tao Yang

Professor Xifeng Yan, Committee Chairperson

March 2015

To my mother, Shuhua Yang,

and to my wife, He Bu,

for all of the sacrifices you've made on me

Acknowledgements

My deepest acknowledge goes to my Ph.D. advisor, Professor Xifeng Yan, for his serious supervision and guidance. His invaluable advice, insightful vision, and constructive criticism have always promoted me to conduct the cutting edge projects in this fast growing era. Without the countless discussion and talk with him, I could not be able to establish the frontier works that could have the impact on the development of this research area. Meanwhile, Professor Yan also demonstrates great enthusiasm for educating us toward a meaningful lifetime career.

I would also like to thank my committee members, Professor Divyakant Agrawal, and Professor Tao Yang, for serving on my committee and providing many insightful suggestions on my research.

My cordial thanks to my collaborators: Professor Yinghui Wu, Dr. Mudhakar Srivatsa, Dr. Arun Iyengar, Dr. Arijit Khan, Dr. Jianguang Lou, Dr. Qiang Fu, Bo Zong, Huan Sun, Yanan Xie, and Tianyu Wu. They always inspired me and played the crucial role in every piece of my work. I also want to thank my friends, who have been supporting me to go through many difficulties in my research work and the Ph.D. life.

It is a great fortune for me to attend several internships during my study. This gives me many chances to interact with other people, discover the industry and review my research. I would like to thank all the mentors, Mudhakar Srivatsa, Arun K. Iyengar and Kedar Bellare, who have seriously supervised me during and after the internships.

Curriculum Vitæ

Shengqi Yang

Education

2010-2015	University of California, Santa Barbara, USA
	Ph.D., Computer Science
	Advisor: Prof. Xifeng Yan
	Research: Graph Search and Graph Management
	Thesis: Querying Large-scale Knowledge Graphs
2007-2010	Beijing University of Post and Telecoms, China
	M.S., Computer Science
2003-2007	Beijing University of Post and Telecoms, China
	B.S., Computer Science

Experience

2010 – 2015	Research Assistant, University of California, Santa Barbara.
	Advisor: Prof. Xifeng Yan
	Project: Graph Search and Graph Management.
Summer 2014	Student Intern, Facebook Inc., Menlo Park, CA.
	Advisor: Dr. Kedar Bellare

Project: Large-scale Entity Deduplication.

Summer 2012	Student Intern, IBM T.J. Watson Research Center, NY.
	Advisor: Dr. M. Srivatsa and Dr. A. Iyengar
	Project: Keyword Search on Large Graphs.
Summer 2011	Student Intern, IBM T.J. Watson Research Center, NY.
	Advisor: Dr. M. Srivatsa and Dr. A. Iyengar
	Project: Distributed Graph Search.

Selected Publications

Z. Guan, **S. Yang**, H. Sun, M. Srivatsa and X. Yan. Fine-Grained Knowledge Sharing in Collaborative Environments. In TKDE, 2015.

S. Yang, Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu and X. Yan. SLQ: A User-friendly Graph Querying System. In *SIGMOD*, 2014 (Demo).

S. Yang, Y. Wu, H. Sun and X. Yan. Schemaless and Structureless Graph Querying. In *VLDB*, 2014

Y. Wu, **S. Yang**, M. Srivatsa, A. Iyengar and X. Yan. Summarizing Answer Graphs Induced by Keyword Queries. In *VLDB*, 2014.

Y. Wu, **S. Yang** and X. Yan. Ontology-based Subgraph Querying. In *ICDE*, 2013 (Best Poster Award).

Y. Li, P. Kamousi, F. Han, **S. Yang**, X. Yan and S. Suri. Memory Efficient Minimum Substring Partitioning. In *VLDB*, 2013.

S. Yang, X. Yan, B. Zong and A. Khan. Towards Effective Partition Management for Large Graphs. In *SIGMOD*, 2012.

Honors and Awards

ICDE Best Poster Award, 2013

SIGMOD Travel Award, 2012

Samsung Scholarship, 2009

National Scholarship (China), 2006

National Scholarship (China), 2004

Abstract

Querying Large-scale Knowledge Graphs

Shengqi Yang

With the rise of the Internet, social computing and numerous mobile applications have brought about a large amount of unstructured entity data, including places, events, and things. On one hand, the entity data, populated by a variety of data sources, is growing at an ever increasing speed. On the other hand, the service suppliers expect to render the entities pertinent to the things present in the information need of the users, rather than the data that merely matches the request strings. To meet these challenges, knowledge graph has been widely adopted in practice and become a fundamental building block for many commercial products from better recommendation systems to enhanced search engines.

As the knowledge graph subsumes humongous valuable content, there is an emerging need for the querying techniques that can extract and deliver the information from the data to the users. However, querying the real-life knowledge graph is not a trivial task. It has to deal with complex, unstructured graph data that can't fit a specific data model. Although there are many research efforts that aim in this direction, they typically have not addressed the real challenges: (1) In contrast to the complex and tedious graph data, the query from the end users tends to be ambiguous and underspecified. (2) It requires semantic query understanding since the rigid string matching is often not desirable. (3) The querying system must scale to large heterogeneous graph data.

In this thesis, we propose to tackle the challenges by primarily introducing a novel framework, Schemaless Graph Querying (SLQ), which supplies a flexible mechanism to querying large knowledge graphs. In essence, the query engine is built upon a set of transformation functions that automatically map keywords and linkages from a query to their matches in a graph. The end users are therefore not required to describe their queries precisely as that by most querying systems. To return the most relevant results in a fast way, SLQ also incorporates a learned ranking model that shall not rely on manually labeled data especially when the training data is difficult to obtain. This in turn gives rise to efficient top-K search techniques.

SLQ is designed inherently to better understand the user query intent. Rather than merely performing syntax based search, SLQ introduces the ontology base matching technique that can as well produce semantically related results. By leveraging an ontology index, an effective filtering mechanism is proposed to fast extract the top-K results based on the similarity quality. Moreover, a result summarization technique is invented to help the user inspect the excessive number of results: The users can enlarge and get detailed information on a summarized result view based on their search intents.

Another key problem in the context of graph querying is that this process needs to run under severe time constraints. The efficient search techniques are introduced from both the algorithm and the system aspects. In terms of the algorithm, we propose STAR, a framework for fast top-K searching for star queries. The technique is further extended to tackle general graph queries. From the system aspect, we investigate the distributed processing of large graphs in a cluster and propose the graph partitioning approaches. The technique is able to adapt in real time to changes in the query workload.

In summary, my research is introduced based on the hypotheses that as the knowledge graph becomes more available, there is a great chance of serving user queries in a better way. My contributions devote to realize this vision mainly for the applications of graph querying.

Professor Xifeng Yan
Dissertation Committee Chair

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Graph querying is widely adopted to retrieve information from emerging graph databases, *e.g.,* knowledge graphs, information and social networks. Given a query, it is to find reasonable top answers (*i.e.,* matches for the query) from a data graph. Searching real-life graphs, nevertheless, is not an easy task especially for non-professional users. (1) Either no standard schema is available or schemas become too complicated for users to completely possess. (2) Graph queries are hard to write and interpret. Structured queries (*e.g.,* XQuery [23] and SPARQL [66, 117]) require the expertise in complex grammars while keyword queries [119, 152] can be too ambiguous to reflect user search intent. Moreover, most of these methods adopt predefined ranking model [66, 119], which is barely able to bridge the gap between the queries and the true matches. (3) Moreover, it is a daunting task for users to inspect a large number of matches produced from querying large-scale heterogeneous graph data.

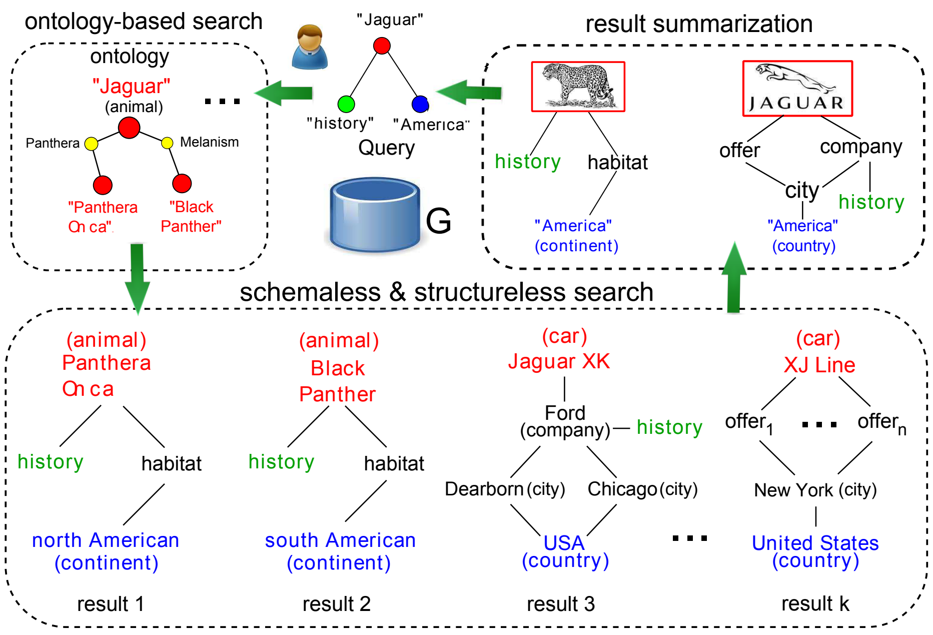


Figure 1.1: Searching a knowledge graph.

Example 1: Consider a query asking "tell me about the history of Jaguar in America." The query can be presented as either a keyword query "Jaguar history America," or a small graph in Figure 1.1. To find answers for such a simple query is, nevertheless, not easy. There are several challenges problems: (1) *How to find matches that are semantically related to the query?* A keyword *e.g.,* "Jaguar" may not have identical matches, but instead can be matched with entities that are semantically close, *i.e.,* luxury cars or animals. (2) *Which answer is better?* A large number of possible answers can be identified by various matching mechanisms. For example, "Panthera Onca" is closely related with "Jaguar" as its scientific name, while "Jaguar XK" is another match obtained simply by string transformations. A ranking model should be employed and tuned with or without manual tuning effort. (3) *How to deliver the good answers to the users in query time?* Since the end users are usually impatient, it is critical to quickly identify the best among a large number of answers. (4) *How to help users better understand results without inspecting them one by one?*. For example, different species related to Jaguar (result 1 and result 2), or various car prototypes (result 3 to result k). This kind of complexity contrasts to the impatience of web users who are only interested in finding good answers in a short time. □

To answer these questions, we propose SLQ, a novel graph querying system for schemaless graph querying [160]. (1) To better understand search intent, SLQ interprets queries with external ontology to find semantically close matches. (2) It automatically supports multiple mapping functions, namely, *transformations*, *e.g.,* synonym, abbreviation, ontology, to identify reasonable answer via learning to rank, and works with both (a) a cold-start strategy that requires no manual effort for system tuning, and (b) a warm-start strategy to adjust the ranking model with available user feedback and query logs. (3) A fast top-k search strategy is integrated with SLQ. It can fast retrieve the best matches of a graph, without relying on precomputed index and match scores. (4) To help users better understand results and refine their queries, it supports concise result summarization.

Users may inspect small summaries, and then decide to (a) drill-down for details, or (b) interactively refine queries with interesting summaries. (5) In response to the widely adopted distributed solutions and the large-scale graph data, SLQ also incorporates a distributed realization of graph querying processing by leveraging novel and effective graph partitioning techniques.

In addition, SLQ supports a wide range of queries. Users may issue (1) a keyword query as a set of keywords, where each keyword describes an entity; or (2) a (not necessarily connected) graph query, where each query node has a set of labels as conditions, and an edge between two nodes, if any, specifies the relation constraints posed on two query nodes by users. We demonstrate SLQ over three major knowledge graphs, *i.e.,* DBpedia, YAGO2 and Freebase as described in the table below. A single such knowledge graph could have more than 10K types of entities, making it difficult for users to fully grasp.

Table 1.1: Real-life knowledge graphs.

Graphs	entities	relations	node types	relation types
DBpedia [1]	3.7M	20M	359	800
YAGO2 [66]	2.9M	11M	6,543	349
Freebase [2]	40.3M	180M	10,110	9,101

To the best of our knowledge, the innovations introduced in SLQ are not seen before in any previous graph querying systems (*e.g.,* [23, 66, 107, 117, 119]). Actually, SLQ system is among the first efforts to develop a unified framework for querying large-scale complex graph data. Our design is to help the user access the graphs in a much easier and powerful manner. It is capable of finding high-quality answers when prior structured query languages do not work.

Before formally presenting the techniques, we summarize our key contributions in SLQ as follows.

1.1 Schemaless Graph Querying

In the core of SLQ is the schemaless querying approach, which introduces a principled way to match the queries from the users to the data entities in the knowledge graph. It consists of two main components, transformation based matching and a ranking model that can be learned in both offline and online manner. As remarked earlier, SLQ does not require a user to describe a query precisely as required by most search applications. Hence to render this high flexibility to the users, we propose a mechanism of *transformation*: given a query, the query evaluation is conducted by checking if its matches in a graph database can be reasonably "transformed" from the query through a set of transformation functions.

Matching Quality Measurement. To measure the quality of the matches induced by various transformations, SLQ adopts a ranking function of weighted transformations. The function incorporates two types of score functions: node matching and edge matching score function. Each score function aggregates the contribution of all the possible transformations with corresponding weights. More specifically, by harnessing the probabilistic graphical model, we define the ranking function with *Conditional Random Fields* [136], since it satisfies two key considerations: using training data, its formulation could optimize the weights of the transformations for good ranking quality; the inference on the model provides a mechanism to search top-k matches quickly.

Ranking model learning. A key issue is how to select a ranking function by determining reasonable weights for the transformations. Instead of assigning equal weights or calibrating the weights by human effort, SLQ automatically learns the weights from a set of training instances. Each instance is a pair of a query and one of its relevant (labeled as "good") answers. The learning aims to identify for each transformation a weight, such that if applied, the model ranks the good answers as high as possible for each instance.

SLQ begins with a cold-start stage where no manual effort is required for instances labeling, and can be "self-trained" in the warm-start stage, using user feedback and query logs. When no user query log is available, SLQ randomly extracts a set of subgraphs from the data graph as "query templates." It then injects a few transformations to the template and generates a set of training queries. Intuitively, each training query should have the corresponding templates as its “good matches,” since the query can be transformed back to the template.

Query Processing. SLQ efficiently finds top matches, leveraging *approximate inferencing* [136]. It treats a query as a graphic model and the matches as assignment to its random variables (query nodes). By propagating messages among the nodes in the graphical model, the inference can identify top assignments (matches) that maximize the joint probability (ranking score) for the model. Together with a graph sketch data structure, this technique dramatically reduces the query processing time, with only small loss of match quality (less than 1% in our validation).

We introduce schemaless graph querying in more details in **Chapter 2**.

1.2 Fast Top-k Graph Querying

Since the queries could be quite ambiguous, the search may result in an excessive number of answers. The top-k graph querying intends to quickly identify the answers of high-quality *w.r.t.* a ranking function. Efficient top-k querying is a well-studied problem in relational databases [70], XML [55, 121], and RDFs [144]. In contrast to these class techniques, knowledge graph querying has its unique properties and comes with new challenges and opportunities: (1) The matching score that combines multiple similarity measures has to be calculated and ranked online, (2) query answers are often inexact, and (3) query graphs are usually small. We show that a direct application of prior approaches in this new setting does not work well.

In SLQ, we propose STAR, a framework to exploit fast top-k search for star queries and efficiently assemble them to answer general graph queries. STAR has two components: A fast top-k algorithm for single star-shaped queries and an assembling algorithm for general graph queries. The assembling algorithm uses fast star querying as building blocks and iteratively sweeps the star match lists with dynamically adjusted bound. When the size of Q and k is bounded by a constant, we show that the time complexity of answering single star queries is linear to $|E|$, the edge number of G. For approximate graph matching where an edge can be matched to a path with bounded length d, we extend the algorithm using message passing and achieve time complexity $O(d^2|E|+m^d)$, where m is the maximum node degree in G. Our algorithm does not require any pre-built indices. Using three real-life knowledge graphs, we experimentally verify that STAR is 5-10 times faster than the state-of-the-art TA-style subgraph matching algorithm, and 10-100 times faster than a graph search algorithm based on belief propagation.

We discuss STAR framework in details in **Chapter 3**.

1.3 Ontology-based Graph Querying

In contrast to conventional graph querying systems, SLQ does not limit itself with single, fixed search semantic. Instead, it employs a set of matching functions that are more capable to find good matches in heterogeneous graphs [160]. We start by demonstrating ontology-based search, as an example for various matching mechanisms integrated in SLQ.

One of the challenges is to find the semantically related matches. SLQ leverages ontology-based search [157] to bridge the entities from queries and data graphs via a set of ontology closeness metrics. Given external ontology graphs (*e.g.,* DBPedia Ontology [1]), SLQ finds the semantically related entities (specified by a closeness measure) in the ontology graphs for each entity (keyword) in a query [157]. A straightforward "substituting-and-querying" method may next interpret the query by substituting the keywords with their related entities, which in turn have matches in the data graph. Top matches can then be extracted by processing each new query. However, it may yield a tremendous number of queries. Instead, SLQ leverages an effective ontology index [157]. In a nutshell, it computes several sketches of the data graph using the ontology graph. Each sketch is induced by grouping the nodes that are semantically close. Upon receiving a query, SLQ can efficiently identify the top matches by querying on these small sketches only.

Example 2: While there are no similarly labeled entities for "Jaguar" in the data graph (Figure 1.1), SLQ checks an ontology graph, and identifies its two semantically related matches (as a type of animal): "Panthera Onca," its scientific name and "Black Panther," its melanistic color variant. These can hardly be found by conventional IR metrics or string similarity, or by using the query and data graph alone. □

SLQ wraps ontology-based searching with (a) an ontology transformation function that maps a keyword to a set of valid entities, and (b) an ontology index. These are seamlessly integrated in the query processing of SLQ. We formally introduce this component in **Chapter 4**.

1.4 Result Summarization

Due to the sheer volume of data, graph querying usually generates a lot of results that are too many to inspect. This not only makes the understanding of the results a daunting task, but also frustrates the users to continue refining the search. The result summarization feature of SLQ addresses the two challenges in a "two-birds-with-one-stone" way by leveraging [156]. (1) Given a query and a set of matches, SLQ effectively describes all the matches (as graphs) with a few *summary graphs*. A summary graph preserves the connectivity of the keyword pairs. By reviewing the summary graphs, users easily get intuitive "big pictures" of all the matches. (2) The summary graphs can further be used to refine the search by suggesting new query nodes or edges, or be issued directly as new queries.

Example 3: SLQ provides two intuitive summaries for the matches in Figure 1.1. The first summary indicates that "Jaguar" refers to animals living in America continents with evolution history. The second shows that it refers to a certain type of cars sold by dealers and companies in major cities of USA, with the company history in car industry. In addition, new nodes or keywords, *e.g.,* "offer," are suggested to users to inspire queries with new interests. □

We introduce the summarization technique in more details in **Chapter 5**.

1.5 Distributed Graph Query Processing

Knowledge graphs are often massive with millions, even billions of vertices, making common graph operations computationally intensive. In the presence of properties associated with nodes/edges, it is clear that graph data can easily scale up to terabytes in size. The recent *Linked Open Data* project has published more than 20 billion RDF triples [61]. Although the RDF data is generally represented in triples, the data inherently presents graph structure and is therefore interlinked. Not surprisingly, the scale and the flexibility rise to the major challenges to the knowledge graph management. Fortunately, as large-scale the commodity clusters become available and affordable, various successful distributed platforms and solutions [53, 54, 94, 99, 124, 129, 162] are emerging as a critical avenue for data intensive computing on massive graphs.

In SLQ, we introduce a Self Evolving Distributed Graph Management Environment (SEDGE), to minimize inter-machine communication during graph query processing in a cluster environment. In order to improve query response time and throughput, SEDGE introduces a two-level partition management architecture with complimentary primary partitions and dynamic secondary partitions. These two kinds of partitions are able to adapt in real time to changes in query workload. Sedge also includes a set of workload analyzing algorithms whose time complexity is linear or sublinear to graph size.

The details of SEDGE is discussed in **Chapter 6**.

1.6 Summary

With the ever-increasing volume and complexity of the real-world knowledge graph, there is an urgent need to make use of the data effectively and efficiently. Among the various applications, intelligent search has been emerging as a critical approach in today's major platforms, such as Apple's Siri, Google's Knowledge

Graph, Microsoft's Cortana, and Facebook's Graph Search. In response to these applications, however, the success of the traditional search techniques over traditional data, such as relational database, XML, and RDFs, cannot be restored to support intelligent search since 1) the data is no long well structured so that a good data schema can be leveraged in the search. In practice, the data is unstructured and usually can be represented as a general graph with rich attributes. Search over graphs of such kind is more complicated because even a simple keyword search in graphs poses an NP-hard problem, *i.e.,*, Steiner Tree problem. 2) the queries from the users are usually quite ambiguous and not well formulated. Although the query models, like SQL, SPARQL, and XQuery, are expressive, they require expert knowledge. This is prohibitive for non-professional users. 3) By leveraging more information, such as the graph structure, node/edge content and the semantic relation (*e.g.,* Ontology), the search system should tend to capture the latent meaning of the user's query and return the most satisfying results in online manner.

In summary, we propose SLQ, which aims at integrating innovative graph querying techniques that allow the users to easily acquire their information need from complex knowledge graph. We endeavor to design efficient algorithms that can process user's query in online manner, while without sacrificing any generality in practice. SLQ is a real system that can not only demonstrate the usability of the proposed techniques, but also suggest great opportunities for leveraging knowledge graphs in many real applications.

The dissertation is organized as follows. Chapter 2 introduces the schemaless graph querying framework, which lies at the core of SLQ. Chapter 3 presents the fast top-k graph querying methods. Chapter 4 discusses the semantic matching when ontology information is available. Chapter 5 introduces the summarization techniques on query results. We also present the distributed query processing in Chapter 6 and the system architecture of SLQ in Chapter 7. We conclude the work and outline the future directions in Chapter 8.

Chapter 2

Schemaless Graph Querying

Querying complex graph databases such as knowledge graphs is a challenging task for non-professional users. Due to their complex schemas and variational information descriptions, it becomes very hard for users to formulate a query that can be properly processed by the existing systems. We argue that for a user-friendly graph query engine, it must support various kinds of transformations such as synonym, abbreviation, and ontology. Furthermore, the derived query results must be ranked in a principled manner.

In this chapter, we introduce a novel framework enabling schemaless graph querying, where a user need not describe queries precisely as required by most databases. The query engine is built on a set of transformation functions that automatically map keywords and linkages from a query to their matches in a graph. It automatically learns an effective ranking model, without assuming manually labeled training examples, and can efficiently return top ranked matches using graph sketch and belief propagation. The architecture of SLQ is elastic for "plug-in" new transformation functions and query logs. Our experimental results show that this new graph querying paradigm is promising: It identifies high-quality matches for both keyword and graph queries over real-life knowledge graphs, and outperforms existing methods significantly in terms of effectiveness and efficiency.

2.1 Introduction

Graph querying is widely adopted to retrieve information from emerging graph databases, *e.g.,* knowledge graphs, information and social networks. Searching these real-life graphs is not an easy task especially for non-professional users: either no standard schema is available, or schemas become too complicated for users to completely possess. For example, a single knowledge graph could have more than 10K types of entities, as illustrated in Table 1.1, not to mention the different presentations of entity attributes.

This kind of complexity contrasts to the impatience of web users who are only interested in finding query answers in a short period. The existing structured query techniques such as XQuery [23] and SPARQL [117] are barely able to address such challenge. There is a usability issue arising from query preparation. Keyword queries (*e.g.,* [75,86,152]) were proposed to shield non-professional users from digesting complex schemas and data definitions. Unfortunately, most of keyword query methods only support a predefined similarity measure, such as approximate string matching [96] and ontology-based matching [157]. A general, systematic approach that automatically supports multiple measures (*e.g.*, synonym, abbreviation, ontology, and several more summarized in Table 2.1) all together is lacking.

In this chapter, we present a principle that could take multiple matchings into account. Under this principle, given a query Q, query evaluation is conducted by checking if its matches in a graph database G can be "transformed" from Q through a set of transformation functions.

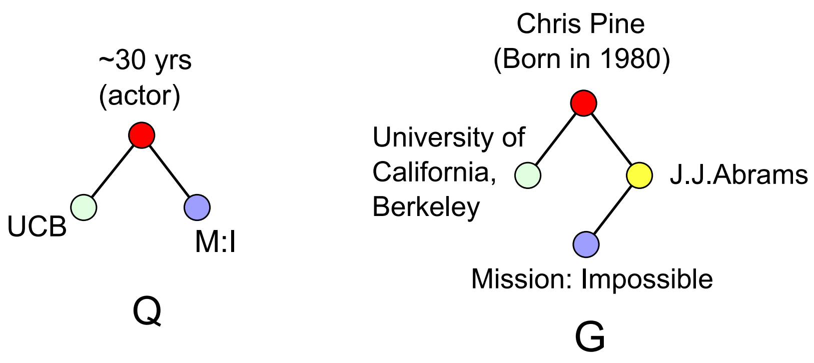


Figure 2.1: Searching with transformations.

Example 4: To find a movie star in a knowledge graph, a graph query Q is issued (Figure 2.1), which aims to find an actor whose age is around 30 ("30

Table 2.1: Transformations in SLQ.

Transformation	Category	Example
First or Last token	String	'Anne Hathaway' → 'Anne', 'Justin Bieber' → 'Bieber'
Abbreviation	String	'Jeffrey Jacob Abrams' → 'J.J. Abrams'
Drop	String	'US Airways Company' → 'US Airways'
Bag of words	String	'Yankees hat' → 'Tom Cruise' ('. . . signs Yankees hat')
Prefix	String	'Street' → 'Str'
Acronym	String	'International Business Machines' → 'IBM'
Synonym	Semantic	'lawyer' → 'attorney'
Ontology	Semantic	'teacher' → 'educator'
Date Gap	Numeric	'2010' → '3 yrs ago' (as of, *e.g.,* 2013)
Range	Numeric	'∼30 yrs' → '33 yrs'
Unit Conversion	Numeric	'0 Celsius' → '32 Fahrenheit', '3 mi'→ '4.8 km'
Distance	Topology	'Pine'-'M:I' → 'Pine'-'J.J.Abrams'-'M:I'

`yrs`"), graduated from UC Berkely ("`UCB`"), and may relate to movie "Mission: Impossible" ("`M:I`"). One may identify a match for Q as shown in Figure 2.1. The match indicates that "`30 yrs`" in Q refers to an actor "Chris Pine" who was born in 1980, "`UCB`" is matched to the University of California, Berkeley, and "`M:I`" refers to the movie "Mission:Impossible." Traditional keyword searching based on IR methods or string similarity cannot identify such matches. □

Given a few transformation functions, one might find many matches of Q in a graph database. A transformation-friendly query engine must address the following two questions: (1) *how to determine which match is better*? (2) *how to efficiently identify the top ranked matches*?

Intuitively, the selectivity, the popularity, and the complexity of transformation functions shall be considered and used as a ranking metric for these matches. How to choose, from many possible transformations, an appropriate ranking metric that leads to good matches? First, a searching algorithm should be deployed to determine the best transformation for different portions of a query. For example, "UCB" should be automatically transformed to entities using it as acronym, rather than string edit distance. This requires a weighting function for various transformations. Second, to identify such a function, manual tuning should be reduced to a minimum level. Instead of asking users to tune the weights, learning to rank [87, 139] is more appropriate. Unfortunately, it usually needs manually labeled training data, again a daunting task for end users. Finally, since there could be too many matches to inspect, it is important to only return top-k results. While desirable, this top-k search problem is much more challenging due to the presence of different transformations, compared to its single transformation counterpart.

Contributions. This work proposes a first-kind of graph querying framework that answers all these questions.

(1) We propose a new, generalized graph searching problem: Given a query Q, a graph G and a library of transformation functions $\mathcal{L}$, where there are multiple matches in G that can be transformed from Q by applying $\mathcal{L}$, it is to find the top-k ranked matches for Q. In contrast to traditional graph searching using single, predefined similarity metric such as string similarity, we use a metric combining transformations of various kinds. The metric itself is automatically learned.

(2) We propose SLQ, a general graph query framework for $\underline{s}$chema$\underline{l}$ess $\underline{q}$uerying. It consists of two phases: *offline learning* and *online query processing.* (a) *Given multiple matches transformed from Q, how to decide a proper ranking metric?* Certainly a manually picked combination function, *e.g.,* averaging, is not going to work elegantly. We show that this problem can be solved by a parameterized ranking model. We adopt conditional random fields [136], as it not only gives a good ranking model, but also indicates a fast matching search algorithm. In the offline learning phase, the framework needs to solve the cold-start problem, *i.e.*,, where to find training samples to train the model. Manually labeled matches might be too costly for a few sample queries. A systematic approach is hence introduced to create sample queries and answers by extracting subgraph queries from G, inject transformations to these queries, and form query-answer pairs for training. (b) *Given a ranking metric, how to efficiently find top ranked matches?* For general graph queries and keyword queries, we prove that the problem is

NP-hard. We propose a polynomial time heuristic top-k algorithm for online query processing. The problem is tractable for tree-structured queries, and an exact, polynomial time algorithm is developed. Both algorithms stop once k best matches are identified, without inspecting every match. In practice, they run very fast.

(3) Using several real-life data/knowledge graphs, we experimentally verify the performance of our graph querying engine. It outperforms traditional keyword (Spark [96]) and approximate graph searching (NeMa [79]) algorithms in terms of quality and efficiency. For example, it is able to find matches that cannot be identified by the existing keyword or graph query methods. It is 2-4 times faster than NeMa, and is orders of magnitude faster than a naive top-k algorithm that inspects every match.

To the best of our knowledge, these results are among the first efforts of developing a unified framework for schemaless and structureless querying. SLQ is designed to help non-professional users access complex graph databases in a much easier manner. It is a flexible framework capable of finding good matches when structured query languages do not work. New transformations and ranking metrics can be *plugged in* to this framework easily. The proposed framework can also be extended to query relational data, where a similar usability problem exists. The contribution of this study is not only at providing a novel graph querying paradigm, but also at the demonstration of unifying learning and searching for much more intelligent query processing. The proposed techniques can be adapted easily to a wide range of search applications in databases, documents and the Web.

2.2 Preliminary

Property graph model. We adopt a *property graph model* [122]. A graph $G = (V, E)$ is a labeled graph with node set V and edge set E, where each node $v \in V$ has a property list consisting of multiple attribute-value pairs, and each edge $e \in E$ represents a relationship between two entities. The model is widely adopted to present real-life schemaless graphs. To simplify our presentation, we will first treat all the information associated with nodes and edges as keywords, and then differentiate type and value in Section 2.7.

Queries. We formulate a query Q as a property graph (V_Q, E_Q). Each query node in Q describes an entity, and an edge between two nodes, if any, specifies the connectivity constraint posed on two query nodes. Q could be disconnected when a user is not sure about a specific connection. This query definition covers both keyword query [152] (query nodes only) and a graph pattern query [28] (connected query graph). For the ease of discussion, we first focus on the query that is connected. How to handle disconnected queries including keyword queries is given in Section 2.7.

Traditional graph querying assumes structured queries formulated from well-defined syntax and vocabulary (*e.g.,* XPath and SPARQL). We consider general queries that might not exactly follow the structure and semantic specifications coded in a graph database.

Transformations and matches. To characterize the matches of Q, we assume a library $\mathcal{L}$ of transformation functions (or simply transformations). A transformation f can be defined on the attributes and values of both nodes and edges of Q. The transformation functions can be specified in various forms, *e.g.,* (string) transformation rules [8]. Table 2.1 summarizes several common transformations. These transformations consider string transformation, semantic transformation, numeric transformation, and topological transformation (as edge transformations). For example, "Synonym" allows a node with label "`Tumor`" to be mapped to the node "`Neoplasm`." All these transformations are supported in our implementation. New transformations, such as string similarity (*e.g.,* spelling error) [95] and Jaccard distance on word sets [79] can be readily plugged into $\mathcal{L}$. Our focus is to show a design combining different transformations, not to optimize a specific transformation.

A node or edge in Q matches its counterparts in a data graph G with a set of transformed attributes/values, specified by a matching (function) ϕ. A match of

Q, denoted as $\phi(Q)$, is a connected subgraph of G induced by the node and edge matches. For each attribute/value, we only consider one-time transformation, as the chance for transforming multiple times is significantly lower.

2.3 Schemaless Graph Querying

In this section, we provide an overview of SLQ, and its three key components: matching quality measurement, offline learning, and online query processing.

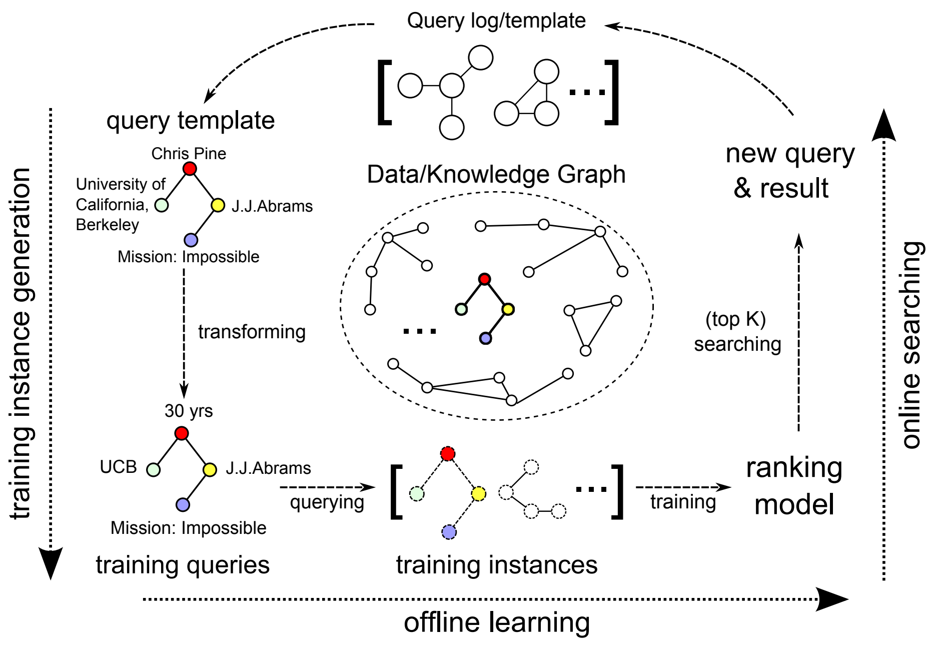


Figure 2.2: Schemaless graph querying framework.

Matching quality measurement. Given Q and a matching ϕ of Q, we need to measure the quality of $\phi(Q)$ by aggregating the matching quality of corresponding nodes and edges. Intuitively, an identical match should always be ranked highest; otherwise, $\phi(Q)$ shall be determined by the transformations, as well as their *weights* to indicate how "important" they are in contributing to a reasonable match. One possible strategy is to assign equal weight to all transformations. Certainly, it is not the best solution. For example, given a single node query, "Chris Pine," nodes with "C. Pine" (Abbreviation) shall be ranked higher than nodes with "Pine" (Last token). A predefined weighting function is also not good, as it is hard to compare transformations of different kinds. We introduce a novel learning approach to figure out their weights (Section 2.4).

Offline model learning. There might exist multiple matches for Q in a graph G using different transformations. An advanced model should be parameterized and be able to adjust the weights of all possible transformations. If a historical collection of queries and user-preferred answers is available, through a machine learning process, one can automatically estimate weights so that the user-preferred answers could be ranked as high as possible.

As suggested from previous work [87], the best practice for learning a model is to employ a query log generated by real users. However, the log might not be available at the beginning. On the other hand, the system does need a set of good-quality query-answer pairs to have its weights tuned. This becomes the chicken or the egg dilemma. In Section 2.4.2, we introduce a method to automatically generate training instances from the data graph.

Online top-k searching. Once the parameters of the ranking function are estimated in the offline learning, one can process queries online. Fast query processing techniques are required to identify top ranked matches based on the ranking function. This becomes even more challenging when multiple transformations are applicable to the same query, and the answer pool becomes very large. While the problem is in general intractable, we resort to fast heuristics. The idea is to construct a small *sketch graph* by grouping the matches in terms of Q and the transformations. The algorithm first finds the matches in the sketch graph that are likely to contain the top-k answers. It then "drills down" these matches to extract more accurate matches from the original graph G. This design avoids the need of inspecting all the matches.

Putting the above components together, Figure 2.2 illustrates the pipeline of SLQ. It automatically generates training instances from data graphs and any available query log. Using the training set, it learns a ranking model by estimating proper weights for the transformations. In the online stage, it applies efficient top-k searching to find best matches for new queries. A user can provide feedback by specifying good answers in the top-k matches, which can be put back to the query log to further improve the ranking model. In the following sections, we discuss each step in detail.

2.4 Offline Learning

Given G and a library $\mathcal{L}$ of transformations, the offline learning module generates a ranking model, without resorting to human labeling efforts. In this section, we present two key components, the parameter estimation and automatic training instance generation.

2.4.1 Ranking Function

Given Q and $\phi(Q)$, a node matching cost function $F_V(v, \phi(v))$ is introduced to measure the transformation cost from a query node v to its match $\phi(v)$. It aggregates the contribution of all the possible transformations $\{f_i\}$ with corresponding weight $\{\alpha_i\}$,

$$F_V(v, \phi(v)) = \sum_i \alpha_i f_i(v, \phi(v)) \tag{2.1}$$

where each f_i returns a binary value: it returns 1 if its two inputs can be matched by the transformation, and 0 otherwise. Analogously, an edge matching cost function is defined as

$$F_E(e, \phi(e)) = \sum_i \beta_i f_i(e, \phi(e)) \tag{2.2}$$

which conveys the transformation(s) from a query edge e to its match $\phi(e)$. $\phi(e)$ can be a path in $\phi(Q)$ with the two endpoints matched with those in e. $\{f_i\}$ can be extended to support real-valued similarity functions. We instantiate our querying framework with a set of commonly used transformations, as in Table 2.1. Other user-specified transformations can also be plugged in.

We now introduce a ranking function that could combine multiple nodes and edges matches together. There are two important factors to consider. First, using training data, it shall be able to optimize parameters $\{\alpha_i\}$ and $\{\beta_i\}$ for good ranking quality. Second, the ranking function shall have a mechanism to search top-k matchings quickly. Enumerating all possible matches of a query graph and then sorting their scores is not a good mechanism. We give a *probabilistic formulation* that satisfies both requirements. The superior performance of SLQ can already be demonstrated by this formulation. We leave the search and comparison of various probabilistic models in terms of ranking quality and query response time to future work.

Given Q and a match $\phi(Q)$, we use probability $P(\phi(Q)|Q)$ as a measure to evaluate the matching quality,

$$P(\phi(Q)|Q) = \frac{1}{Z}\exp(\sum_{v\in V_Q} F_V(v,\phi(v)) + \sum_{e\in E_Q} F_E(e,\phi(e))) \tag{2.3}$$

where Z is a normalization function so that $P(\cdot) \in [0,1]$.

The ranking function $P(\phi(Q)|Q)$ can be naturally interpreted with *conditional random fields* (CRFs), a widely applied graphical model (see [136] for more details). In our formulation, the nodes and edges in each query Q are regarded as the observed nodes and structures in CRFs; the nodes and edges in each match $\phi(Q)$ to be predicted are regarded as the output variables. CRFs directly models the distribution of the output variables given the observed variables, which naturally serves as our matching quality measure.

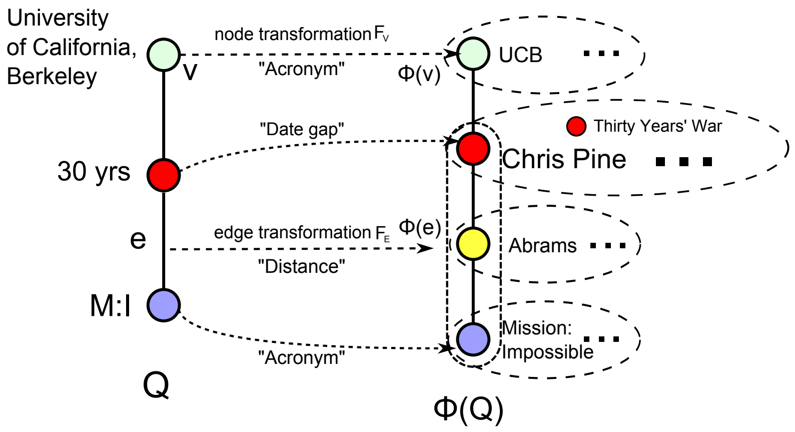


Figure 2.3: Ranking function.

Example 5: Recall the Q and a match $\phi(Q)$ (Figure 2.3). Each node, *e.g.,* `30 yrs`, may have multiple matches via multiple transformations, as remarked earlier. The quality of the match $P(\phi(Q)|Q)$ is computed by aggregating the quality of each node and edge match in $\phi(Q)$, determined by a weighted function of all transformations. □

Two key differences between SLQ and the existing graph query algorithms are (1) we support multiple transformations; and (2) the weight of these transformations are learned, rather than user-specified. The probabilistic ranking function is a vehicle to enable these two differences.

2.4.2 Transformation Weights

To determine the weights of transformations $W = \{\alpha_1, \alpha_2, ...; \beta_1, \beta_2, ...\}$, SLQ automatically learns from a set of training instances. Training instances can be regarded as past query experiences, which can teach the system how to rank the results when new queries arrive. Each training instance is a pair of a query and one of its relevant answers. Intuitively, we want to identify the parameters W that can rank relevant answers as high as possible for a given query in the training set T. We choose parameters such that the log-likelihood of relevant matches is maximized,

$$W = \arg\max_{W} \sum_{T} \log P(\phi(Q)|Q) \tag{2.4}$$

Optimizing objective functions like Eqn. 2.4 has been studied extensively in machine learning community [136]. We adopt the standard Limited-memory BFGS (L-BFGS) [89] algorithm, as it requires less memory than other approaches.

Complexity. Based on the analysis of [136], the worst time complexity of training CRFs in our problem setting is $O(N|Q||V_m|^2|T|)$, where (1) N is the number of gradient computations performed by the optimization, (2) $|Q|$ is the size of the largest query in the training set, (3) $|V_m|$ is the largest number of the matches a query node or edge may have, and (4) $|T|$ is the number of training instances. Experimental results (Table 2.3 in Section 2.8) show that its training time is affordable for large real-life graphs, as only a small sample of the graph is needed. $|V_m|^2$ is also not an issue here as one can avoid using less-selective queries.

2.4.3 Automatic Training Instance Generation

A key issue in SLQ is how to cold-start the system when no user query log is available. We developed an innovative strategy to generate artificial training instances. It turns out that this strategy works far better than just giving equal weight to all transformations. Our system first randomly extracts a set of subgraphs from the data graph and treat them as query templates. For each template $\hat{Q}$, it injects a few transformations to $\hat{Q}$ and generates a set of training queries Q. Intuitively, each training query Q should have $\hat{Q}$ as its good match, since Q can be transformed back to $\hat{Q}$. The system also identifies exact matches of Q in G. Consequently, the matches identical to Q form training instances too. The weights of transformation functions are learned by ranking $\hat{Q}$ as high as possible in the matches of Q, but below those identical matches of Q in G.

The identical matches play a key role of determining the weight of transformations. For example, with respect to a query template "Barack Obama," a match "B. Obama" is more preferred than "Obama" as there are less identical matches of "B. Obama" (*i.e.,* with higher selectivity). Therefore, by populating the training instances with random queries and results, the method can gauge the impact of transformations automatically in terms of selectivity. The second reason for this cold-start strategy to work well is that it covers different cases comprehensively, as it randomly and uniformly samples subgraphs from the data graph.

2.5 Online Query Processing

In this section, we introduce the online query processing technique that finds top-k ranked matches for Q in G with the highest scores. To simplify the discussion, we assume that each transformation f_i checks if a query node (resp. edge) matches a node (resp. path) in G in constant time.

The query processing problem is in general NP-hard, as one may verify that subgraph isomorphism [113] is its special case. To precisely compute $P(\phi(Q)|Q)$, one has to inspect every possible match, which is a daunting task. A straightforward algorithm identifies the match candidates for query node/edge via all transformations in $O(|Q||G||\mathcal{L}|)$ time, enumerates all possible result matches, and computes their rank scores to find top-k ones. Its complexity is $O(|Q||G||\mathcal{L}| + |G|^{|Q|})$, which does not scale over large G.

Observing the hardness of the exact searching (e.g., subgraph isomorphism), one shall not expect a fast solution with complete answers (except for tree queries). Instead, we resort to two heuristics. The first one leverages an inference technique in graphical models that has been verified to be efficient and accurate in practice [161] (Section 2.5.1). The second one further improves it by building a sketch of G so that low-score matches can be pruned quickly (Section 2.5.2). Our top-k algorithm based on these two techniques (Section 2.5.3) could reduce the query processing time in orders of magnitude, while only small loss of answer quality is observed (less than 1% in our experiments). Moreover, it can deliver *exact* top-k matches when Q are trees (Section 2.5.4), which is desirable as many graph queries are indeed trees.

Section 2.5.1 briefly introduces the first heuristic, LoopyBP, which needs some background knowledge to digest [153]. The readers may skip it without difficulty in understanding the remaining sections.

2.5.1 Finding Matches

The idea of LoopyBP is to treat Q as a graphical model, where each node is a random variable with a set of matches as possible assignments. It finds top assignments (matches) that maximizes the joint probability for Q (with highest matching quality). To this end, LoopyBP leverages inferencing techniques [153], which iteratively propagates "messages" among the nodes to estimate the matching quality.

Given Q, LoopyBP identifies a match $\phi(Q)$ that maximizes $P(\phi(Q)|Q)$ by seeking $\max_{u_i} b(u_i)$ [161]. For each node $v_i \in V_Q$ and its match u_i, $b(u_i)$ is formulated as:

$$b(u_i) = \max_{u_i} F_V(v_i, u_i) \Pi_{v_j \in N(v_i)} m_{ji}^{(t)}(u_i), \tag{2.5}$$

for each match u_i of v_i and each v_j in the neighborhood set $N(v_i)$ of v_i in Q. Here $m_{ji}^{(t)}(u_i)$ is a message (as a value) sent to u_i from the matches of $v_j \in N(v_i)$ at the

t^{th} iteration:

$$m_{ji}^{(t)}(u_i) = \max_{u_j} F_V(v_j, u_j) F_E((v_j, v_i), (u_j, u_i)) \cdot \prod_{v_k \in N(v_j) \setminus v_i} m_{kj}^{(t-1)}(u_j), \tag{2.6}$$

for each match u_j of v_j. (u_j, u_i) represents the match of the query edge (v_j, v_i). Intuitively, the score $b(u_i)$ is determined by the quality of u_i as a node match to v_i (F_V), the quality of edge matches, *e.g.,* (u_j, u_i), attached to u_i (F_E), and the match quality of its neighbors u_j as messages ($m_{ji}(u_i)$). Hence the node u with the maximum $b(\cdot)$ and its "surrounded" node and edge matches naturally induce a match with good quality in terms of matching probability.

Algorithm. Based on the formulation, LoopyBP finds top matches in three steps. (1) It first initializes the messages of each node $m^{(0)}(\cdot) = 1$. (2) It iteratively updates $b(\cdot)$ following message propagation until none of $b(\cdot)$ in successive iterations changed by more than a small threshold. (3) LoopyBP identifies best node matches $u = \mathsf{argmax}_u b(u)$, and then extracts top-k matches $\phi(Q)$, following a backtracking strategy [161]. More specifically, LoopyBP first selects a match u with the highest score $b(\cdot)$, and induces a top 1 match $\phi(Q)_1$ following the node matches with top $b(\cdot)$ scores connected to u. It then finds a next best match by performing two message propagations: (a) it identifies a match u' of a node v in Q with the second highest score $b(\cdot)$ among all the matches and is not in $\phi(Q)_1$, and then performs a propagation to find a top match $\phi(Q)_2$, fixing $\phi(v) = u'$; (b) it performs a second round of propagation where $\phi(v) \neq u'$, to "trace back" to an earlier state of the scores in (a), and prepare to extract a next best match. It repeats the above process until k matches are identified (see details in [161]).

Complexity. The propagation only sends messages following edge matches as paths of bounded length d constrained by edge transformation in $\mathcal{L}$. Thus each propagation traverses, for each match u of v, up to the set V_d of d hops of u in G. The algorithm takes $O(I|Q||V||V_d|)$ time for message propagation in total I iterations, where a single iteration completes when each node exchanges a message with each of its neighbors. In addition, it takes $O(|Q|)$ time to construct a best (top-1) matching ϕ for Q. Putting these together, the process of finding one match takes overall $O(I|Q||V||V_d|)$ time. To find k matches, at most $2k$ rounds of propagation are conducted with backtracking, where each round denotes the start to the convergence of a propagation. Hence it identifies top k matches in $O(k * I|Q||V||V_d|)$ time. Note that d is typically small: Edges are usually matched with short paths as observed in keyword and graph searching [79, 86].

2.5.2 Sketch Graph

With LoopyBP, one still needs to inspect a large number of node and edge matches. Observe that these matches can be naturally grouped in terms of trans-

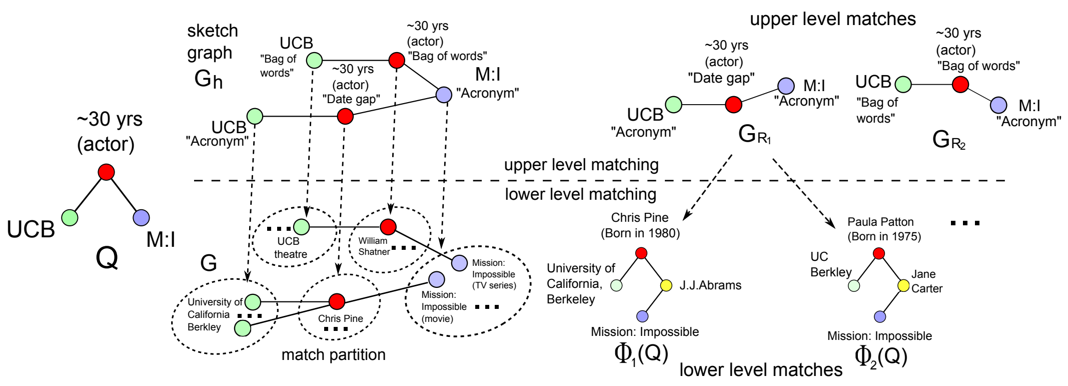


Figure 2.4: Graph sketch and top-k searching.

formations: Each match contributes the *same* matching score when it conducts the same type of transformation. Following this, we construct a *sketch graph* G_h from G induced by Q and $\mathcal{L}$. The idea is to efficiently extract matches from a much smaller G_h, and then drill down to find more accurate "lower level" ones.

We denote as σ_i the set of all matches for a node v_i in Q. (1) A *match partition* of σ_i is a set of partitions $\{\sigma_{i1}, \ldots, \sigma_{in}\}$ of σ_i, such that for any two nodes in σ_{ij}, they can be mapped to v_i via the same transformation $f_j \in \mathcal{L}$. (2) The sketch graph G_h of G contains a set of *hyper nodes*, where each hyper node $u_{(v_i,f_j)}$ denotes a match set σ_{ij} of v_i in Q induced by f_j. There is an edge connecting two hyper nodes $u_{(v_i,f_m)}$ and $u_{(v_j,f_n)}$ if and only if (v_i, v_j) is an edge of Q. Thus the match score of an edge in G_h establishes an *upper bound* of its underlying edge matches in G, since any edge in G_h is an exact match of an edge in Q. Intuitively, G_h

sketches G by grouping the matches of each query node as a single node, as long as they can match to the query node by the same transformation. Note that a sketch graph G_h can also be queried by LoopyBP. We denoted as G_R an "upper level match" from G_h, and distinguish it from a "lower level match" G_r as a subgraph of G. G_r is contained in G_R if each node of G_r is in a hyper node of G_R.

One may verify that the rank score of each upper level match G_R indicates an *upper bound* of the rank scores of all the lower level matches it contains:

Lemma 1: *For any upper level match G_R (specified by matching ϕ_R) and any lower level match G_r contained in G_R (specified by ϕ_r), $\max_{\phi_r(v_i)} b(\phi_r(v_i)) \leq \max_{\phi_R(v_i)} b(\phi_R(v_i))$, where v_i ranges over the query nodes in V_Q.* □

Proof sketch: We prove by induction on the iterations that for any $v_i \in V_Q$ at any iteration t, $m_{ji}^{(t)}(\phi_r(v_i)) \leq m_{ji}^{(t)}(\phi_R(v_i))$. (1) Let $t = 1$. Since $F_V(v_i, \phi_r(v_i)) = F_V(v_i, \phi_R(v_i))$ and $F_E(e, \phi_r(e)) \leq F_E(e, \phi_R(e))$, we have $m_{ji}^{(1)}(\phi_r(v_i)) \leq m_{ji}^{(1)}(\phi_R(v_i))$, by the definition of G_R and Eqn. 2.6. (2) Assume $m_{ji}^{(t)}(\phi_r(v_i)) \leq m_{ji}^{(t)}(\phi_R(v_i))$ for $t < n$. When $t = n$, one can verify that $m_{ji}^{n}(\phi_r(v_i))$ is no larger than $m_{ji}^{n}(\phi_R(v_i))$, again with Eqn. 2.6. Hence, by Eqn. 2.5, $b(\phi_r(v_i)) \leq b(\phi_R(v_i))$. The Lemma hence follows. □

Note that the size of G_h is independent of $|G|$: it is bounded by $O(|Q|^2|\mathcal{L}|^2)$ where $|Q|$ and $|\mathcal{L}|$ are typically small. Moreover, G_h can be efficiently constructed using indexing techniques (Section 2.6).

Example 6: A sketch graph G_h is illustrated for the query Q in Figure 2.4. A node UCB with label "Acronym" in G_h points to a group of matches via transformation "Acronym." Given Q and G_h, LoopyBP provides an upper level match G_{R_1}, which contains two lower level matches $\phi_1(Q)$ and $\phi_2(Q)$, with rank scores bounded by that of G_{R_1}. □

2.5.3 Top-k Search

Using LoopyBP and sketch graph as building blocks, we next present our top-k searching algorithm. The algorithm, denoted as topK, is illustrated in Figure 1.

Given Q, G, $\mathcal{L}$ and integer k, topK initializes a top k match list L, and a Boolean flag terminate to indicate if the termination condition (as will be discussed) is satisfied (line 1). It next constructs a sketch graph G_h (lines 2-4). Given G, Q and G_h, it dynamically updates L with newly extracted matches, by applying LoopyBP over the sketch graph G_h and G iteratively (lines 5-9). More specifically, topK repeats the following two steps, until the termination condition is satisfied (terminate = true).

Algorithm 1 Algorithm topK

```
Input: $G$, $Q$, $\mathcal{L}$, integer $k$;
Output: $L$: top $k$ ranked matches;
1: top $k$ list $L = \emptyset$; terminate =false;
2: for each $v$ in $Q$ do
3:     initialize valid match candidates w.r.t. $\mathcal{L}$;
4: end for
5: construct sketch graph $G_h$;
6: $G_R$ := LoopyBP($G_h$);
7: while terminate = false do
8:     update $L$ with top $k$ matches from LoopyBP($G_R$);
9:     $G_R$ := LoopyBP($G_h$);
10:    update terminate;
11: end while
```

(1) The algorithm topK first performs LoopyBP over G_h, and produces a best upper level match of Q, *e.g.,* G_R, as a subgraph of G_h (line 5). Note that G_R corresponds to a subgraph of G, induced by all the nodes in G that are contained in the hyper nodes of G_R following edge matches.

(2) topK then "drills down" G_R to obtain the subgraph it corresponds to, and

conducts LoopyBP over the subgraph to update L with more accurate lower level matches (line 7). Matches in L are replaced with new matches with higher scores. In addition, topK also performs necessary propagation over the subgraphs from earlier upper level matches, if they contain nodes with updated scores due to messages from the updated matches in L. It updates L with new lower level matches from these subgraphs, if any, until no more new matches can be identified to update L. It next extracts a next upper level match G_R from G_h (line 8).

The above steps (lines 7-8) complete a round of processing. At the end of each round, topK checks if the termination condition below is satisfied (line 9): (a) L already contains k matches, and (b) the match ranked at k in L already has a score higher than the next upper level match G_R (if any) from G_h. If the condition is satisfied (or all possible matches in G are visited), topK terminates and returns L (line 10). Otherwise, it extracts a new high level match from G_h, and repeats steps (1) and (2).

Analysis. topK always terminates, as the message (value) propagation stops when the change of the value is below a threshold. Moreover, the top k matches returned by topK will be the same as those returned by LoopyBP if sketch graph is not involved, due to Lemma 1.

For the complexity, one may verify the following. (1) It takes $O(|V_Q||V||\mathcal{L}|)$ time to identify all the partition sets, and construct G_h (lines 2-4). Note that we assume every transformation checking is done in constant time, as remarked earlier. (2) The total runtime consists of two parts: the upper level LoopyBP (over G_h) and the lower level LoopyBP (over G). The upper level LoopyBP (line 5,8) takes in total $O(I_1|\mathcal{L}|^2|Q|^3)$ time, since G_h has in total $|V_Q||\mathcal{L}|$ nodes, and it takes $O(I_1|Q|(|\mathcal{L}||V_Q|)^2)$ time for upper level LoopyBP, where I_1 denotes the total number of upper level iteration. Note that the performance of upper level LoopyBP is independent of the size $|G|$. (3) The lower level LoopyBP (line 7) takes in total $O(I_2|Q||V_t|^2)$ time, where I_2 is the iteration number for lower level LoopyBP, and $|V_t|$ denotes the total number of nodes in G visited by lower level LoopyBP when topK terminates. Putting these together, algorithm topK takes in total $O(|V_Q||V||\mathcal{L}| + I(|Q|^3|\mathcal{L}|^2 + |Q||V_t|^2))$ time, for in total I (*i.e.,* I_1+I_2) iterations.

In practice, $|Q|$ and $|\mathcal{L}|$ are typically small. Moreover, indexing techniques to efficiently identify node matches (lines 2-4) can be readily applied, reducing its time complexity from $O(|V_Q||V||\mathcal{L}|)$ to $O(|V_Q|)$ (see Section 2.6). Our experiments show that topK achieves near-linear runtime w.r.t. graph size (see Figure 2.10 in Section 2.8). A possible reason is that most of possible node matches are not connected with each other in terms of edge matches. The number of message passing among them is much smaller than the worst case $|V_t|^2$. They cannot form a high quality subgraph that matches the entire query graph. In the first few iterations, they are quickly pruned by LoopyBP.

Example 7: Consider the query Q in Figure 2.4. The algorithm topK finds top 2 matches for Q in G as follows. (1) topK first computes a sketch graph G_h of G. (2) It then computes a top ranked result G_{R_1} from G_h, where the node `UCB` in Q is matched (via transformation "Acronym") with a hyper node that contains the node `University of California, Berkeley`. topK then computes a top K list by drilling down G_{R_1} (Figure 2.4), and identifies two lower level matches $\phi_1(Q)$ and $\phi_2(Q)$ from G_{R_1}, indicating actors in the movie "Mission:Impossible." (3) It next identifies a second high level match G_{R_2}, specified by "Bag of words" and "Acronym." Without drilling-down to lower level matches, topK identifies that the ranking score of G_{R_2} is already lower than $\phi_2(Q)$. This indicates that no lower level matches better than $\phi_2(Q)$ can be found. topK thus returns $\phi_1(Q)$ and $\phi_2(Q)$ as the top 2 matches. □

2.5.4 Exact Matching for Trees

When Q is a tree, which is quite common in practice, topK can be readily revised, leading to efficient *exact* top-k search.

Algorithm. The algorithm topK for tree queries iteratively performs LoopyBP over G_h and G, similarly as for general graph queries. The difference is that it uses a simplified propagation: it only performs two passes of propagation to extract an optimal match [153]. More specifically, given a tree query Q, it designates a *root* in Q, and denotes all nodes as *leaves*. topK then computes a top ranked match by conducting two passes of propagation: one from the matches for all leaves to those of the root, and the other from the matches of the root to all the matches of the leaves. It repeats the process to fetch top k best results.

Correctness and Complexity. Following [153], the two passes of propagation in topK for a tree query Q is guaranteed to converge in at most m steps, where m is the diameter of Q, *i.e.,* the length of the longest shortest path between two nodes in Q. Moreover, the propagation computes the exact rank value $P(\phi(Q)|Q)$ (Section 2.4). The correctness of topK hence follows. One may verify that topK is in total $O(|V_Q||V||\mathcal{L}|+|Q|^3|\mathcal{L}|^2+|Q||V_t|^2)$ time over tree queries, with (a) 2 passes of propagations, and (b) each propagation directs messages up to a few steps in both G_h and G. Here V_t is similarly defined as its counterpart for general queries.

2.6 Indexing

The remaining issue is to find transformed matches of query nodes quickly. For example, a node in Q with label "Chris Pine" shall be matched to a node in G has a label "Chris," "Pine," "C. Pine," etc. A straightforward method rewrites each label l from Q to a label set using all possible transformations, and inspects

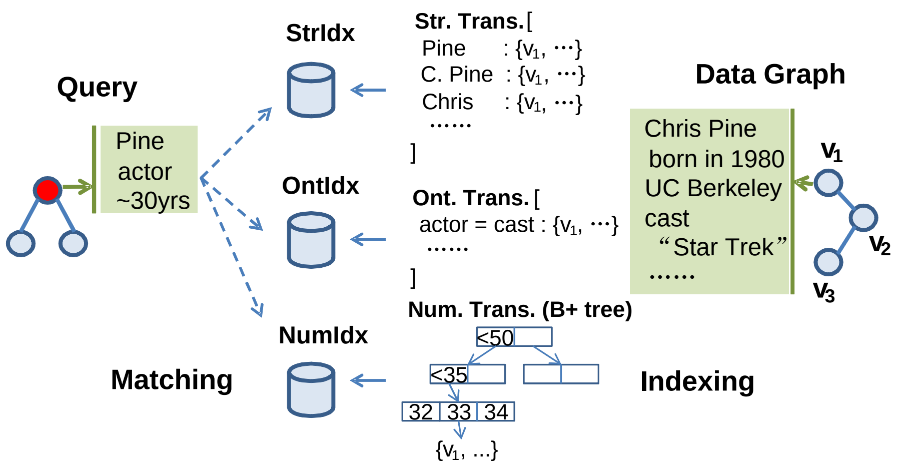


Figure 2.5: Indexing for matching.

every node label in G to find matches. Obviously, scanning the entire graph is expensive. For each (or each category of) transformation, an appropriate index is needed to support fast search.

Several indices are adopted in SLQ, in accordance with the category of the transformations it supports in Table 2.1. Nevertheless, experimenting various kinds of indexing techniques is not the focus of this work.

(1) String index, StrIdx, is built for all the string labels in G. The index contains a list of key-value pairs $<l, S_l>$, where (a) each key is a distinct label l, and (b) S_l is a node set, such that each node v in S_l has a label l_v, such that $f_i(l, l_v) = 1$ for string transformation f_i. In other words, S_l corresponds to the matches of nodes

who have labels that can be transformed from label l via string transformations. The nodes in S_l are further grouped in terms of their associated transformations to form a partition of S_l.

Let D be the set of all the labels in G. To construct StrIdx, each transformation is applied on each label of all the nodes in G. The transformed label set is denoted as Λ, which hence forms the keys in StrIdx. For each key l, nodes with labels associated to l via a transformation are grouped as a set S_l. The pair $<l, S_l>$ is then inserted to StrIdx as an entry. One may verify that (1) the construction of StrIdx takes $O(|\mathcal{L}||D|)$ time, and (2) the space cost of StrIdx is in $O(|\mathcal{L}||\Lambda||V|)$ for at most $|\mathcal{L}|$ string transformations. SLQ does not necessarily build a specific index for each transformation. (1) Transformations can be grouped according to their category (*e.g.,* "String"), supported by a single index (*e.g.,* StrIdx). (2) Searching for some transformations, *e.g.,* Unit Conversion, can be trivially performed as direct mapping. As demonstrated in Section 2.8, the worst case space cost is seldom demonstrated. The index size can be further reduced by index optimization *e.g.,* [154].

(2) Semantic index, OntIdx, leverages the indexing techniques in [157] and [85], to help identify the matches based on semantic transformations, *e.g.,* Ontology and Synonym.

(3) Numeric index, NumIdx, is constructed for searching involving labels with numeric values, *e.g.,* $\leq$ 35 yrs (Range). SLQ builds NumIdx as B+ tree over numeric values.

Figure 2.5 illustrates the above indexing techniques. As an example, for a node v_1 with label "`Chris Pine`" in G, StrIdx performs string transformations, *e.g.,* Last token, and identifies the label "`Pine`" as a key. It then insert node v_1 into the value entry corresponding to key "`Pine`." Analogously, the nodes, *e.g.,* "`Robert Pine`" and "`Peter Pine`," in G will be mapped to the same entry, associated with key "`Pine`" and transformation Last token.

For each label l in every query node, SLQ searches for the node match candidates by looking up the label (key) in the indices StrIdx, NumIdx and SemIdx. The candidates for label l refer to all the entry values in the indices corresponding to the key l. Thus, it takes only $O(|V_Q|)$ to find the transformed match candidates. Indeed, as verified in Section 2.8, with the indices, the time for finding transformed match candidates accounts for less than 2% of the total search time.

2.7 Extensions

The architecture of SLQ can also support *typed queries*, and *partially connected queries*.

Typed queries. Users may pose explicit type constraints on queries. For example, the query node "`30 yrs`" (Figure 2.1) can be specified with a type "`actor`." To cope with typed queries, SLQ defines a *type feature function* for a query node v and its type s_v as

$$F_S(v, \phi(v)) = \sum_i \gamma_i f_i(s_v, s_{\phi(v)}) \tag{2.7}$$

with the transformations $\{f_i\}$ applied to the node types.

Partially connected queries. A partially connected query Q contains several connected components. Note that a keyword query is a case of partially connected queries. A user submits partially connected queries when he is not clear about the connection among these nodes. To cope with such queries, a new query Q' is constructed by inserting a set $\tilde{E}$ of implicit edges, where each edge $\tilde{e}$ bridges a pair of nodes from different components. An *implicit edge feature function* can be readily introduced as

$$F_{\tilde{E}}(\tilde{e}, \phi(\tilde{e})) = \sum_i \delta_i f_i(\tilde{e}, \phi(\tilde{e})) \tag{2.8}$$

Both $F_S(v, \phi(v))$ and $F_{\tilde{E}}(\tilde{e}, \phi(\tilde{e}))$ can be plugged into the ranking function Eq. 2.3, where $\{\gamma_i\}$ and $\{\delta_i\}$ can be learned using the same training strategy.

2.8 Experimental Evaluation

In this section, we perform a set of experiments using real-life large graphs, to demonstrate `SLQ` framework in terms of effectiveness, efficiency and scalability.

2.8.1 Experimental Settings

Datasets. We use three real knowledge graphs in Table 1.1. (1) *DBpedia* [1] is a knowledge base. Each node represents an entity associated with a set of properties, (*e.g.,* `name`='california', `type`='place', `area`='163,696 sq mi'). The labeled edges indicate various relationships. (2) *YAGO2* [66] is a knowledge base gathered from several open sources. Similarly as *DBpedia*, its nodes and edges preserve rich information. (3) *Freebase* [2] is a collaboratively created graph base that has over 40M topics (nodes) and 1.2B facts. As public repositories, these graphs are maintained by multiple communities, containing highly diverse and heterogeneous entities, attributes and values.

Transformations. Our system integrated all of the transformations in Table 2.1 including ontology [157]. More transformations can be seamlessly adopted.

Queries. In the experiments, two sets of query benchmarks are employed. (1) The *DBPSB* benchmark [106] is derived from *DBpedia*. The benchmark is a set of 25 query templates that are originally expressed in *SPARQL* format. The templates resemble real query workload and cover queries with different complexity. The queries can be converted to graph queries. (2) The templates in *DBPSB* have limited types (e.g., "Person") and simple topology (*e.g.,* tree). We hence designed a second set of 20 templates that explore more diverse topics and complex (*e.g.,* cyclic) graph structures.

In offline learning, query templates are generated by instantiating the query benchmarks with the labels from the data graphs. Here a label can be any property of the corresponding entity. These instances also serve as ground truth for the queries. We then perform transformations on randomly selected labels in each query instance, which yield training queries. We show three such queries and their matches in Figure 2.6. Query 1 is to find an athlete in football team "San Francisco 49ers" who is about 30 years old. Query 2 is to find a person who served in the Union army and attended a battle, and these information *maybe* related with "Missouri." Query 3 identifies a current US senator at his 60 who lives in "NJ" and knows "F. Lautenberge."

Algorithms. We chose the CRFs model as defined in Eq. 2.3 and developed SLQ in Java. For comparison, the following algorithms are also developed with the best effort.

Baselines. To compare the match quality, we consider the following state-of-the-art techniques. (1) Spark [96], a keyword-based search engine. It supports IR-style ranking heuristics. Since Spark only supports exact string matching, we modified it to accept transformed matches. Spark does not consider edge information in a query as it is keyword oriented. (2) Unit is a variant of SLQ. The only difference is it uses a revised ranking model with equal weight for all the transformations; (3) Card also implements SLQ, while revises its ranking model with weights equal to the selectivity of the transformations as $\frac{1}{card(f)}$. Here $card(f)$ refers to the average size of the matches for a randomly sampled node (or edge) in the graph using transformation f.

For efficiency comparison, we compare SLQ with (1) Exact, which enumerates all possible matches based on the subgraph search algorithm [86, 113], and then rank them with the learned ranking model. This strategy ensures that all the matches including the ground truth can be obtained and ranked. (2) Approximate searching in NeMa [79]. The method directly applies a propagation strategy similar to LoopyBP over data graphs. Note that NeMa only extracts the most probable result, *i.e.,* top 1 match. We enhanced it by applying the techniques in [161] to identify top-k results. For fair comparison, the above baselines are also equipped with our predefined transformations and the indices.

Metrics. Given a query workload $\mathcal{Q}$ as a set of queries Q, we adopt several metrics for the rank evaluation: (1) Precision at k ($P@k$), the number of the top-k answers that contain the ground truth; (2) Mean Average Precision (MAP @k), which means $\mathsf{MAP}@k = \frac{1}{|\mathcal{Q}|}\sum_{Q\in\mathcal{Q}}\frac{1}{k}\sum_{i=1}^{k}P(i)$, where $P(i)$ is the precision at cut-off position i when the ith result is a true answer and $P(i) = 0$, otherwise; (3) Normalized Discounted Cumulative Gain ($\mathsf{NDCG@k}$), as $NDCG@k = \frac{1}{|\mathcal{Q}|}\sum_{Q\in\mathcal{Q}}Z_k\sum_{i=1}^{k}\frac{2^{r_i}-1}{\log_2(i+1)}$, where r_i is the score of the result at rank i. Following convention, we set r_i as 3 for the *good* match, 1 for the *relevant* match and 0 for the *bad* match. Z_k is a normalization term to let the perfect ranking have score 1. We also tested other metrics, such as SoftNDCG [139]. They share similar intuition and thus are not elaborated. In the experiments, unless otherwise specified, each query workload refers to $1,000$ randomly generated queries using different query templates. Note that the set of training queries is different from that for testing.

Setup. We compressed each data graph, *e.g.,* same predicates in the RDFs, and built index based on the transformations. The indexing time and the size is: 61.8min/1.02GB (DBpedia), 37.4min/0.78GB (YAGO2), 263min/12.91GB (Freebase). All the experiments were performed on a machine with Intel Core i7 2.8GHz CPU and 32GB RAM. For each test, we report the average value over 5 runs.

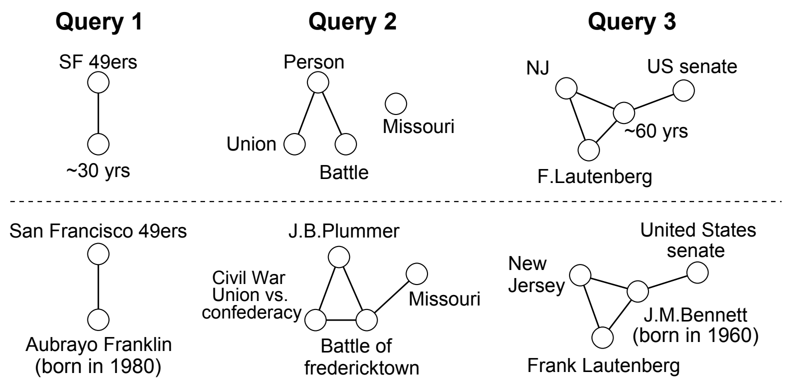


Figure 2.6: Case study: querying knowledge graphs.

2.8.2 Case Study: SLQ vs. IR-based Search

We provide a case study using *DBpedia*. Consider the three queries in Figure 2.6. For each query, SLQ identifies meaningful matches of high quality. For example, for Query 2, a historical figure, Colonel J.B. Plummer, is identified to match `Person` who fought in the Battle of Fredericktown during the Civil War in Missouri. Our framework is able to tell the importance of different transformations: for `Person`, *Ontology* is a proper transformation; while for `Union`, *Bag of words* is promoted in the ranking. `Missouri` is selected as an exact match. In addition, the match suggests a direct connection between Missouri and Battle in Query 2, indicating a *refinement* of Query 2 in future. In all cases, Spark gives low IR score and cannot identify matches for Query 2.

2.8.3 Experimental Results

Exp-1: Manual evaluation. We first conduct manual evaluation on 75 queries that are randomly constructed from the three datasets. 10 students help evaluate the results returned by our search algorithm. For each result, a label, *i.e., Good*, *Relevant* or *Bad*, was assigned by the students regarding the query. The labels are thus considered as the ground truth. The students were not trained beforehand and thus the labels were assigned merely based on their intuition. The metric, $NDCG@k$, can be calculated based on the rank order of the results and the corresponding labels. Table 2.2 presents the quality of top-5 returned answers. The result confirms that SLQ shows a substantial improvement over the baselines. In terms of answer quality, it is very close to the exhaustive search algorithm, Exact. On the other hand, SLQ is up to 300 times faster than Exact (see Exp-3 for query processing time comparison).

Graph	Spark	Unit	Card	SLQ	Exact
DBpedia	0.707	0.790	0.858	0.935	0.935
YAGO2	0.682	0.849	0.852	0.926	0.928
Freebase	0.636	0.751	0.768	0.859	0.865

Table 2.2: Manual evaluation ($NDCG@5$).

In the following experiments, we verify the performance of our algorithms by varying query size and transformation ratio. Since finding good matches manually is very costly, we focus on two kinds of intuitively good matches: the original subgraph from which a query is transformed from, and all the identical matches of the query. A good algorithm shall at least rank these good matches as high as possible.

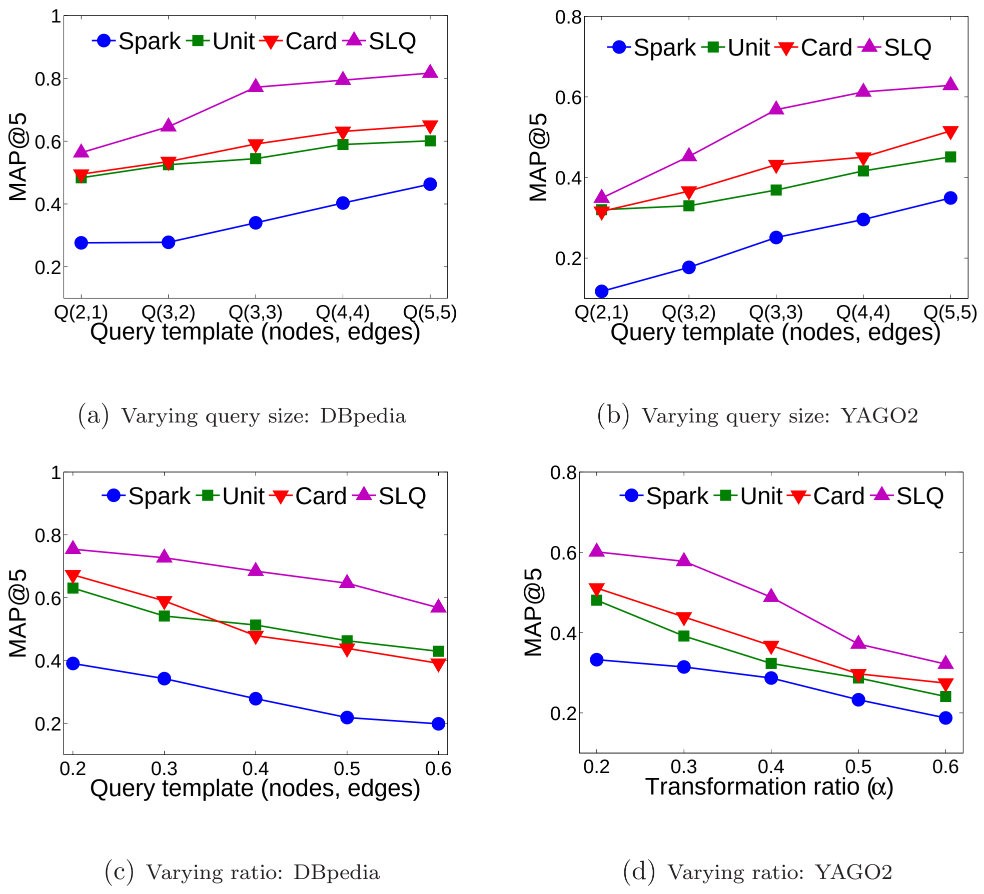


Figure 2.7: Effectiveness of ranking ($MAP@5$).

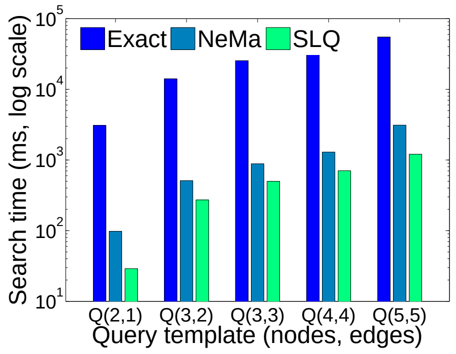
Exp-2: Effectiveness of ranking. This experiment examines the answer quality of SLQ. There are several factors, such as query size, query topology, transformation ratio and data graph, that may affect the ranking. We first study the impact of the query size and topology while fixing the others. The following test is based on the evaluation of the query workload that are randomly sampled from the data graph *w.r.t.* the query templates. Each query is modified by applying random transformations with the ratio $\alpha = 0.3$. The ranking model was trained beforehand for each graph (see Exp-4 for the report on offline training).

Given the queries and the corresponding results, we employ $MAP@k$ as the metric to evaluate the rank and plot the scores in Figure 2.7(a-b) for *DBpedia* and *YAGO2*, respectively. The results tell us that the methods Unit, Card and SLQ significantly outperform the IR based technique, Spark. The ranking model in Spark only considers a linear combination of keywords' IR scores. It does not take the selectivity of different transformable conventions and the relations (connections) of the keywords into account. SLQ also achieves better ranking result than its two variants, Unit and Card, indicating that automatic learning of transformation weights could improve answer quality. Figure 2.7(a-b) also show that when the query size increases, the score increases for all the methods. This is due to the fact that a query with larger size provides more evidences, which help identify good matches easily. This phenomenon implies great potential of the schemaless and structureless querying model: As long as a user provides enough evidences, she can find the answer even her query does not fully comply with the schema and the structure of the underlying graph database.

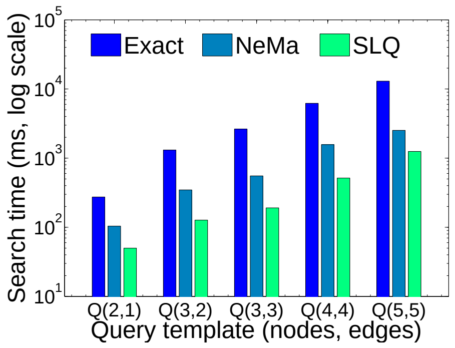
We next validate the ranking quality with respect to the transformation ratio α. In this test, we derive a set of query workload by varying α of the queries from 0.2 to 0.6. Intuitively, to raise the transformation ratio will increase the "ambiguous" level of the query, making it more difficult to find the true match in the top-k matches. The result is depicted in Figure 2.7(c-d). As expected, the performance of all the algorithms degrades along with the increase of α. However, our algorithm is still the best. We also examined the performance of Exact, which is slightly better (by $\leq 1\%$) than SLQ and thus is not shown in Figure 2.7 for simplicity.

Exp-3: Efficiency of top-k search. In this experiment, we demonstrate the runtime improvement of SLQ over Exact and NeMa. SLQ employs graph sketch to quickly skip the low-quality matches. We choose $k = 20$, and use the same query workload as in the previous experiment. The runtime examined here also contains the index search time, which accounts for less than 2% of the total time. The runtime of Unit and Card is not reported as it is close to that of SLQ.

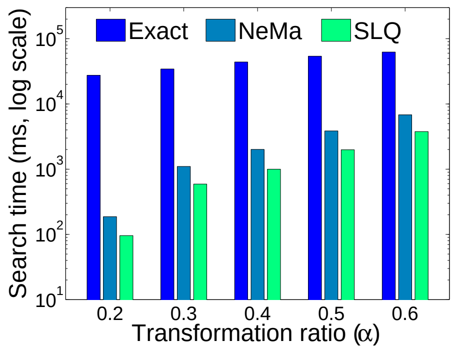
Figure 2.8(a-b) shows the runtime with varying query size, and fixed transformation ratio (0.3). For both graphs, SLQ and NeMa are 5-50 times faster than Exact. This advantage is achieved by top-k search and the merit of approximate search (LoopyBP). Meanwhile, SLQ is 2-4 times faster than NeMa. It implies the graph sketch method can indeed avoid some unnecessary verification. We also



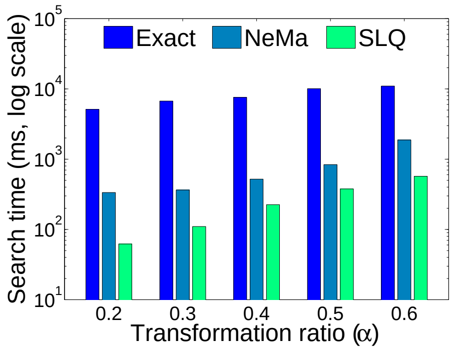
(a) Varying query size: DBpedia



(b) Varying query size: YAGO2



(c) Varying ratio: DBpedia

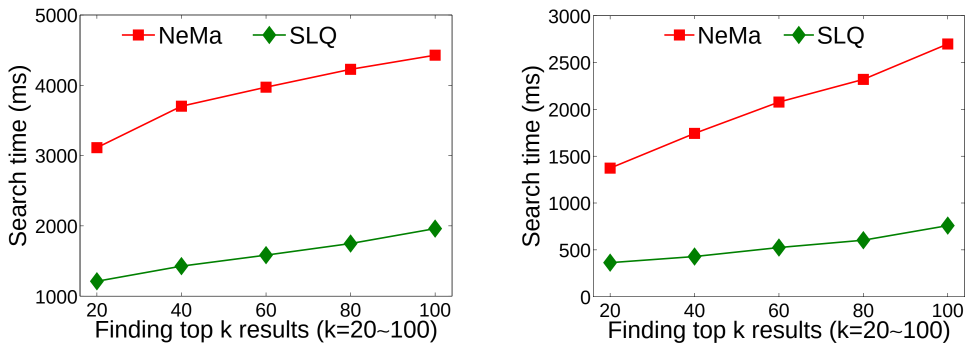


(d) Varying ratio: YAGO2

Figure 2.8: Efficiency of top-k search ($k = 20$).

evaluate the runtime of SLQ by varying the transformation ratio from 0.2 to 0.6. Figure 2.8(c-d) show a clear advantage of SLQ over other approaches. For most queries, SLQ can finish the execution within 1 second. Its runtime can further be reduced by employing a multi-thread implementation.

Figure 2.9(a-b) plots the search time with different k values for *DBpedia* and *YAGO2*, respectively. The test queries are randomly generated with transforma-



(a) Top-k search: DBpedia (b) Top-k search: YAGO2

Figure 2.9: Search time: effect of k (20 $\sim$ 100).

tion ratio 0.2 $\sim$ 0.6. Our algorithm again demonstrates outstanding performance on runtime, which is up to 1/3 of the time by NeMa.

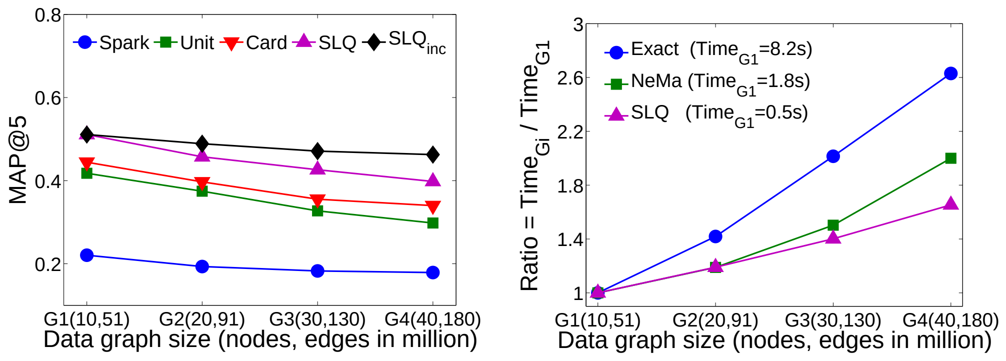
Exp-4: Offline learning. We study the impact of sample size and offline training on the quality of ranking. Recall that the training queries along with the ground truth are randomly sampled from the graph, the coverage of the queries plays a pivotal role in the training. The coverage of a query workload $\mathcal{Q}$ is defined as $C(\mathcal{Q}) = \frac{|\bigcup_{Q \in \mathcal{Q}} Q|}{|G|}$. Since the graphs are highly heterogeneous, we speculate with larger coverage, the learned model would have a better ranking result. To inspect the effect, we conduct two tests with different workload coverage: 0.5% $\sim$ 2.0% (*DBpedia*) and 0.05% $\sim$ 0.2% (*Freebase*). The queries in each training workload are generated from randomly selected query templates.

DBpedia			Freebase		
Sample	Time	$P@5$	Sample	Time	$P@5$
0.5%	795s	0.650	0.05%	$1,695$s	0.685
1.0%	$1,588$s	0.715	0.1%	$3,125$s	0.712
2.0%	$3,028$s	0.722	0.2%	$5,828$s	0.725

Table 2.3: Sample coverage for training.

The training time and the quality of ranking ($P@k$) are shown in Table 2.3. The transformation ratio for each training set is controlled by a 5-fold *cross validation*. Note the test queries are different from those for training. For both of the two datasets, the training time is nearly linear *w.r.t.* $C(\mathcal{Q})$. It can be seen that with higher coverage, we can achieve a clear better ranking performance, with the cost of extra training time. For *DBpedia*, when the coverage increases from 1.0% to 2.0%, the improvement is marginal, *i.e.,* $\leq$ 1.0%. The same effect can be observed for *Freebase*, when we increase the coverage from 0.1% to 0.2%. The experiment validates that only a small sample of the raw data for offline training is enough for good performance.

Exp-5: Scalability w.r.t. graph size. We next evaluate the scalability of SLQ by varying the size of the Freebase graph. Specifically, we initialize a subgraph G_1 from Freebase with size $(10M, 51M)$ (*i.e.,* 10M nodes and 51M edges) and gradually grow it to $G_4(40M, 180M)$. This setting will test the performance of SLQ in a streaming mode. Figure 2.10(a) depicts the result. Specifically in the

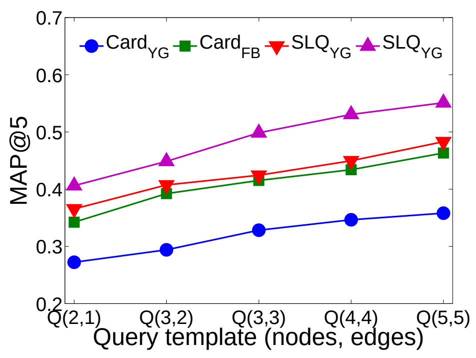


(a) Effectiveness: MAP@5 (b) Efficiency: search time

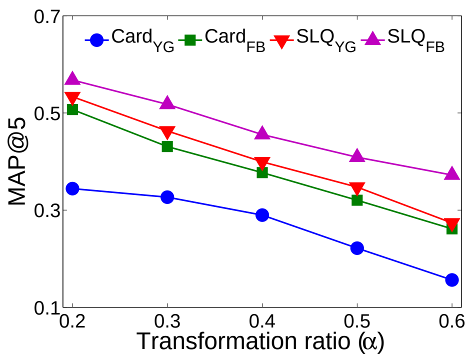
Figure 2.10: Scalability evaluation on Freebase.

figure, SLQ shows the performance of the ranking model trained only based on the initial graph (G_1), while SLQ$_{inc}$ shows the performance of the ranking model with incremental update based on the growing graph. The test queries are generated separately for each graph. With the growing of the graph, the rank performance generally decreases since there are more data to confuse top-k ranking. Among the four algorithms, SLQ$_{inr}$ is the best. Moreover, although it degrades *w.r.t.* SLQ$_{inr}$, SLQ still outperforms the other methods dramatically, indicating a comparatively stable result.

In terms of search time, to illustrate the significant time difference, we plot the runtime increasing ratio, $\frac{Time_{G_i}}{Time_{G_1}}$, in Figure 2.10(b), for top-k search ($k = 20$). All the algorithms take more time for searching larger graphs. Moreover, despite the significant difference on the search time on G_1, *i.e.*, $Time_{G_1}$ as shown in the legend of Figure 2.10(b), SLQ achieves near-linear runtime increase regarding the size of the graph. It takes up to 25% of the time by NeMa and is at least one order of magnitude faster than Exact. We also inspect the runtime of SLQ$_{inr}$. Recall that the model in SLQ$_{inr}$ is continuously updated, the training time is negligible. With the setting of 0.1% training sample coverage and 1000 test queries, the amortized runtime of SLQ$_{inr}$ is at least as twice as that of SLQ and thus is not shown in Figure 2.10(b) for simplicity.



(a) Varying query size



(b) Varying ratio

Figure 2.11: Cross query evaluation on Freebase.

Exp-6: Training on YAGO2 and querying Freebase. Finally, we do a bold experiment: Can we apply the model trained on one graph and query another graph? To answer this question, we test the model trained on YAGO2 by ranking the results of queries on Freebase (SLQ$_{YG}$), and compare it with the models trained on Freebase (SLQ$_{FB}$). Figure 2.11(a) reports the result by varying the query size.

The transformation ratio is 0.3. The model in SLQ_{YG} is the same as that trained for YAGO2 in Exp-2. It shows SLQ_{YG} still works, and is even slightly better than Card_{FB}. This is a strong evidence showing that the knowledge learned (the weights of different transformations) can be transferred between different graph databases. A similar result is also observed when we vary the transformation ratio, as shown in Figure 2.11(b).

2.9 Related Work

Graph searching is studied for structured queries (*e.g.,* XQuery, SPARQL), keyword queries [75,86,152] and graph pattern queries (*e.g.,* [10]). These methods focus on fixed schemas and ranking functions. To relax the constraints of schema and structure, approximate matching is studied, for *e.g.,* graph pattern matching [28, 79], and for keyword queries over knowledge graphs [75]. The searching semantics are relaxed to identify more meaningful matches with similar structures or similar attributes to a given query.

Closer to our work is NeMa [79] and NAGA [75]. (1) NeMa defines node similarity by comparing the neighborhood similarity of two nodes, and iteratively infer the matching quality using similarity propagation as in a graphical model. (2) NAGA supports keyword querying over the YAGO knowledge base. It defines match quality with confidence, informativeness and compactness, and ranks the answers based on probabilistic models, where the parameters in the ranking model are tuned by users. Nevertheless, all of these studies use predefined ranking metrics. This significantly limits the power of these methods as it is hard to justify them. Our work shows that a ranking model shall be learned automatically through the existing queries and their associated answers, not given beforehand.

Machine learning techniques are leveraged to find matched entity pairs by combining multiple similarity metrics. For example, weights of various transformation rules are learned for object identification [140]. These methods differ from ours in the following. (1) Time-consuming manual labeling and training data. In contrast, our system requires no manual effort for generating training examples. (2) Homogeneous data. Thus, they can not be easily extended to deal with heterogeneous graphs as studied in this work.

There are several other topics complementary to our focus, including keyword search in graph data using IR techniques. Query interpretation [36] provides a user with multiple plausible interpretations of a query. These techniques can be combined with our framework to further improve the quality of query results.

2.10 Summary

We identified a key problem that frustrates the users for accessing emerging graph databases. We argued that a user-friendly query engine must support various kinds of transformations directly, such as synonym, abbreviation, and ontology. We developed a novel searching framework, SLQ, to (a) learn a ranking model that combines multiple transformations, which does not require manually labeled training instances; and (b) efficiently find top-k matches for graph and keyword queries. As verified by our experiments, SLQ achieves much better query results in comparison with the existing approaches and is able to process queries quickly. Better still, SLQ can be readily extended to integrate new transformations, indices and query logs. Surrounding this new query paradigm, there are a few emerging topics worth studying in the future, *e.g.,* comparison of different probabilistic ranking models, compact transformation-friendly indices, and distributed implementation.

Chapter 3

Fast Top-k Search in Knowledge Graphs

Top-k graph querying is routinely performed in real-life graph data. Given a graph query Q posed on a knowledge graph G, the problem is to find k (approximate) matches in G with highest matching scores. In contrast to conventional graph databases, knowledge graph querying comes with new challenges and opportunities: (1) Matching scores are dynamically generated; pre-built indices are not available, (2) Query answers are often inexact, and (3) Query graphs are usually small. We show that the threshold algorithm (TA) is not instance-optimal any more in this new setting.

In this chapter, we introduce STAR, a framework to exploit fast top-k search for star queries and efficiently assemble them to answer general graph queries. STAR has two components: A fast top-k algorithm for single star-shaped queries and an assembling algorithm for general graph queries. The assembling algorithm uses fast star querying as building blocks and iteratively sweeps the star match lists with dynamically adjusted bound. When the size of Q and k is bounded by a constant, we show that the time complexity of answering single star queries is linear to $|E|$, the edge number of G. For approximate graph matching where an edge can be matched to a path with bounded length d, we extend the algorithm using message passing and achieve time complexity $O(d^2|E|+m^d)$, where m is the maximum node degree in G. Our algorithm does not require any pre-built indices. Using three real-life knowledge graphs, we experimentally verify that STAR is 5-10 times faster than the state-of-the-art TA-style subgraph matching algorithm, and 10-100 times faster than a graph search algorithm based on belief propagation.

3.1 Introduction

Top-k subgraph search has been applied to extract best answers from real-world graphs, *e.g.,* information and social network [44,58], knowledge graphs [67, 160,169] and communication networks [166]. *Given a data graph G, a scoring function F, and a query Q, top-k subgraph search over G returns a set of k answers with highest matching scores.* Top-k subgraph search is fundamental in many graph analytical tasks.

A common practice in existing top-k subgraph querying [29,38,55,121,141,163, 167] assumes pre-sorted lists of matches for single nodes/edges or small subqueries, and follows conventional top-k aggregation methods over relational databases, *e.g.,* threshold algorithm [41], to find top matches by traversing the lists. Nevertheless, emerging real-world graphs such as knowledge graphs having rich node/edge information often require inexact matches in terms of content and structure. This new requirement, together with the large graph size, introduces new challenges and opportunities.

(1) The matching scores are *dynamically* generated. That is, the matching score of potential answers are computed at run time based on a similarity function applied on the query graph and the matches, rather than predefined weights or similarity matrix [167]. It is costly to sort all possible matches from scratch for each individual query. Sorting all the node/edge lists takes $O(|V|\log|V| + |E|\log|E|)$ time for G with $|V|$ nodes and $|E|$ edges, which is not going to provide real-time response to queries.

(2) Queries typically have *inexact* matches that can no longer be captured by strict isomorphic mapping. A common example is subgraph searching in knowledge bases [160]. A query node, often described as a few keywords, can be matched ambiguously with a number of entities. A query edge could have valid matches

with paths of bounded length. Finding such inexact matches are costly over big graphs. While indices can be constructed to speedup searching, it often comes with expensive preprocessing, *e.g.,* $O(|V|^3)$ for computing transitive closure [55, 121]. This is no longer practical for big graphs in terms of time and space.

(3) Query graphs are usually not big. As observed in [47], most real-world SPARQL queries in RDF stores such as DBpedia are star-like; and 98% of knowledge-base graph queries have diameter 2. In this case, to optimize the complex graph search is an overkill. Instead, it is good enough to construct a fast query processing engine for simple structures.

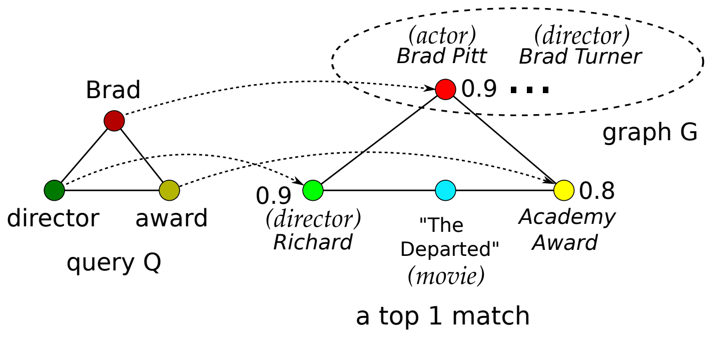


Figure 3.1: Top-k subgraph querying.

Example 8: Consider graph query Q on a movie knowledge graph, shown in Figure 3.1. It searches for top-2 movie directors who worked with "Brad" and have won awards.

It is nontrivial to find the top answers. Each query node and edge may correspond to an excessive number of possible matches. For example, a node `Brad` may have matches with any person whose first or last name is `Brad`, *e.g.,* `Brad Pitt` and `Brad Turner`. An edge (`director`, `award`) may match a path through an intermediate node `movie`. It is not practical to find the best answer for Q by first enumerating all the possible matches and then ranking them.

Furthermore, the quality of a match for Q can only be dynamically evaluated by a similarity function, rather than being aggregated from static, predefined weights. It is not practical to assume there is a pre-sorted node and edge list, or an index for all kinds of query nodes and edges. $\square$

This calls for an efficient top-k subgraph search framework to cope with the new challenges. Specifically, we ask (1) How to find top answers in the presence of dynamically generated scores, where expensive preprocessing (*e.g.,* indices and sorted lists) is no longer practical? (2) How to efficiently find top-k matches for inexact (*e.g.,* edge to path) matching over big graphs? Conventional top-k querying strategies no longer fit the new demand.

In this chapter, we introduce STAR, a top-k subgraph matching framework that copes with the new challenges. In a nutshell, it develops a fast query processing engine for popular star-shaped queries and makes use of this engine to solve more complex graph queries.

(1) Given a star-shaped query Q^* and data graph G, STAR finds top-1 matches for the center query node (called *pivot node*) in G and expands new matches from there. For each node matching the pivot node, it is able to generate the best matches of Q^* in decreasing order of matching score. The algorithm takes $O(|E|)$ time, assuming k and the size of Q^* are bounded by a constant.

(2) We further extend the algorithm to support inexact matching where an edge can be matched to a path with bounded length d. For each node/edge in data graph, it propagates dynamically generated matching scores to their neighbors, retains the maximal matching score and then propagates further. Since it does not compute transitive closure, the time complexity is reduced to $O(d^2|E| + m^d)$, where m is the maximum node degree of G.

(3) Given a more complex graph query Q, STAR decomposes it to a set of star-queries, and assembles top answers from individual star queries. Since STAR generates top answers of star-queries in monotonic decreasing order of matching score, the answer set is equivalent to a pre-sorted list! This nice property makes it possible to apply monotonic ranked joins such as [121] to produce the final top-k answers for Q.

(4) Since a query Q can be decomposed to multiple star queries in different manners, we introduce a few query optimization opportunities unique in our problem setting. While this work is not going to test many optimization ideas developed for relational databases, we experiment a few designs and demonstrate their effectiveness.

(5) We evaluate the scalability of our algorithms. In comparison with a highly-optimized threshold-based algorithm (TA) and a belief propagation method (BP) employed in [79, 160], it was found that STAR is 5-10 times faster than TA and 10-100 times faster than BP.

We conclude that optimizing star query processing not only solves the most popular queries in knowledge graphs, but also contributes a building block for answering more complicated graph queries. By effective query decomposition and fast star query processing, top-k matches can be found in a much faster manner.

To the best of our knowledge, we are among the first to recognize the problem of searching top-k (approximate) subgraph matches with dynamically generated matching scores, and the difficulty of building indices and transitive closures to facilitate such search in large graphs. Most of the existing algorithms for keyword-based top-k search like [59], twig/tree queries [55,121], and ranked joins [112] that rely on indices are not valid any more for this new setting. Approximate search that supports edge-path inexact matching using transitive closures [55] is not scalable in terms of graph size. Traditional network alignment tools developed for biological networks such as [131] are an overkill for small graph queries popular in knowledge graphs and cannot provide real-time response. As shown in our experiments, the recently proposed brief propagation method (BP) for top-k graph search [160] is not competitive with STAR. STAR is one to two orders of magnitude faster than the BP algorithm; it is able to answer graph queries for DBpedia in 100 milliseconds using a single Intel CPU core.

3.2 Preliminaries

We start with several notions and problem formulation of top-k subgraph querying.

Data graphs. We consider *data graph* G as a labeled graph $(V, E, \mathcal{L})$, with node set V and edge set E. Each node $v \in V$ (edge $e \in E$) has a description $\mathcal{L}(v)$ ($\mathcal{L}(e)$) that specifies node (edge) information, and each edge represents a relationship between two nodes. $\mathcal{L}$ could be structured with a schema (*e.g.,* in XML, RDF, and Freebase), not structured (*e.g.,* keywords only), or with mixed structure, *e.g.,* DBpedia. $\mathcal{L}$ may also include heterogeneous entities and relations of various types, entity name or attribute values [95].

Queries. We consider query Q as a graph (V_Q, E_Q). Each *query node* in Q provides information/constraints about an entity, and an edge between two nodes specifies the relationship or the connectivity constraint posed on the two nodes. Specifically, we use Q^* to denote star-shaped query.

Example 9: Figure 3.1 illustrates querying without node schema. The query Q contains nodes as simple keywords, *e.g.,* `Brad`, to describe the entities it refers to. For each node in the data graph G, a node description $\mathcal{L}$ may specify a type (*e.g.,* `actor`) and an entity name (*e.g.,* `Brad Pitt`), or simply a keyword (*e.g.,* `Academy Award`). Note that $\mathcal{L}$ may also pertain to specified schema, where each node has uniformed attributes, and attribute values in accordance. □

Subgraph Matching. Given a graph query Q and a data graph G, a match $\phi(Q)$ of Q in G is a subgraph of G, specified by a one-to-one matching function ϕ. It maps each node u (resp. edge e=(u', v)) in Q to a node match $\phi(u)$ (resp. edge match $\phi(e)$=$(\phi(u), \phi(u'))$) in $\phi(Q)$. In Section 3.5, we will relax the edge mapping to path mapping to support approximate matching.

Assume there exists a similarity function F_V (resp. F_E) that determines a similarity score from a node (resp. edge) to its match. Given Q and a match

$\phi(Q)$, the matching score is computed by a function $F(\phi(Q))$ as

$$F(\phi(Q)) = \sum_{v \in V_Q} F_V(v, \phi(v)) + \sum_{e \in E_Q} F_E(e, \phi(e)) \tag{3.1}$$

The function measures the matching quality as the total score of the node and edge similarity it specifies. There are a plenty of similarity functions available. For example, in Section 2.4.1, we adopt a probabilistic approach to learn a similarity function based on similar matches automatically generated from data graphs. When mapping a query node/edge to a data node/edge, it supports various kinds of transformations such as synonym, abbreviation, and ontology. For example, “teacher” can be matched with “educator,” and “J.J. Abrams” with “Jeffrey Jacob Abrams.” Each match produces a similarity score. All the scores are combined together with a learned weighting function to produce a final score between query Q and its match $\phi(Q)$. In this work, we assume node and edge similarity functions, F_V and F_E are given. When it is not ambiguious, we write $F_V(v, \phi(v))$ as $F(\phi(v))$, $F_E(e, \phi(e))$ as $F(\phi(e))$ respectively.

Top-k subgraph querying. Given Q, G, and $F(\cdot)$, the top-k subgraph querying is to find a set of k matches $Q(G, k)$, such that for any match $\phi(Q) \notin Q(G, k)$, there exists a match $\phi'(Q) \in Q(G, k)$, where $F(\phi'(Q)) \geq F(\phi(Q))$.

Example 10: For Q and G in Figure 3.1, a match $\phi(Q)$ consists of nodes `Brad Pitt`, `Richard` and `Academy Award`, where the function ϕ maps `Brad` to

`Brad Pitt` with score 0.9. Let the three edge match score be 1.0, 1.0 and 0.8, then the total score $F(\phi(Q))$ is 5.1, the sum of node and edge matching scores. Note that an edge (`director`, `award`) in Q is matched with a path from `Richard` to `Academy Award` in G. □

3.3 Threshold Algorithm

We first introduce a top-k graph querying procedure based on threshold algorithm (TA) [41]. The procedure is adopted in several state-of-the-art graph pattern matching methods for *e.g.,* knowledge graph searching [169] and information network [58]. We analyze the limitation of this approach.

The threshold algorithm [41] finds the top-k best tuples from a relational table by optimizing a monotonic aggregation function. The common practice in existing top-k subgraph matching is to treat each query node and edge as an attribute, with its matching score as an attribute value. If a set of matches can be joined to form a complete match, they are selected to compute a threshold. An upper bound of the matching score is estimated from the rest "unseen" matches. Following threshold algorithm, top-k matches are identified when the upper bound is smaller than the threshold. The procedure and its variants are invoked in a range of existing subgraph matching methods [29, 38, 141, 169].

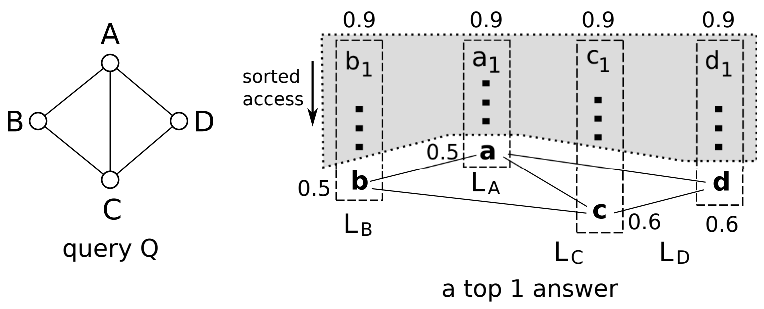


Figure 3.2: The enumerations in graphTA.

A TA-algorithm for subgraph matching. We outline the procedure, denoted as graphTA, in Alg. 2. The procedure typically follows three steps. (1) It initializes a candidate list L for each query node and edge. (2) It then sorts each list following certain ranking function. For each sorted list, a cursor is assigned at the head of the list. (3) graphTA iteratively starts an exploration based subgraph isomorphism search to expand the node match pointed by each cursor, until a complete match is identified. It moves all cursors one step forward. (4) The above step repeats until k matches are identified and it is impossible to generate better matches, or no match can be generated from the lists.

To achieve early termination, graphTA maintains a dynamically updated lower bound θ as the smallest top-k match score so far. It also maintains an upper bound to estimate the largest possible score of a complete match from unseen matches. For example, an upper bound can be established by aggregating the

Algorithm 2 Algorithm graphTA

```
Input: $G$, $Q$, integer $k$;
Output: top-k match set $Q(G, k)$;
1: initialize candidate list $L$ for each node and edge in $Q$;
2: sort each $L$ following the ranking function;
3: Set a cursor to each list; set an upper bound $U$;
4: for each cursor $c$ in each list $L$ do
5:     generate a match that contains $c$; update $Q(G, k)$;
6:     update a threshold $\theta$ with the lowest score in $Q(G, k)$;
7:     move all cursors one step ahead;
8:     update the upper bound $U$;
9:     break if k matches are identified and $\theta \geq U$;
10: end for
```

score of the next match from each list. If the upper bound is smaller than the current lower bound, graphTA terminates.

Limitation of graphTA. We use an example to demonstrate the limitations of directly applying TA-style top-k algorithm. Consider a subgraph query Q and its top-1 answer in Figure 3.2. We observe the following limitations.

(1) It is very costly to prepare the sorted lists for each node and edge. This invokes

a great amount of online computation on the match scores and the examination of the edges.

(2) Matches for nodes and edges with high matching score alone do not necessarily indicate top answers. For example, the top-1 answer is joined from a set of node and edge matches with quite low matching scores, if ranked independently (Figure 3.2). Sorted accessing over single node and edge match lists, as in graphTA, leads to an excessive amount of useless visits and enumeration of partial matches.

(3) To explore single node/edge match in a large graph often leads to expensive match expansion, resulting in significant performance degradation. For example, each time a new match is visited in a list, expanding from single node match requires a subgraph isomorphism search [169].

(4) It is often hard to estimate a tight enough upper bound, by using the node or edge matches alone. For example, if one follows sorted access to L_B to b, while all other cursors are at the top of L_A, L_C and L_D, respectively, the current upper bound, determined by 0.5, 0.9, 0.9 and 0.9, can be far from the "real" upper bound determined by 0.5, 0.5, 0.6 and 0.6. Indeed, the upper bound in conventional TA algorithm is designed for joining attribute values, where no topological linkage

is enforced. This typically generates quite loose upper bound that reduce the possibility of early termination.

3.4 Star-based Top-k Matching

Algorithm 3 Algorithm STAR

```
Input: $G$, $Q$, integer $k$;
Output: top-k match set $Q(G, k)$;
1: decompose $Q$ to a star query set $\mathcal{Q}$;
2: while top-k matches are not identified do
3:     invoke stark or stard to retrieve new top matches for queries in $\mathcal{Q}$;
4:     invoke starjoin to assemble new matches;
5:     update $Q(G, k)$;
6: end while
```

While a straightforward application of TA has the limitations in subgraph querying, we next outline a framework to mitigate it by utilizing larger structures as building blocks. The idea is to find maximal subqueries for which (a) top-k matches can be quickly retrieved without any TA-based joins, and (b) the matches of subqueries can be effectively assembled for the top-k complete matches. We identified *star shaped queries* as such structures. This framework kills two birds with one stone. First, it is observed that most of real-life subgraph queries on knowledge graphs are "star-like" queries [47, 67]. To deriving a fast solution for star queries is very appealing. Second, as a basic building block, it will lead to efficient top-k search for complex graph queries.

The top-k querying framework, denoted as STAR and illustrated in Alg. 3, has the following steps.

(1) *Query decomposition.* Given a query Q, STAR invokes a procedure to decompose Q to a set of star queries $\mathcal{Q}$ (Section 3.6.2). A star query contains a pivot node and a set of leaves as its neighbors in Q. After query decomposition, $\mathcal{Q}$ is sent to the star querying engine.

(2) *Star querying engine.* Using $\mathcal{Q}$ generated in (1) as input, a procedure, called stark (resp. stard for approximate matching) efficiently generates a set of top matches for each star query in $\mathcal{Q}$ (Section 3.5). stark guarantees that the matches are generated progressively in the descending order of the match score for each star query.

(3) *Top-k rank join.* The top matches produced by stark (or stard) from multiple star queries are collected and joined together, following the procedure starjoin, to produce top-k complete matches for Q (Section 3.6). It terminates once the top-k matches are identified, and there is no chance to generate better matches.

In the following sections we introduce the details of STAR.

3.5 Star Query

We first examine how to process star queries, the most popular query form in knowledge graphs. For simplicity's sake, we describe our algorithms only using node matches; our implementation fully supports edge matches.

3.5.1 Top-k Search

Top-k tree pattern matching have been extensively studied, *e.g.,* [55, 121] and its newest improvement [24]. Obviously, star query is a specific case. While the design of these algorithms can be reused, there are two additional problems that need special handling: (1) Most of these studies assume there is a pre-sorted node(edge) match list with respect to query node (edge), which is not true in our problem setting: $F_V(v, \phi(v))$ and $F_E(e, \phi(e))$ are computed online. We shall try to avoid a complete sorting. (2) For edge-to-path approximate graph matching, the existing studies typically require the construction of transitive closure, which is infeasible over large graphs. In the following two sections, we are going to address these two problems.

It is a well-known result as there are $O(n)$ selection algorithms finding the k-th largest number in a list. Our goal is to sort as small number of node/edge matches as possible in the course of finding the top-k star query answers.

Lemma 2: *[11] Given a set of n numbers and an integer k, finding the top-k numbers in the set is $O(n)$ and the sorted top-k numbers is $O(n + k \log k)$.* □

Proposition 3: *Given graph G, star query Q^* and k, when $|Q^*|$ and k are bounded by a small constant, the top-k matches $Q^*(G, k)$ can be computed in $O(|E|)$ time and space.* □

Algorithm 4 Algorithm stark

```
Input: $G$, $Q$, integer $k$;
Output: top-k match set $R$;
1: initializes set $R=\varnothing$;
2: initializes priority queue $P=\varnothing$;
3: find top-1 match pivoted at each node $v$ in $G$;
4: while $|R| < k$ do
5:     pop the best match $M$ (pivoted at $v$) from $P$; $R = R \cup \{M\}$;
6:     generate next best match $M'$ pivoted at $v$;
7:     insert $M'$ to $P$;
8: end while
```

Our star query processing engine stark takes the following steps (its pseudo code is provided in Alg. 4):

Step 1: Treat each node in the knowledge graph as a possible match to the pivot query node. Find the top-1 match for each of them, among which, select top-k matches to form a candidate answer pool P.

Step 2: Pop up the best match M from P, insert it into the answer set R. For the pivot node in M, generate the next best match M', insert it to P.

Step 3: Repeat Step 2 until $|R| = k$.

In the first step, stark performs pair-wise similarity calculation between query nodes/edges and data nodes/edges. This takes $|V^*||V| + |E^*||E|$ in the worst case (both time and space). [160] discussed how to reduce the cost. After that, for each node $v \in V$, stark treats it as a potential match to the pivot query node and tries to find a top-1 match pivoted at v. Given a node $v \in V$, we define *a match M pivoted at v* if v is matched to the pivot node in Q^*. stark finds the best matches for the leaf nodes of Q^* in v's neighbors and assemble them as the top-1 match pivoted at v. It scans all v's neighbor nodes, thus taking $|V^*||E|$ time. It then finds the k best matches among these matches. It takes $O(|V|)$ time (Lemma 2). Therefore, the first step takes $O(|V^*||V| + |Q^*||E|)$ time, i.e., $O(|E|)$ when $|Q^*|$ is bounded by a small constant. Its space complexity is $O(|E|)$.

Pivot Node Set V_p. Given G, Q, F, and k, let $\mathcal{M}$ be the set of top-1 matches pivoted at each node in G. Pivot node set is defined as $V_p = \{v|M$ is among top-k in $\mathcal{M}$, M is pivoted at $v\}$.

For the top-1 matches generated in Step 1, V_p is the set of nodes that match the pivot node in Q^*.

Lemma 4: *Top-k matches of* Q^* *can only come from the matches pivoted at* v, $v \in V_p$. $\square$

In Step 2, stark retrieves the top-1 match M from P with pivot node v. It then fetches the second best match pivoted at v. If v's neighbor nodes are not sorted with respect to their similarity to the leaf nodes in Q^*, one has to scan the entire list of neighbors to find the second largest value w.r.t each query leaf node. Assume the maximum degree in G is m. Since we need to find the second largest match for each query leaf node, it will take $O(m|V^*|)$. The problem becomes severe if all of the remaining top-k matches actually come from v. The cost will grow to $O(km|V^*|)$ as Step 2 will run k-1 times on v. In such a case, it is better to find top-k node matches w.r.t each query leaf node and sort them, taking time $O((m + k\log k)|V^*|)$. The following theorem shows that this result can be further improved to $O(m|V^*| + k\log k)$, which is optimal in the worst case.

Theorem 5: *Given s lists of unsorted numbers L_1, L_2, ..., L_s, and an aggregation function,*

$$F = \sum_{i=1}^{s} x_i, x_i \in L_i,$$

there will be a set $\bar{L} \subseteq L = \bigcup_i L_i$, $|\bar{L}| \leq k + s - 1$, s.t., any number in $L \setminus \bar{L}$ is not going to contribute to the top-k values of F. It takes $O(sm)$ to find $\bar{L}$. □

Proof: We first construct $\bar{L}$. Denote the largest number in L_i as x_i^{max}, $\hat{L}_i = \{x - x_i^{max} | x \in L_i\}$, $L^{max} = \{x_i^{max}\}$ and $\hat{L} = \bigcup_i \hat{L}_i$. Let $\bar{L} = \{x \in L \setminus L^{max} | x - x_i^{max}$ ranks top-k+s-1 in $\hat{L}\}$. We then prove the theorem by contradiction: Suppose $x' \in L \setminus (\bar{L} \cup L^{max})$ contributes to one of the top-k sums, denoted as F'. It is easy to see $F' \leq x' + \sum_{j \neq 1} x_j^{max} \leq x' - x_1^{max} + \sum_j x_j^{max}$, where w.l.o.g. $x' \in L_1$. However, besides the s largest numbers $\{x_i^{max}\}$, there are at least $k-1$ numbers $x_i \in \bar{L}$, such that $x' - x_1^{max} \leq x_i - x_i^{max}$. Thus F' is not among the top-k sums since there are at least k sums no less than F'. □

Theorem 5 shows that in order to find top-k results *w.r.t.* F, we need not find top-k numbers for each list L_i. Instead, with a modification, we only need to find $k + s - 1$ numbers in the union of the lists.

Example 11: Consider three lists L_B, L_C and L_D in Figure 3.3, with the largest number x_B^{max}=0.7, x_C^{max}=0.9 and x_D^{max}=0.8, respectively. To find the top-3 aggregation values *w.r.t.* function F, it only requires the largest number in each list and

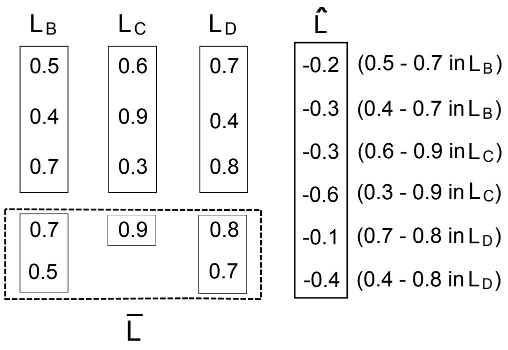


Figure 3.3: Optimal match selection.

two additional numbers. For this purpose, a set $\hat{L}$ is constructed by subtracting the largest number in each list from each other numbers in the list, *e.g.*, -0.2 in $\hat{L}$ is from L_B by $0.5 - 0.7$. The set $\bar{L}$ is then obtained by including the three largest numbers from each list and two additional numbers 0.7 and 0.5, corresponding to the top-2 numbers, -0.1 and -0.2 in $\hat{L}$, respectively. □

Following Theorem 5, given a star query $Q^* = (V^*, E^*)$ and a node v, we only need retain the $s + k - 1$ numbers from $\bigcup_i L_i$ to fetch the top-k matches pivoted at v, where $s = |V^*| - 1$.

The remaining algorithm of stark follows the concept of lattice search introduced by [55] (Actually a slightly better algorithm can be derived from [24]). It maintains a priority queue to remember the top-k matches it has found. The priority queue size is at most k. For each match, it records its pivot node and a cursor to remember the index of sorted lists. It takes $O(k \log k)$ to put the matches generated by Step 1 into the priority queue. In Step 2, when it pops up the current best match from the the queue, it retrieves the cursor. Let $s = |V^*| - 1$. Assume the cursor index is $(l_1, l_2, \dots l_s)$, it is going to calculate the F value for $(l_1 + 1, l_2, \dots l_s)$, $(l_1, l_2 + 1, \dots l_s)$, $\dots$, $(l_1, l_2, \dots l_s + 1)$. Hence, there are in total s matches, which shall be pushed into the queue if they are greater than the minimum value in the queue. The time cost is $s \log k$.

Combine all the above cost together. Step 2 takes $O(m|V^*| + k \log k + |V^*| \log k)$ time, which is iterated $k-1$ times. When k is a small constant, the time complexity is governed by $O(m|V^*|)$, where m is the maximum node degree.

Analysis. For the time complexity, Step 1 takes $O(|V^*||V| + |Q^*||E|)$ time to find best k top-1 matches. Step 2 takes $O(mk|V^*| + k^2 \log k + |V^*|k \log k)$, in total. Assuming Q^* and k bounded by a small constant, stark is linear in terms of $O(|E|)$. The above analysis completes the proof of Proposition 3. In practice, not every node in G will be matched with the query pivot node. A cutoff threshold will be applied to retain a few candidate nodes. Let n^* be the size of candidate nodes. In this case, Step 1 takes $O(n^* m|V^*|)$. When $n^* < k$, the complexity of Step 2 will dominate. The optimization of Step 2 will play a more important role. When n^* is very large, the aggregation overlay graph and a “push” strategy from [105] could be applied to enable shared computation.

3.5.2 d-bounded Star Query

As remarked earlier, an edge may be matched to a path of bounded length in knowledge graphs. Given G, Q and an integer d, the d-bounded subgraph querying extends subgraph querying by a matching function ϕ_d, such that each edge $e = (u, u')$ can be mapped to a path $\phi_d(e)$, connecting two node matches $\phi(u)$ and $\phi(u')$ with the length bounded by d. In this work, we do not consider the situation where two query nodes are matched to one node, as in knowledge graphs, each node usually represents a unique entity or concept.

Edge-Path Similarity Function. When an edge e in a star query is matched to a path $\phi_d(e)$ in G, we need to define a similarity function $F(e, \phi_d(e))$. The algorithm proposed in this section is valid as long as $F(e, \phi_h(e))$ is monotonically decreasing in terms of d. A typical example is $F(e, \phi_d(e)) = \lambda^{(h-1)}$, $\lambda \in (0, 1)$, where h is the length of path $\phi_d(e)$.

For d-bounded star querying, a straightforward method is to traverse d-hop neighborhood of a pivot node and run stark. This reduces the d-bounded star querying to its 1-bounded counterpart. Nevertheless, the bottleneck becomes the excessive cost of graph traversal. Precomputing d-hop neighborhood for each node is no longer practical for large graphs. Let $\bar{m}$ be the average degree of G. For d=2, it already visits $O(\bar{m}|E|)$ edges. Even such index is available, accessing $O(\bar{m}|E|)$ edges is not going to scale well.

The major challenge of stark is to identify the pivot node set V_p by retrieving top-1 matches from a potentially large number of matches. V_p contains pivot nodes that have best top-1 matches among all top-1 matches pivoted at $v \in V$. Following Lemma 4, the next step is to find top-k matches pivoted at nodes in V_p, where traversing is typically more affordable for a small k. We have the following result.

Proposition 6: *Given G, Q^*, k, and d, when $|Q^*|$ and k are bounded by a small constant, there exists an algorithm that finds the pivot node set V_p in $O(d^2|E|)$ time.* □

We next introduce a message propagation algorithm, stard, which achieves the above time complexity. In a nutshell, it leverages message passing to exchange the maximal node and edge matching scores.

Message propagation. The algorithm stard identifies all the node matches v for each query leaf node u^* in Q^*. Instead of "pulling" the neighbors' score for each potential pivot node match in G, it collects, aggregates and propagates messages encoding the matching score of each node in G to its 1-hop neighbors and repeats d times.

Message. stard encodes a message m as a set of triples $<(u^*, v), F, h>$. If a node receives such a message, it means in h hops, there is a node v matched to u^* with score F.

Example 12: Consider query Q^* in Figure 3.4. A message m is initialized at c_1 as $<(C, c_1), 0.9, 0>$, indicating that c_1 is matched to query node C, with node score 0.9, and the message resides at node c_1 (with hop number 0). $\square$

Message propagation. stard initializes a message m that contains a single triple $<(u^*, v), F(u^*, v), h = 0>$ at each match v of u^*. It then propagates m by forking it to multiple copies and distributing all the copies to its 1-hop neighbors at the same time. For each node v that receives a message $m_1 = <(u^*, v_1), F_1, h_1>$, it increases h_1 to $h_1 + 1$ and then performs the following aggregation task:

1. If v has no local copy of any message containing u^*, it keeps a copy of m_1.
2. If v has a local copy of a message, $m_2 = <(u^*, v_2), F_2, h_2>$. If $F_1 \leq F_2$ and $h_1 \geq h_2$, discard m_1; Otherwise keep both m_1 and m_2.

Intuitively, stard always keeps track of the node match with a greater "potential" to be the top-1 match, measured by the sum of its node score and "up to the moment" edge score $F(e)$ at the hth hop of propagation.

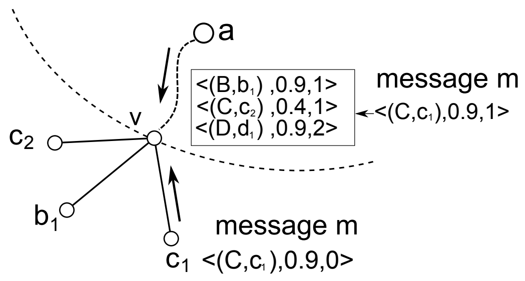


Figure 3.4: d-bounded star querying.

Example 13: Given query Q^* in Figure 3.4 for 3-bounded search, F_E is defined as 0.8^{h-1} for a path match of length h. stard iteratively propagates m from node c_1 to its neighbors. When m is propagated to node v, it finds a local copy of message m' with an entry $<(C, c_2), 0.4, 1>$, indicating that a match c_2 is 1 hop away from v. stard replaces the entry of c_2 with c_1, and continue propagation with m. □

Algorithm stard. We now give an outline of the complete algorithm stard. Given a d-bounded star query Q^*, it performs d-round message propagation from all the leaf node matches in G. After that, it selects V_p along the same line as stark. For each match in V_p, it performs a traversal to collect the distance and score information to compute top-k d-bounded matches, similar to stark.

Ping-Pong effect. There is a possibility that a node v could have a similarity score with both the pivot node and a leaf node u^* in Q^*. When a message initiated at v for matching u^* is passed around, it is possible that it arrives at v again. When

v is matched to the pivot node, it might lose the trace of any other node that could be matched to u^*. In this case, we can not derive the top-1 match pivoted at v correctly. One way to solve this problem is to record two best matches for u^* and pass them around. This will guarantee at least one match can be used later.

Analysis. Once the message propagation terminates, the algorithm stard correctly computes top-k matches for Q^*, following stark. Hence it suffices to show that all the top-1 matches are correctly gathered and computed. Indeed, stard keeps the invariant below: (1) at any time during message propagation, the message which carries the information of top-1 node and edge matches are not replaced by any other message; and (2) when the propagation terminates, all the message in (1) are guaranteed to be fetched. The correctness of stard hence follows.

For the time complexity, the main time cost of stard is dominated by the message passing. There are at most d rounds of message propagation for every node. For each node in G, we need to maintain at most $d|V^*|$ messages. Hence the total time is in $O(d^2|E||V^*|)$ for finding the pivot node set V_p. Once it is found, the time to find top-k matches is in $O(m^d|V^*| + k^2 \log k + |V^*|k \log k)$. The space complexity is $d|V^*||V|$. When $|Q^*|$ and k are bounded by a small constant, the total time complexity is $O(d^2|E| + m^d)$. The above analysis completes the proof of Proposition 6. The most recent work [24] on the top-k tree matching in graphs proposes an optimal solution that runs in $O(m_R + k(n_T + \log k))$, where n_T is the node number of the tree and m_R is the edge number of a runtime graph that is extracted from a transitive closure of the data graph. Note that the average degree of knowledge graph could be large, $\bar{m} > 30$ v.s. $\bar{m} = 2 \sim 3$ in [24]. This may lead to a huge runtime graph, making m_R prohibitively large.

The implementation of stard allows multi-level of parallelism. In vertex-centric programming [94], each node can exchange messages between their neighbors in parallel. All message propagation can be done in at most d rounds.

3.6 Top-k Star Join

The star query processing engine stark can not only process star queries quickly, but also serve as a foundation to answer general graph queries. A graph query Q can be decomposed to a set of star-shaped queries $\{Q^*\}$. Top-k answers to Q can be assembled by collecting the top matches of each Q^*, followed by a multi-way join process.

There is a great advantage of leveraging star queries. First, stark is able to quickly generate matches in a monotonic decreasing order of the matching score. As manifested in Section 3.6.1, this property is critical when joining multiple subqueries: It produces an upper bound for those matches that have not been seen yet. Second, although a similar method [121] exists for other basic structures like edges, a bigger structure like star-shaped subqueries can reduce the number of joins, thus improving query processing time. There are two challenges for star match assembling:

1. Query decomposition. Consider different query decomposition strategies and determine an efficient way to execute a query.

2. Top-k rank join. Efficiently construct the join matches from star matches and derive an upper bound for the remaining possible matches.

We first investigate the top-k rank join problem and then develop the intuition that can be applied to query optimization in Section 3.6.2.

3.6.1 Top-k Star Rank Join

Given a query Q decomposed to a set of star queries $\mathcal{Q} = \{Q_1^*, Q_2^*, \dots Q_m^*\}$, the *rank join* is to find the top-k matches for Q by assembling the matches retrieved by stark on each Q_i^*. This is outlined as starjoin in Alg. 5.

starjoin performs in a similar way as the hash rank join strategy (*HRJN* [69]). It iteratively fetches k matches for each star and joins them with the existing matches for the other stars (line 5 and 6). In order to compute the joins, a hash table for each L_i maintains the mapping of the joint nodes to the matches seen so

Algorithm 5 Algorithm starjoin

```
Input: $\mathcal{Q} = \{Q_1^*, Q_2^*, \ldots Q_m^*\}$;
Output: top-k join matches;
1: while $\mathcal{Q} \neq \varnothing$ do
2:     for each $Q_i^* \in \mathcal{Q}$ do
3:         invoke stark on $Q_i^*$ to find the next match $M$;
4:         join $M$ with $L_j$ $(j \neq i)$ and add the join results to $R$;
5:         update $\theta$ as the k-th score in $R$ if $|R| \geq k$;
6:         compute upper bound $\theta_i$ based on $M$;
7:         add $M$ to $L_i$; remove $Q_i^*$ from $\mathcal{Q}$ if $\theta_i < \theta$;
8:     end for
9: end while
10: return the first k results in $R$;
```

far. starjoin keeps track of lower bound θ as the k-th match in the priority queue R (line 7). It can be seen that the efficiency of the algorithm relies on the upper bound θ_i for each star (line 8 and 9).

Upper bound [69]. Consider m match lists $\{L_1, \ldots, L_m\}$. For a list L_i of size n_i, denote ϕ_{ij} as the jth ranked match in L_i. The upper bound θ_i is defined as

$$\theta_i = F(\phi_{in_i}) + \sum_{j=1, j\neq i}^{m} F(\phi_{j1}). \tag{3.2}$$

Intuitively, an upper bound is estimated as the sum of the scores from the last match in one list and the top-1 matches from all the others.

The *HRJN* strategy was widely adopted in RDBMS and demonstrated the superior performance over the traditional join-then-sort approach [69]. However, there is a difference between *HRJN* and starjoin. Directly applying θ_i as Eq. 2 results in an invalid upper bound, as the scores for the joint nodes shared by several stars are counted multiple times. This can be seen as the example shown in Figure 3.5(a). Given a query Q and the score function F, let $(A = a_n, U = u_n)$ and $(B = b_1, U = u_1)$ be the n-th match and the first match in L_1 and L_2, respectively. According to Eq. 2, $\theta_1 = F(a_n) + F(u_n) + F(u_1) + F(b_1)$, which cannot be considered as the upper bound and directly compared with the lower bound θ for the top-k join results. To overcome this problem, we introduce the starjoin with the α-*scheme*.

Rank Join with α-scheme. Let U be the set of the joint nodes for two stars Q_1^* and Q_2^*, and A (resp. B) is the set of nodes that appear only in query Q_1^* (resp. Q_2^*). Then based on a parameter α, we introduce a new ranking function scheme, denoted as $F'(\phi(Q_1^*)) = F(\phi(A)) + \alpha \cdot F(\phi(U))$ for Q_1^* and $F'(\phi(Q_2^*)) = F(\phi(B)) + (1 - \alpha) \cdot F(\phi(U))$ for Q_2^*. Accordingly, given the two match lists, L_1 for Q_1^* and L_2 for Q_2^*, the upper bound can be refined as

$$\theta'_1 = F'(\phi_{1n_1}) + F'(\phi_{21}), \theta'_2 = F'(\phi_{11}) + F'(\phi_{2n_2}), \tag{3.3}$$

where ϕ_{1n_1} and ϕ_{21} are the last match and top match in L_1 and L_2, ϕ_{11} and ϕ_{2n_2} are the top match and last match in L_1 and L_2, respectively. When $\alpha \in [0, 1]$, one may verify that θ'_1 and θ'_2 are valid upper bound for the search on Q^*_1 and Q^*_2, respectively. It is worth mentioning that the selection of α affects the number of matches to be fetched for assembling.

Example 14: Given query Q in Figure 3.5(a) that is decomposed to two stars Q^*_1 and Q^*_2. Denote $a_i(j)$ in the figure as the i-th largest entry in the match list for A with the match score j. For example in L_1 in Figure 3.5(c), $a_2(0.9)$ in the third entry refers to the match a_2 for A with score 0.9 and $u_1(0.5)$ refers to the match u_1 for U with score $1.0 * \alpha = 0.5$. To identify the top-4 join matches as in Figure 3.5(b), it only needs to reach the top-3 matches in L_1 and L_2 with $\alpha = 0.5$. While for $\alpha = 0.9$, at least top-3 and top-11 matches in L_1 and L_2, respectively, are required. $\square$

The effectiveness of starjoin can be evaluated by the *total search depth*, $D = \sum_i |L_i|$, when the algorithm terminates. Example 14 implicates that when using a proper α, starjoin will likely require a smaller D to identify the top-k join matches, *e.g.,* $D = 6$ (resp. $D = 14$) when $\alpha = 0.5$ (resp. $\alpha = 0.9$) in the example. To determine an optimal α value for minimizing D, nevertheless, is not trivial. We introduce a principled way to determine α in Section 3.6.3.

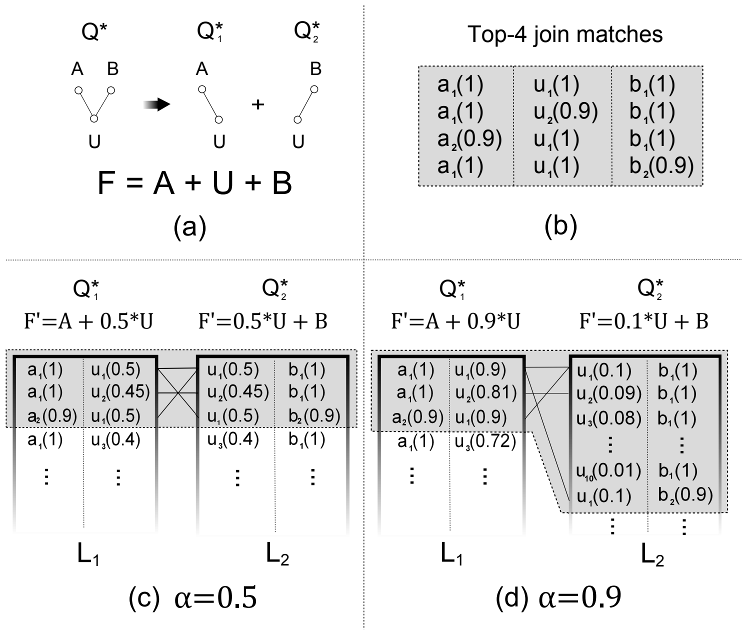


Figure 3.5: Selection of α value.

The α scheme works for assembling two star matches, *i.e.,* two-way join. For multiple stars, we perform a sequence of two-way join (as a left-deep pipeline [69]) and apply the α scheme for each two-way join.

3.6.2 Query Decomposition

We next discuss the query decomposition problem, which has been studied for solving complex queries, *e.g.,* twig queries on XML data [121,141] and SPARQL on RDFs [67]. However, the traditional techniques are not applicable in our problem setting since the match score has to be calculated online.

Given a query graph, we expect a decomposition to generate a set of star subqueries that minimize the total depth D. Since all match scores are generated on the fly, it is very challenging to analyze the search depth accurately. We investigate several heuristics and evaluate their performance on real-world graphs.

First, a reasonable decomposition derives as small number of stars as possible, which intuitively reduces the number of join operations. Second, to lower down the upper bound in Eq. 3.2 (Section 3.6.1), we shall make $F(\phi_{in_i})$ as small as possible. Therefore, a large score decrement for the matches in L_i will likely lead to small search depth. Third, we observe that many real-world star queries share the similar distribution of the match scores with a long-tail effect, as illustrated in Figure 3.6. Given a query decomposed to several stars, the search for each star that stops at similar positions, say n_b, is likely to yield smaller D, in comparison with the case that one star search stops at n_a while the others stop at n_c with a much larger position gap. Based on these observations, the third intuition is to decompose a query to a few stars that have similar distribution of matching scores. While it is hard to derive the actual distribution, we approximately characterize it with similar size, similar top-1 match score or similar match score decrement.

Based on the above intuitions, given Q, the objective of the query decomposition is to derive a minimum number of stars with similar features, such that the score decrement of the matches for each star Q^*_i can be maximized. This can be

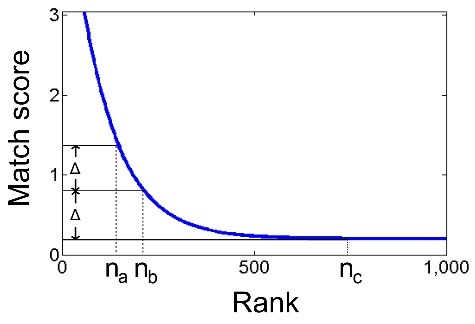


Figure 3.6: The distribution of the top-k matching score.

described as an optimization problem,

$$\underset{\{Q_1^*,\ldots,Q_m^*\}}{\text{maximize}} \qquad \sum_{i=1}^{m} \delta(Q_i^*) - \lambda \sum_{i=1}^{m} |f(Q_i^*) - \bar{f}| \qquad (3.4)$$

$$\text{subject to} \qquad \text{minimum m}, \qquad (3.5)$$

where $\delta(Q_i^*)$ is the score decrement of the top matches in L_i, $f(Q_i^*)$ is the feature score of Q_i^* while $\bar{f}$ is their average, *i.e.*, $\frac{1}{m}\sum_{i=1}^{m} f(Q_i^*)$. Intuitively, it aims to maximize the score decrement and minimize the feature difference of the subqueries, where λ is a parameter to make a trade-off.

Since it is costly to accurately compute the score decrement δ and exhaust all the feature measurements, we consider several simple but effective features below:

SimSize: $f(Q_i^*) = |E_i^*|$: Star size.

SimTop: $f(Q_i^*) = F(\phi_{i1})$, where ϕ_{i1} is the top-1 match for Q_i^*. Unfortunately, ϕ_{i1} is difficult to observe without executing Q_i^*. Hence we use the top-1 pivot node match score to represent $F(\phi_{i1})$. In practice, we sample nodes in knowledge graphs and calculate the match score.

SimDec: $\delta(Q_i^*) = f(Q_i^*) = \frac{F(\phi_{i1})-F(\phi_{in_i})}{n_i}$, where n_i is the number of top matches checked for Q_i^*. SimDec measures the average match score decrement for Q_i^*. In practice, we approximate n_i by $p^{|V_i^*|-1}\prod_{v\in V_i^*} n_v$, where n_v is the number of node matches and p is the probability that two node matches are connected. p is a parameter estimated off-line by conducting a set of edge queries. n_v is estimated by sampling nodes in knowledge graphs calculating their match score with the pivot node of Q_i^*, and selecting relevant ones.

Query decomposition based on SimSize only considers query structures. Nevertheless, such problems (balanced edge partition) are in general hard (NP-hard) [132]. We employ the efficient greedy algorithm designed in [132] for SimSize. In practice, since most queries would not have many star subqueries, we use dynamic programming to enumerate possible star decompositions starting with $m = 2$. For each m, the decomposition with the best score of Eq. 5 will be picked and returned immediately.

3.6.3 Optimization: Determine the Parameters

The above top-k rank join technique has two parameters, α and λ, which can be learned off-line by a testing and validation method. Suppose we have a sample query workload W. Our top-k join algorithm is assumed as a black-box A with three input α, λ and W. The output of A is the aggregated total depth D for the queries in W. Let $\alpha \in [0, 1.0]$ and $\lambda \in [0, 2.0]$. By iteratively running A and setting a small constant *e.g.,* 0.1 as the adjustment step for α and λ, we can derive an optimal setting of α and λ that minimizes D. As verified in Section 3.7, with proper α and λ, our query optimization technique can achieve up to 45% runtime improvement over the baseline algorithms.

STAR is currently memory-based; it can be conveniently adapted to existing distributed memory-based platforms, such as Spark [159], where a single machine is not capable to handle a large-scale graph. stark can be executed in individual machines managed by Spark, and starjoin can be performed in the master node. A distributed implementation is under development.

3.7 Experimental Evaluation

We conduct a set of experiments, using real-world knowledge graphs to examine the performance of STAR and its components including the star query engine, stark/stard, and the top-k rank join, starjoin.

3.7.1 Experimental Setup

Datasets. We employ the same set of knowledge graphs as introduced in Section 2.8 in the following evaluation.

Query workload. Two sets of query workload are designed for the evaluation. (1) We adopt the *DBPSB* benchmark [106] and derive 50 star query templates. Each template contains a set of nodes and edges which are augmented by either the real labels, e.g., '*Person*', or a variable label '*?*'. The percent of the variable label is $\leq 50\%$ in each template. The variable node (resp. edge) in a template query can be matched to any node (resp. edge) in the graph. To generate a query, we search the template in the graphs and select the most common labels from the data entity that are matched to the variables. The selected labels are then used to instantiate the variable nodes/edges in the template. (2) Since the templates are only stars, we extend the templates by adding nodes and edges to generate queries with cycles or multiple star structures. Figure 3.1 shows a sample query.

Algorithms. We implement the STAR framework, including algorithms stark, stard and starjoin. In order to run stark for d-bounded star queries, d-hop traversal is performed for each node match of the pivot node. For comparison, we also developed two top-k search algorithms, graphTA and BP.

(1) The algorithm graphTA (Section 3.3) is a direct application of the threshold (TA) algorithm over top-k subgraph querying [169]. For a fair comparison, we implement graphTA with two optimizations. (a) The neighbors and their matching scores are cached in each node when the node is visited during the traversal. The cache serves as an index to reduce the unnecessary graph traversal when the node is visited again; (b) Instead of using the widely adopted DFS traversal, it adopts BFS traversal so that the neighbor nodes are sorted based on their scores before carrying out the next round of exploration. These two strategies reduce the runtime of graphTA by 90%.

(2) The label propagation based algorithm [79, 160], such as Belief Propagation (BP), was also employed recently for approximate top-k pattern matching. BP [160] considers the nodes/edges in a query as a set of random variables and converts the top-k matching problem to the probabilistic inference on the label (match) for each random variable. It finds approximate matches for cyclic patterns, by exchanging probability scores as messages among node matches. For

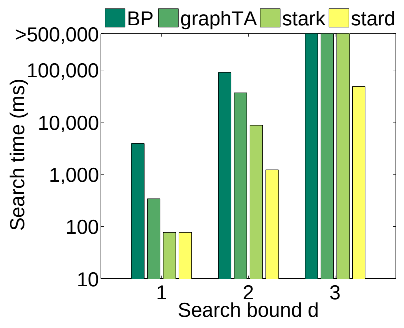
acyclic queries, BP outputs the exact top-k matches. But for cyclic queries, different from the STAR framework, it does not guarantee the completeness. We did not employ the graph sketch technique developed in [160] as it can benefit all the search algorithms.

Metrics. Given the query workload, the search runtime corresponds to the end-to-end query processing time, *i.e.,* the total CPU time spent from receiving the query to the output of the top-k results. The time includes not only the cost of the top-k search time but also the cost of other tasks, such as node matching and query decomposition, which account for a small amount of runtime ($\leq 1\%$).

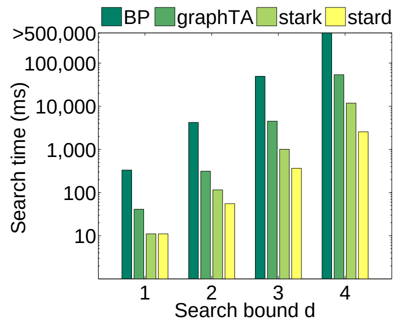
Setup. All the algorithms are implemented in Java. We conduct the experiments on a server with Intel Core i7 2.8GHz CPU and 32GB RAM, running 64-bit Linux. To serve online queries, the graph is stored in main memory while the rich information attached to the nodes/edges is stored in a MongoDB server on a 512GB SSD. Each result reported in the following is averaged over 5 cold runs.

3.7.2 Evaluation Results

Exp-1: Runtime over star queries. In this experiment, we examine the impact of the search bound d. We employ a query workload consisting of $1,000$ star queries which are randomly generated based on the query templates with



(a) Varying d: DBpedia



(b) Varying d: YAGO2

Figure 3.7: The effect of search bound d.

different size. By fixing k=20 and varying d, we compare the performance of stark, stard, graphTA and BP.

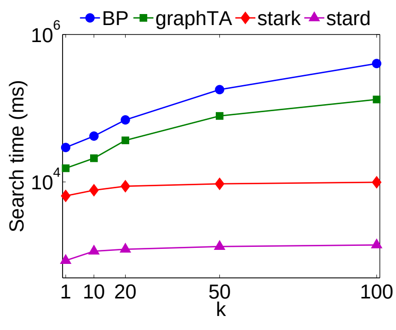
The results over *DBpedia* and *YAGO2* are reported in Figure 3.7(a) and (b) respectively, in log scale. The result shows that stark and stard outperform BP and graphTA by almost one order of magnitude. Note that when $d = 1$, stard degrades to stark, thus having the same runtime. The results also demonstrate superb performance of stard when $d \geq 2$. Indeed, for large d, BP, graphTA and stark may incur a humongous amount of message passing and neighborhood exploration, which can be reduced by stard.

Exp-2: Impact of k and query size. In this set of experiments, we evaluate the impact of k and query size to the runtime. Fixing $d = 2$ and use the same query workload as in the previous experiment, we vary k from 1 to 100. The results are plotted in Figure 3.8(a-b), which shows that the runtime of BP and graphTA grows dramatically when k increases. Indeed, both BP and graphTA use top scored node matches to find complete matches, which incurs considerable useless enumeration and traversal, especially for larger k. The top scored node matches might not lead to the best matches of the query. In contrast, stark and stard outperform all other methods in orders of magnitude, and their performance is much less sensitive to the growth of k. We observe that the main bottleneck for stark is the expensive graph traversal, especially for larger d and denser graphs (*DBpedia*). stard copes with this quite well: Almost all results are acquired in 1 second.

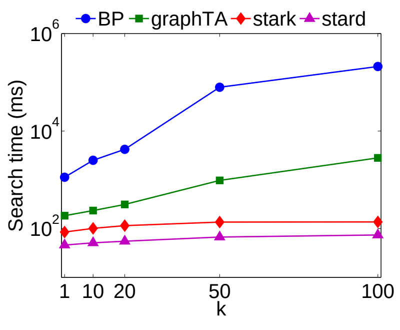
To evaluate the impact of query size, we use star query templates with different numbers of nodes varying from 2 to 6. We generate 5 query workloads accordingly, each contains $1,000$ instantiated queries. We fix d=2 and k=20. Figures 3.8(c-d) show the exponential runtime growth of BP and graphTA, while stark and stard are less sensitive. stark (resp. stard) improves BP and graphTA better over larger queries, and is twice (resp. 8 times) faster than graphTA for even single edge query with 2 nodes.

We conduct the above experiments on more complicated graph queries and had very similar observations. The reason is obvious. Since stark and stard optimize the search based on bigger structures (star vs. single node/edge), their search will have a lower chance to be stuck in local optimum.

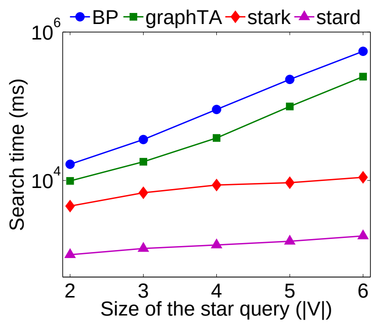
Exp-3: Efficiency of top-k join. This experiment examines the proposed top-k rank join technique. The three query decomposition methods, *i.e.,* SimSize,



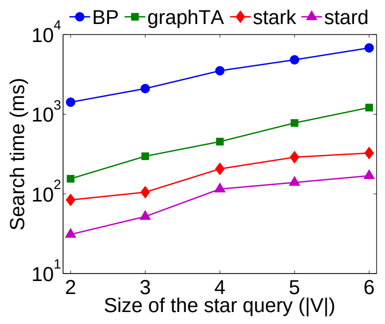
(a) Varying k: DBpedia



(b) Varying k: YAGO2



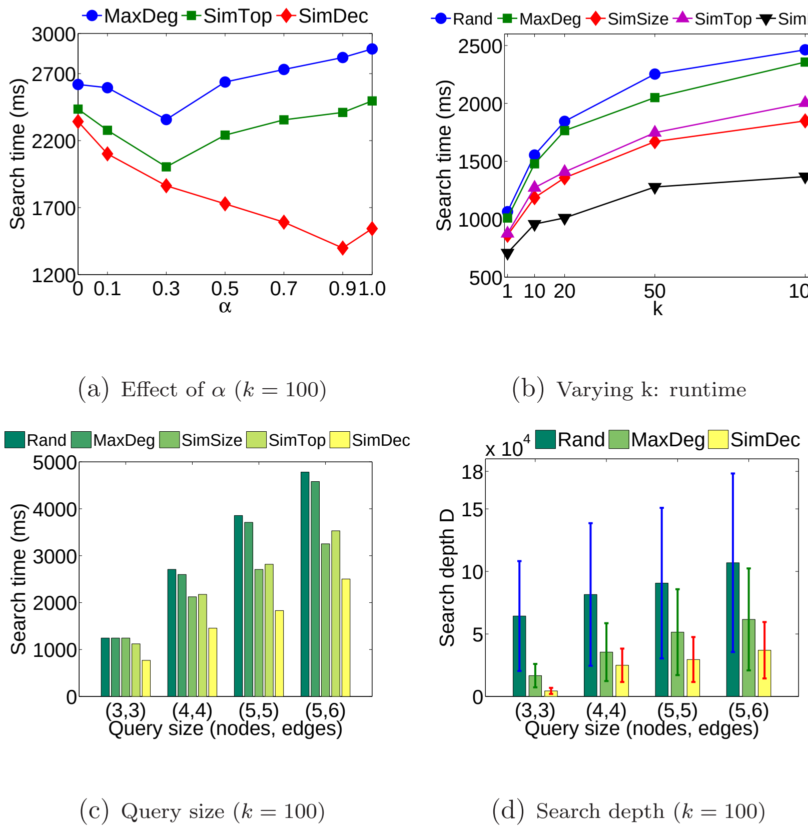
(c) Varying query size: DBpedia



(d) Varying query size: YAGO2

Figure 3.8: Efficiency of star querying.

SimTop and SimDec, are inspected, respectively. The node score variance in SimTop and SimDec is estimated online by randomly sampling 200 matches for each query node. The sampling time only accounts for $\leq 1\%$ of the total search time and hence is not reported separately. In SimDec, $p = 4.5 \times 10^{-4}$, estimated by checking a set of edge queries. Additionally, two baselines are compared: (1) Rand refers to a method that randomly selects the pivot nodes to generate star



(a) Effect of α ($k = 100$)

(b) Varying k: runtime

(c) Query size ($k = 100$)

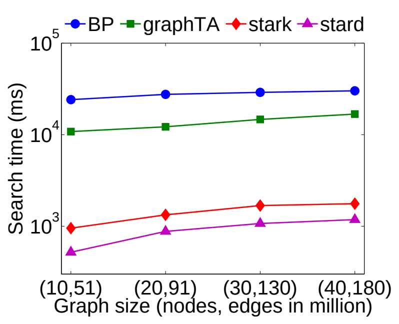
(d) Search depth ($k = 100$)

Figure 3.9: Evaluation on the top-k join.

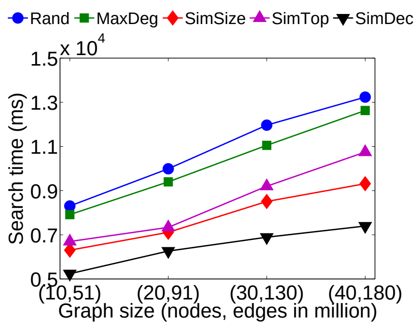
subqueries; (2) MaxDeg greedily selects the pivots with the highest degree in the query graph.

We first test the effect of the α-schema. A query workload is generated using randomly selected query templates. We choose k=100 and d=1 in the experiment. Figure 3.9(a) depicts the average search time by varying α. It shows that a well selected α value indeed leads to less runtime. Considering each method, the best performance can be achieved when $\alpha = 0.3$ for MaxDeg, $\alpha = 0.3$ for SimTop and $\alpha = 0.9$ for SimDec ($\lambda = 1.0$), respectively. These α values are used in the following tests. We choose $\alpha = 0.5$ for Rand and SimSize due to their random and symmetrical nature (verified by real test). We also evaluate each method by varying k and plot the time efficiency in Figure 3.9(b). The result tells when k increases, the search time increases accordingly. Moreover, SimSize, SimTop and SimDec demonstrate constantly better runtime performance than Rand and MaxDeg for each k setting. Among all the methods, SimDec performs best, saving up to 45% *w.r.t.* Rand in terms of search time.

The experiment in Figure 3.9(c) examines top-k join by the query workloads with different query size, ranging from $Q(3,3)$ to $Q(5,6)$. We observe when the query size increases, the runtime increases for all the methods. This is because a larger query is usually decomposed into more stars, leading to more expensive multi-way joins. In the figure, SimDec shows the best time efficiency compared with the others. Moreover, the top-k join incurs large search depth for each star subquery. This effect can be seen in Figure 3.9(d), which reports the average search depth. Among all the methods, SimDec results in the smallest search depth for each query workload. Figure 3.9(d) also shows the average standard deviation as the error bar for each workload. When serving a query, small depth deviation indicates similar search depth for each star subquery, leading to a balanced search effort. As shown in the figure, the heuristic employed in SimDec is quite effective, showing the smallest deviation. Note that this balance merit might have significant impact on distributed graph query processing and thus is worth investigating in the future.



(a) Efficiency of star query



(b) Efficiency of top-k join

Figure 3.10: Scalability evaluation of top-k search on Freebase.

Exp-5: Scalability. This experiment studies the scalability of the algorithms over *Freebase*. Specifically, we extract a graph $G_1(10M, 51M)$, *i.e.,* 10M nodes and 51M edges, from Freebase and expand it in a BFS manner (each time randomly pick up a node and add the new edge from Freebase) to three larger graphs $G_2(20M, 91M)$, $G_3(30M, 130M)$ and $G_4(40M, 180M)$. We use a query workload with $1,000$ randomly generated queries and fixed $k = 20$ and $d = 2$. Since k is fixed, the runtime might not increase linearly w.r.t the graph size. Figure 3.10(a) reports the result of top-k star querying in log scale. When the graph size increases, the runtime of all the algorithms increases, as expected. stark and stard outperform their competitors by at least one order of magnitude. Moreover, stard further improve stark by $35\% - 45\%$.

We also verify the scalability of starjoin and report the result in Figure 3.10(b). Using the α schema, the proposed decomposition techniques, SimSize, SimTop and SimDec, are 20% $-$ 44% faster than the baselines Rand and MaxDeg. This again demonstrates the effectiveness of the α-scheme and the decomposition heuristics.

3.8 Related Work

The top-k search has been studied extensively in various contexts, including relational data, XML and graph.

Relational data. Top-k search over relational data is to find top-k tuples for a scoring function [70]. Given a monotonic aggregation function, and a sorted list for each attribute, Fagin's algorithm [40] reads the attribute values from the lists and constructs tuples with the attributes. It stops when k tuples are found from the top-ranked attributes that have been seen. It then performs random access to find missing scores. The algorithm is optimal with high probability for some monotonic scoring functions. The threshold algorithm [41] improves Fagin's algorithm in that it is optimal for all monotonic scoring functions, and allows early termination. In a nutshell, it reads the scores of a tuple from the lists and performs sorted access to tuples by predicting maximum possible score in the unseen ones, until top-k tuples are identified. Besides simple aggregation, selection queries are studied in [17] following a similar idea.

Ranked join queries are studied over relational data [69, 109, 158, 168]. Assuming that random access is not available, J^* search [109] tries to identify a combination of attributes at the top of priority queues by selecting the stream to be joined, and pulls the next tuple from the selected stream. Ranked join queries are also studied in NoSQL databases [112], which leverage indices and MapReduce optimization, as well as statistical structures (histogram and bloom filter) to reduce the cost and identify promising values. Distance join index is proposed in [168] to find matches with static scores for graph patterns, where edges can be matched to paths. A recent work [123] introduces the hybrid indexing on weighted attribute graphs. The indexing considers the weights of the attributes on the nodes as well as the structure of the graph.

In this work we study top-k queries on knowledge graphs. (1) We do not assume static node/edge weights; instead, the matching scores are computed online. (2) We study a general graph matching problem, where the matching quality is determined by scores from nodes and edges, and edges can be matched to paths of bounded length.

Keyword search. Keyword search in XML finds top-k subtrees of a XML documents that contain all the required keywords, instead of a subgraph that matches a pattern query [56, 59]. When XPath and XQuery are considered, it is to find top ranked matches for tree patterns in terms of keywords and IR metrics, *e.g.,* TFIDF [101].

Twig query. In a more general setting, top-k graph pattern matching for twig queries are studied [24, 55, 121]. A bottom-up strategy is studied [55] where sorted access is used to generate matches for the leaf nodes in the twig query, and top matches for subqueries are obtained by merging top matches from their leaf nodes. [121] studies top-k graph pattern matching when strict monotonicity may not hold for some twig queries. These studies typically require sorted node/edge matches and the construction of transitive closure for the data graph, which are expensive over large graphs. In contrast, our method does not require transitive closure and is able to perform top-k join using partial matching lists generated online.

Graph query. Top-k search for general graph queries was studied [29,38,141,150, 163,169]. The common practice in these studies for early termination is, in general, conservative TA-style test, while combined with scheduling approaches and tuning elements. They often follow several steps as follows: (1) Fetch matching lists for nodes and edges; (2) iteratively perform sorted access over a selected list, and expand a partial match by joining node and edge matches from other list, and (3) update top-k matches with seen matches. A threshold derived from seen matches and an upper bound estimation from unseen matches are adjusted, to dynamically determine a termination condition. A closely related method is [29], which uses multiple match lists of spanning trees from a pattern to answer top-k graph pattern matching. Instead of accessing node/edge matches in the list, we resort to big structure – star subquery, which can be solved in a very efficient manner. This is not addressed in [29]. Furthermore, our decomposition technique identifies promising stars as the subqueries when serving the general graph query.

Another related work is the best-effort algorithm [143]. It returns k matches based on heuristic rules, *i.e.,* first finding the most promising match vertex v and then extending it to a complete match for the remaining nodes and edges. However, this method do not guarantee that the k discovered matches are the best ones over all matches.

3.9 Summary

We developed STAR, a top-k subgraph pattern matching framework over knowledge graphs. We have shown that STAR can efficiently solve popular star queries posed on knowledge graphs. It can also be conveniently exploited for large graph queries with cycles, by incorporating a top-k rank join algorithm and an upper bound scheme. STAR does not require any pre-built index. Experimental results show that STAR is 5-10 times faster than the state-of-the-art TA-style subgraph matching algorithm.

Chapter 4

Ontology-based Graph Querying

Traditional subgraph querying based on subgraph isomorphism requires identical label matching, which is often too restrictive to capture the matches that are semantically close to the query graphs. This chapter extends subgraph querying to identify semantically related matches by leveraging ontology information. (1) We introduce the *ontology-based subgraph querying*, which revises subgraph isomorphism by mapping a query to semantically related subgraphs in terms of a given ontology graph. We introduce a metric to measure the similarity of the matches. Based on the metric, we introduce an optimization problem to find top K best matches. (2) We provide a filtering-and-verification framework to identify (top-K) matches for ontology-based subgraph queries. The framework efficiently extracts a small subgraph of the data graph from an *ontology index*, and further computes the matches by only accessing the extracted subgraph. (3) In addition, we show that the ontology index can be efficiently updated upon the changes to the data graphs, enabling the framework to cope with dynamic data graphs. (4) We experimentally verify the effectiveness and efficiency of our framework using both synthetic and real-life graphs, comparing with traditional subgraph querying methods.

4.1 Introduction

Traditional subgraph querying adopts identical label matching, where a query node in Q can only be mapped to a node in G with the same label. This is, however, an overkill in identifying matches with similar interpretations to the query in some domain of interest. In such matches, a query node may correspond to a data node in G which is *semantically related*, instead of a node with an identical label. The need for this is evident in querying social networks [30], biological networks [146] and semantic Web, among others.

Example 15: Consider the graph G shown in Figure 4.1 which depicts a fraction of a social travel network [5]. Each node represents an entity of types such as tourist groups (`Holiday Tours (HT)`, `Culture Tours (CT)`), attractions (`Disneyland`, `Royal Gallery (RG)`), leisure centers (`Holiday Plaza (HP)`, `Royal Palace (RP)`), or restaurants (`Holiday Cafe (HC)`, `riverside`); and each edge represents a relation between two entities, *e.g.,* "has guides for" (`guide`), or "recommend" (`recom`).

Consider a query Q given in Figure 4.1 from a tourist. It is to find some other tourists who (1) recommend museum tours with guide services, and (2) favor a restaurant named "moonlight," which in turn is close to the museum. Traditional subgraph isomorphism cannot identify any match for Q in G with identical labels. Indeed, there is no node in G with the same, or even textually similar labels for the labels in Q. However, there are data nodes in G which are *semantically close* to the query nodes, and thus should be considered as potential matches. For example, node `Royal Gallery` in G is intuitively a kind of `museum` in Q. Nevertheless, it is also difficult to determine their closeness by using Q and G alone. □

The above example illustrates the need to identify node matches that are close to the query nodes, rather than those with identical or similar labels. Several extensions for subgraph isomorphism have been proposed to identify matches with node similarity [31, 43, 148], while assuming as input the similarity information between query nodes and data nodes. However, as observed in [33], users may not have the full knowledge to provide such information.

To this end, we need to understand the semantic relationships among the query nodes and data nodes, *i.e.,* given the label of a query node, which labels are semantically close to the label, in terms of standard description of entities.

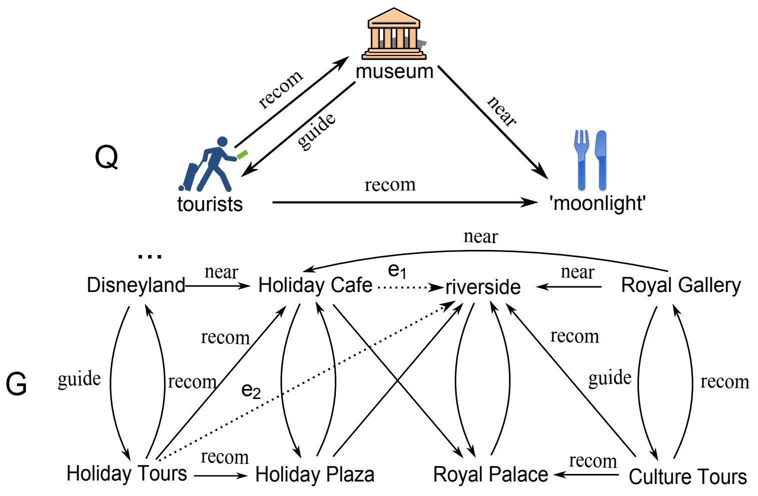


Figure 4.1: Searching a graph for traveling.

This is possible given the emerging development of *ontology graphs* [27, 37, 146]. An ontology graph typically consists of (1) a set of concepts or entities, and (2) a set of semantic relationships among the nodes. The ontology graphs may benefit the subgraph query evaluation by providing additional information about the relationships and similarity among the entities. Consider the following example.

Example 16: Figure 4.2 illustrates a travel ontology graph O_g [30] provided by a travel social network service, which illustrates the relationships between the entities in G (Figure 4.1). According to O_g, (a) `RG` is a kind of `Museum`, while `Disneyland` is not, (b) `riverside` and `moonlight` refer to the same restaurant in O_g, while `HC` is a different restaurant, and (c) `CT` and `HT` are both close to the term `tourists`. Given this, the subgraph G' of G given in Figure 4.2 should be a match close to Q. Indeed, each edge of Q (*e.g.,* (`museum`, `tourist`)) can be mapped to an edge of G' with highly related nodes (*e.g.,* (`Royal Gallery`, `Cultural Tour Community`)). earlier.

On the other hand, consider the subgraph G'' (not shown) induced by `Disneyland`, `Holiday Cafe` and `Holiday tours`. Although its three nodes are related with `Museum`, `tourists` and `moonlight`, respectively, they are not as close as the nodes in G' according to O_g. For example, `Disneyland` is more similar to the term `Park` than `Museum`. Thus, G' should be considered as a better match for Q, according to O_g. □

The ontology information has been used in *e.g.,* keyword searching [65], semantic queries [33, 37, 80], and social networks [30]. Nevertheless, little is known on how to exploit ontology graphs for effective subgraph querying. Moreover, it is important to develop efficient query evaluation techniques, especially when a query may have multiple "interpretations" and matches in terms of ontology-based similarity [65].

Contributions. We develop query evaluation techniques to efficiently identify matches that are close to a given query graph, by exploiting the ontology graphs.

(1) We propose *ontology-based subgraph querying* in Section 4.2. (a) Given a data

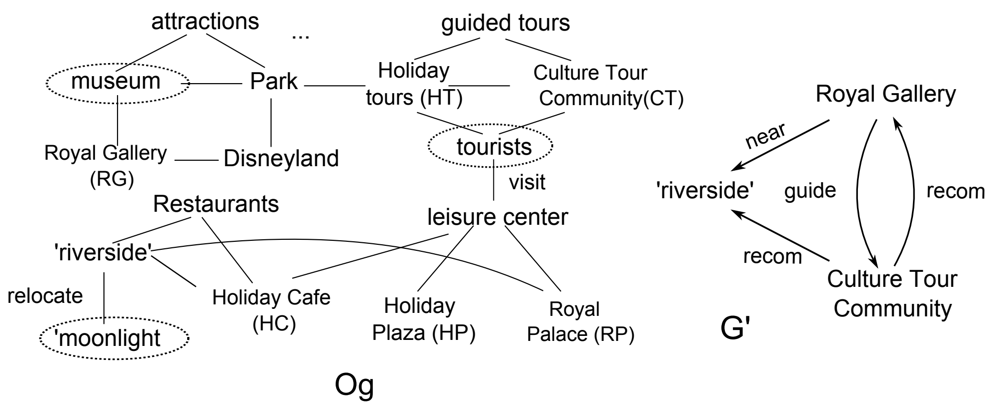


Figure 4.2: Ontology-based matching.

graph G, a query graph Q, and an ontology graph O which provides the semantic relationships among different ontologies, the ontology-based subgraph querying is to identify the matches for Q in G, where the nodes in the matches and the query are semantically close according to O. In contrast to subgraph isomorphism and its extensions, ontology-based subgraph querying measures the similarity of the nodes by exploiting the ontology graphs. (b) We introduce a metric to rank the matches of Q, based on the overall similarity of the labels between the query nodes and their matches, in terms of the ontology graphs. The metric gives rise to the the *top K matches problem*, which is to identify the K closest matches of Q in G.

(2) Based on the metric, we propose a filtering-and-verification framework for

computing top-K matches (Section 4.4). (a) We introduce an *ontology index* based on a set of *concept graphs*, which are abstractions of the data graph G *w.r.t.* O. We show that the index can be constructed in quadratic time, by providing such an algorithm. (b) Using the index, we develop a filtering strategy, which extracts a small subgraph of G as a compact representation of the query results, in quadratic time. The time complexity is determined *only* by the size of the index and the query, rather than the size of entire G. (c) We provide a query evaluation algorithm (Section 4.3) to compute the (top-K) matches following the filtering-and-verification strategy, which computes matches directly from the extracted subgraph without searching G.

(3) We experimentally verify the effectiveness and efficiency of our querying algorithms, using real-life data and synthetic data. We find that the ontology-based subgraph querying can identify much more meaningful matches than traditional subgraph querying methods. Our query evaluation framework is efficient, and scales well with the size of the data graphs, queries and ontology graphs. For example, our evaluation algorithm only takes up to 22% of the running time of a traditional subgraph querying method over graphs with 7.7M nodes and edges. Moreover, the construction and incremental maintenance of the index is efficient. The incremental algorithm outperforms the batch computation, and only takes

up to 20% of the running time of batch computation in our tests. We contend that the framework serves as a promising method for subgraph querying using ontology graphs in practice.

4.2 Ontology-based Subgraph Querying

Below we introduce data graphs and query graphs, as well as the ontology graphs. We then introduce the notion of the ontology-based subgraph querying.

4.2.1 Graphs, Queries and Ontology Graphs

Data graph. A *data graph* is a directed graph $G = (V, E, L)$, where V is a finite set of *data nodes*, E is the edge set where $(u, u') \in E$ denotes a *data edge* from node u to u'; and L is a labeling function which assigns a label $L(v)$ (resp. $L(e)$) to a node $v \in V$ (resp. an edge $e \in E$)

In practice the function L may specify (1) the node labels as the description of entities, *e.g.,* URL, location, name, job, age; and (2) the edge labels as relationships between entities, *e.g.,* links, friendship, work, advice, support, exchange, co-membership [102].

Query graph. A *query graph* is a directed graph defined as $Q = (V_q, E_q, L_q)$, where (1) V_q and E_q are a set of *query nodes* and *query edges*, respectively; and

(2) L_q is a labeling function such that for each node $v \in V_q$ (resp. $e \in E_q$), $L_q(u)$ (resp. $L_q(e)$) is a node (resp. edge) label.

Ontology graph. In real applications the ontologies and their relationships can typically be represented as standardized *ontology graphs* [13, 27, 37, 80]. An ontology graph $O{=}(V_r, E_r)$ is an undirected graph, where (1) V_r is a node set, where each node $v_r \in V_r$ is a label referring to an entity; and (2) E_r is a set of edges among the labels, where each edge $e_r \in E_r$ represents a semantic relation (*e.g.,* “refer to,” “is a,” “specialization” [80]) between two nodes.

In addition, we denote as $\mathsf{sim}(v_{r_1}, v_{r_2})$ a *similarity function*, which computes the similarity of two nodes v_{r_1} and v_{r_2} in O as a real value in $[0, 1]$. Following ontology-based querying [80], (1) $\mathsf{sim}(v_{r_1}, v_{r_2})$ is a *monotonically decreasing function* of the distance from v_{r_1} to v_{r_2} in O, and (2) $\mathsf{sim}(v_{r_1}, v_{r_2}) = \mathsf{sim}(v_{r_2}, v_{r_1})$. Intuitively, the closer v_{r_1} and v_{r_2} are in O, the more similar they are [33, 34, 80]. For example, $\mathsf{sim}(v_{r_1}, v_{r_2})$ can be defined as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$, where $\mathsf{dist}(v_{r_1}, v_{r_2})$ is the distance from v_{r_1} to v_{r_2} in O [80].

Remarks. In practice, the ontology graphs and sim () can be obtained from, *e.g.,* semantic Web applications [37], Web mining [83], or domain experts [13]. While proposing more sophisticated models for ontologies and similarity functions are beyond the scope of this work, we focus on technique that applies to a class of similarity functions $\mathsf{sim}\ ()$. Note that $\mathsf{sim}\ ()$ can also be revised for directed ontology graphs.

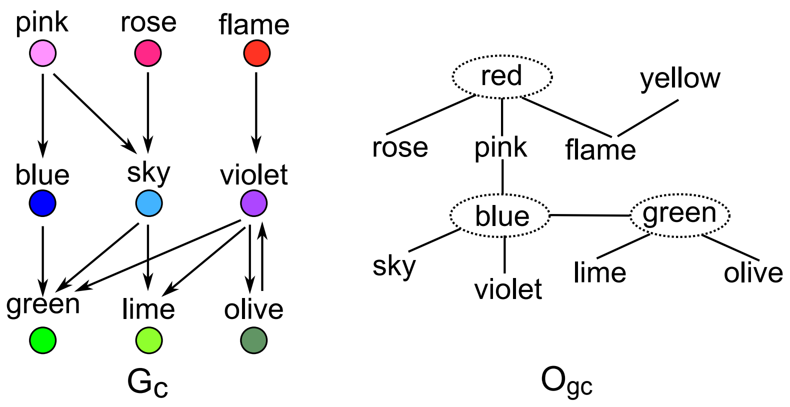


Figure 4.3: Data graph and ontology graph.

Example 17: The graph Q (resp. G) in Figure 4.1 depicts a query graph (resp. a data graph). There are three types of edge relations in both G and Q, *i.e.,* `recom`, `near`, and `guide`. All the other edges in G share a same type (not shown). The ontology graph O_g in Figure 4.3 illustrates the relationships among the entities in G, *e.g.,* `moonlight` is *relocated* as `riverside` (edge $e(\texttt{moonlight}, \texttt{riverside})$). A similarity function $\mathsf{sim}(v_{r_1}, v_{r_2})$ for O_g can be defined as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$. For example, $\mathsf{sim}(\texttt{museum}, \texttt{Disneyland}) = 0.9^2 = 0.81$.

As another example, consider the data graph G_c and an ontology graph O_{g_c} given in Figure 4.3. The nodes in G_c are labeled with colors (*e.g.,* `blue`). All the edges in G_c indicates the relationship "similar with," *e.g.,* the edge $(\texttt{red}, \texttt{rose})$

indicates that `red` is close to `rose`. Similarly, we define $\mathsf{sim}(v_{r_1}, v_{r_2})$ as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$ for nodes v_{r_1} and v_{r_2} in O_{g_c}. □

4.2.2 Ontology-based Subgraph Querying

Given a query graph $\mathsf{Q} = (V_q, E_q, L_q)$, an ontology graph O, a data graph $G = (V, E, L)$, a similarity function $\mathsf{sim}()$ and a similarity threshold θ, the *ontology-based querying* is to find the subgraphs $G' = (V', E', L')$ of G, such that there is a *bijective function* h from V_q to V' where (a) $\mathsf{sim}(L(h(u)), L_q(u)) \geq \theta$, and (b) (u, u') is a query edge if and only if $(h(u), h(u'))$ is a data edge, and they have the same edge label. We refer to G' as a *match* of Q in G induced by the mapping h, and denote all the matches in G for Q as $\mathsf{Q}(G)$. In addition, the *candidate set* for a query node u as the set of nodes v where $\mathsf{sim}(u, v) \geq \theta$. Here we assume *w.l.o.g.* that all the node labels in G are from O.

Top-K subgraph querying. In practice one often wants to identify the matches that are semantically "closest" to a query. We present a quantitative metric for the overall similarity between a query graph Q and its match G' induced by a mapping h, defined by a function C as follows.

$$C(h) = \sum \mathsf{sim}(L_q(u), L(h(u))), u \in V_q.$$

The metric favors the matches that are semantically close to the query: the larger the similarity score $C(h)$ is, the better the mapping is. On the other hand, if a subgraph G' matches Q with identical node labels, *i.e.,* via a subgraph isomorphism mapping h, $C(h)$ has the maximum value. Indeed, traditional subgraph isomorphism is a special case of the ontology-based subgraph querying, when the similarity threshold $\theta = 1$.

The metric naturally gives rise to an optimization problem. Given Q, G, O and an integer K, the *top K matches problem* is to identify K matches for Q in G with the largest similarity.

Example 18: Recall the query Q, the data graph G in Figure 4.1 and the ontology graph O_g in Figure 4.2. Assume the similarity threshold $\theta = 0.9$. One may verify that the candidate set of query node `museum` can(`museum`)$=$ {`Royal Gallery`,`attractions`, `park`}, and similarly, can(`moonlight`) $=$ {`riverside`, `Holiday Cafe`, `Holiday Plaza`}. The match G' for Q in G has the maximum similarity sim(`museum`, `Royal Gallery`) $+$ sim(`tourists`, `Culture Tour Community`) $+$ sim(`moonlight`, `riverside`) $= 0.9 * 3 = 2.7$. $\Box$

One may verify that the top K matches problem is NP-hard. Indeed, the traditional subgraph isomorphism is a special case of the problem, which is known to be NP-complete [148]. We next provide a query evaluation framework for the problem.

4.3 Querying Framework

Traditional ontology-based querying, by and large, relies on query rewriting techniques [20], which replaces query nodes with its candidates and may yield an exponential number of queries. These queries are then evaluated to produce all the results. This may not be practical for ontology-based subgraph querying. Alternatively, a similarity matrix can be computed, where each entry records the similarity between the query nodes and its candidates. Nevertheless, (1) the matrix incurs high space and construction cost ($O(|Q||G|)$), and needs to be computed upon each query, and (2) the time complexity is relatively high for both the exact algorithms (*e.g.,* [148]) and approximation algorithms [43] over the entire data graph.

Using ontology graphs, we can do better. Since it is hard to reduce the complexity of the isomorphism test, we develop a filtering-and-verification framework to reduce the input of the ontology-based querying. Upon receiving a query, the framework evaluates the query as follows. (1) During the filtering phase, the framework uses an *ontology index* to either extract a small subgraph of the data graph that *contains* all the matches, or determine the nonexistence of the match, in *polynomial time*; and (2) during the verification phase, the framework extracts the best matches from the small subgraph in (1), *without* searching the entire data graph.

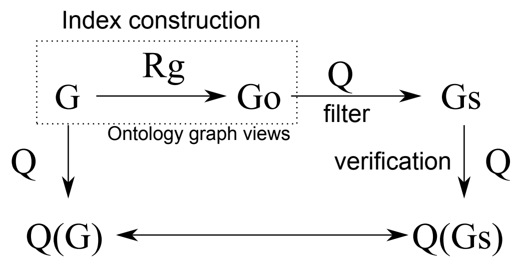


Figure 4.4: Ontology-based querying framework.

Overview of the framework. The framework has three components, as illustrated in Figure 4.4. The ontology index is constructed once for all in the first phase, while the query is evaluated via the filtering and verification phases.

Index construction. The framework first constructs an *ontology index* for a data graph G, as a set of *concept graphs*. Each concept graph is an abstraction of G by merging the nodes with similar labels in the ontology graph. The index is precomputed once, and is dynamically maintained upon changes to G.

Filtering. Upon receiving a query Q, the framework extracts a small subgraph as a compact representation of all the matches that are similar to Q, by visiting the concept graphs iteratively. If such a subgraph is empty, the framework determines that Q has no match in G. Otherwise, the matches can be extracted from the subgraph directly without accessing G.

Verification. The framework then performs isomorphism checking between the query and the extracted subgraph to extract the (top K) matches for Q.

We next provide the details of each phase of the framework.

4.4 Ontology-based Indexing

In this section we introduce the indexing and filtering phases of the ontology-based subgraph querying framework. We introduce the ontology index in Section 4.4.1, and present the filtering phase in Section 4.4.2 based on the index.

4.4.1 Ontology Index

The ontology index consists of a set of abstractions of a data graph G. Each abstraction, denoted as a *concept graph*, is constructed by grouping and merging the nodes in G, which all have a label similar to a label in the ontology graph O.

Given a data graph $G = (V, E, L)$ and an ontology graph O (with similarity function sim), as well as a similarity threshold β, a *concept graph* $G_o = (V_o, E_o, L_o)$ is a directed graph where (1) V_o is a partition of V, where each $v_o \in V_o$ is a set of data nodes, (2) each v_o has a label $L_o(v_o)$ from O, such that for any data node $v \in v_o$ and its label $L(v)$, $\mathsf{sim}(L(v), L_o(v_o)) \geq \beta$, and (3) (v_{o_1}, v_{o_2}) is an edge in E_o if and only if for *each* node v_1 in v_{o_1} (resp. v_2 in v_{o_2}), there is a node v_2 in v_{o_2} (resp. v_1 in v_{o_1}), such that (v_1, v_2) (resp. (v_2, v_1)) is an edge in G. We refer the set of the labels $L_o(v_o)$ to as *concept labels*.

Intuitively, a concept graph provides a "perspective" of the data graph in terms of several concept labels from the ontology graph. (1) Each node in the concept graph represents a group of nodes that are all similar to (extended from) a same label as a "concept" [80]. (2) Each edge in the concept graph represents a set of edges connecting the nodes in the two groups of nodes corresponding to two concepts. Hence, a concept graph is an abstraction of a data graph, by capturing both the semantics of its node labels as well as its topology.

Remarks. The abstraction of a graph is typically constructed by grouping a set of similar or equivalent nodes. Bisimulation [114] and regular equivalence [14] are used to generate abstract views of graphs [103], where two nodes are equivalent if they have a set of equivalent children with the same set of labels. In contrast, the nodes in a concept graph contains the nodes that are similar to a same label in a given ontology graphs, even they themselves may not have the same label.

Based on the concept graphs, an *ontology index* $\mathcal{I}$ of G is a set of concept graphs $\{G_{o_1}, \ldots, G_{o_m}\}$, where each concept graph G_{o_i} has distinct concept label

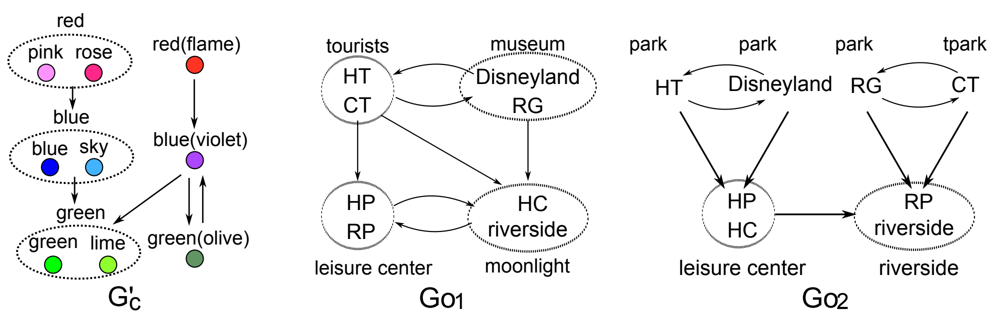


Figure 4.5: Ontology index and concept graphs.

set and similarity threshold β. Note that we distinguish the similarity threshold β for generating concept graphs from the threshold θ for the queries (Section 4.2), although they may have the same value.

Example 19: Consider the data graph G_c and the ontology graph O_{g_c} in Figure 4.3. Fixing a similarity threshold $\beta = 0.81$, and setting $\Sigma = \{\texttt{red}, \texttt{blue}, \texttt{green}\}$ in O_{g_c} as concept labels, a concept graph G'_c *w.r.t.* Σ is as shown in Figure 4.5. Each node in G'_c represents a set of nodes with labels similar to a concept label, *e.g.*, the node `red` is a set $\{\texttt{rose}, \texttt{pink}\}$, where $\mathsf{sim}(\texttt{red}, \texttt{rose})$ and $\mathsf{sim}(\texttt{red}, \texttt{pink})$ are both 0.9 (as defined in Example 17). On the other hand, although the node `violet` is similar to a concept label `blue`, it is not grouped with the node `sky` in G'_c. Indeed, while `violet` has a parent `olive` similar with the concept label `green`, the node `sky` has no such parent. Figure 4.5 illustrates two concept graphs G_{o_1} and G_{o_2} for the data graph G in Figure 4.1, where the similarity threshold β is 0.81.

The concept graphs G_{o_1} and G_{o_2} are constructed in terms of two sets of concept labels {`museum`,`tourists`,`moonlight`, `leisure center`}, and {`park`,,`riverside`, `leisure center`}, respectively. An ontology index $\mathcal{I}$ is the set $\{G_{o_1}, G_{o_2}\}$. □

Index construction. We next present an algorithm to construct the ontology index for a given data graph, in quadratic time.

Proposition 7: *Given a data graph $G(V, E, L)$, an ontology index can be constructed in $O(|E| \log |V|)$ time.* □

Algorithm 6 Algorithm OntoIdx

```
Input: $O$, $G$, similarity threshold $\beta$, integer $N$;
Output: Ontology index $\mathcal{I}$;
1: $\mathcal{I}$=$\emptyset$;
2: generate $N$ distinct concept label sets $\{C_1, \ldots, C_N\}$;
3: for each $C_i$ do
4:     $\mathcal{I}$=$\mathcal{I} \cup$ CGraph$(\beta, C_i, O, G)$;
5: end for
```

The algorithm, denoted as OntoIdx, takes as input the graphs G and O, a similarity threshold β, and an integer N as the number of the concept graphs to be generated. As shown in Alg. 6, the algorithm first initializes a set $\mathcal{I}$ as the ontology index (line 1). It then performs the following two steps.

Algorithm 7 Algorithm CGraph

```
Input: $O$, $G$, threshold $\beta$, concept label set $C$=$\{l_1, \ldots, l_m\}$;
Output: concept graph $G_o = (V_o, E_o, L_o)$;
1: construct partition $V_o$ of $V$ as $\{V_1, \ldots, V_m\}$, where $L_o(V_i)$=$l_i$, $V_i = \{v|\mathsf{sim}(L(v), l_i) \geq \beta\}$;
2: set $E_o := \{(V_1, V_2)|(v_1, v_2 \in E), v_1 \in V_1, v_2 \in V_2\}$;
3: while there is change in $V_o$ do
4:     if there is an edge $(v_{o_1}, v_{o_2})$ where $v_1 \in v_{o_1}$ has no child in $v_{o_2}$ (resp. $v_2 \in v_{o_2}$ has no parent in $v_{o_1}$ ) then
5:         SplitMerge($v_{o_1}, G_o$) (resp. SplitMerge($v_{o_2}, G_o$));
6:         update $G_o$;
7:     end if
8: end while
9: return $G_o := (V_o, E_o, L_o)$;
```

Concept labels selection (line 2). OntoIdx uses the following strategy to generate concept label sets by exploiting the partition techniques. For a given similarity threshold β, (1) it partitions O via graph clustering or ontology partitioning techniques [4,126,134], where the nodes in O are partitioned into several clusters, and (2) for each cluster, OntoIdx iteratively selects a label l and add it into a set C, and removes all the labels l' where $\mathsf{sim}(l, l') \geq \beta$ in the cluster. The process repeats until there is no label remains in the cluster, and the set C is returned after all the clusters are processed in O. One may verify that the strategy produces a set of concept labels C, such that for any label in a data graph l', there exists a concept label $l \in C$ where $\mathsf{sim}(l, l') \geq \beta$. OntoIdx uses the strategy to generate N distinct sets of concept labels (line 2).

Concept graph construction (lines 3-5). After the concept labels are selected, OntoIdx then invokes procedure CGraph to compute a concept graph and extend $\mathcal{I}$ (lines 3-4) for each concept label set C_i, until all C_i are processed (line 5).

Procedure CGraph constructs a concept graph G_o as follows. It constructs the node set V_o as a node partition of the data graph G, where each node of V_o consists of the nodes with similarity to a concept label bounded by β (line 1). The edge set E_o is also constructed accordingly (line 2). It then checks the condition that whether for each edge (v_{o_1}, v_{o_2}) of G_o, each node in v_{o_1} (resp. v_{o_2}) has a child in v_{o_2} (resp. parent in v_{o_1}) (line 4). If not, it invokes a procedure SplitMerge (omitted) to refine V_o by splitting and merging the node v_{o_1} (resp. v_{o_2}) to make the condition satisfied (line 5). The graph G_o is updated accordingly with the new node and edge set (line 6). The refinement process repeats until a fixpoint is reached, and G_o is returned as a concept graph (line 7).

Example 20: Recall the data graph G_c and ontology graph O_{g_c}in Figure 4.3. To compute a concept graph of G_c, The algorithm OntoIdx first generates a set of concept labels as $\{\texttt{red}, \texttt{blue}, \texttt{green}\}$ (line 2). It then invokes CGraph to construct a node partition of G as the node set of G_c (line 4), and generates G_{c_0} as shown in Figure 4.6. Each node and edge is refined according to the definition of the concept graph (lines 3-6). For example, the node (`green, lime, olive`) labeled with `green` is split into two nodes (`green, lime`) and `olive` by invoking procedure SplitMerge (line 5), which updates G_{c_0} to G_{c_1} (Figure 4.6). Similarly, CGraph (1) splits the node (`blue, sky, violet`) into (`blue, sky`) and `violet`, and updates G_{c_1} to G_{c_2}, and (2) splits the node (`pink, rose, flame`) to produce G'_c (Figure 4.5) as the final concept graph. □

Correctness and Complexity. The algorithm CGraph correctly computes a set of concept graphs as the ontology index. For the complexity, (a) the concept labels can be selected in $O(|\mathrm{O}|)$ time; (b) the time complexity of SplitMerge and CGraph is $O(|V| + |E|)$ and $O(|E| \log |V|)$, respectively; and (c) the procedure CGraph is invoked at most N times (lines 4-5). Thus, the total time complexity of OntoIdx is $O(N * |E| \log |V|)$. As N is typically small comparing with $|V|$ and $|E|$, the overall complexity of OntoIdx is thus $O(|E| \log |V|)$. The above analysis also completes the proof of Proposition 7.

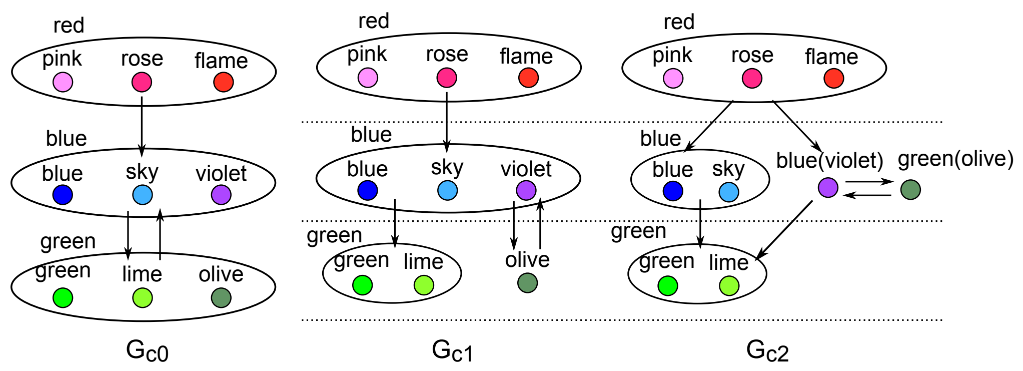


Figure 4.6: Construction of concept graphs.

4.4.2 Ontology-based Filtering

In this section, we illustrate the filtering phase of the query evaluation framework based on the ontology index. As remarked earlier, instead of performing subgraph isomorphism directly over the data graph G, we extract a (typically small) subgraph of G that contains all the matches of the query. Ideally, one wants to identify the minimum subgraph which is simply the union of all its matches. Nevertheless, to find such an optimal subgraph is already NP-complete [48].

Instead, we use ontology index to efficiently reduce the nodes and edges that are not in any matches as much as possible, and extract a subgraph G_v of G, which is induced by a relation M between the query nodes in a query Q and the nodes in a concept graphs G_c. The relation M is a relaxation of the subgraph isomorphism which guarantees the following condition: (1) for each query node u

in Q and its matches v (if any), v is in a node $M(u)$ in G_c, (2) for each query node u and each edge (u, u_1) (resp. (u_2, u)) in Q, $(M(u), M(u_1))$ (resp. $(M(u_2), M(u))$ is an edge in G_c. The subgraph G_v is extracted from G_c by "collapsing" $M(u)$ for each query node u to a set of corresponding data nodes in G. If multiple matching relations are computed from a set of concept graphs in $\mathcal{I}$, for each query node u, $M(u)$ is refined as $\bigcap M_i(u)$, where $M_i(u)$ is collected from G_{c_i} in the ontology index.

The following result shows the relationship between the subgraph G_v and ontology index.

Proposition 8: *Given an ontology index $\mathcal{I}$, a query graph Q and a data graph G, if the subgraph G_v is empty, then $\mathsf{Q}(G)$ is empty; otherwise, $\mathsf{Q}(G) = \mathsf{Q}(G_v)$, and (2) G_v can be computed in $O(|\mathsf{Q}||\mathcal{I}|)$ time, where $|\mathcal{I}|$ (resp. $|\mathsf{Q}|$) is the total number of nodes and edges in $\mathcal{I}$ (resp. $|\mathsf{Q}|$).* □

To see Proposition 8 (1), observe that if Q has a match G' induced by an ontology-based isomorphism mapping h, then a relation M can be constructed such that for any query node u, $h(u) \in M(u)$. Thus, G_v contains all the matches of Q, and $\mathsf{Q}(G) = \mathsf{Q}(G_v)$. On the other hand, if G_v is empty, then no match exists in G for Q and $\mathsf{Q}(G)$ is empty, since no relation M exists even as a relaxation of subgraph isomorphism.

To prove Proposition 8 (2), we introduce an algorithm, denoted as Gview, to generate G_v from $\mathcal{I}$ in $O(|\mathsf{Q}||\mathcal{I}|)$ time.

Algorithm. The algorithm Gview takes as input a query Q, data graph G and a user-defined similarity threshold θ. It has the following three steps.

Initialization. For each query node v_q, it initializes a *match set* $\mathsf{mat}(v_q)$, to record the final matches identified by the matching relation M (as remarked earlier), as well as the candidate set $\mathsf{can}(v_q)$ to keep track of the matches when a single concept graph is processed.

Matching and refinement. Gview computes the relation M as follows. It first initializes the candidate set $\mathsf{can}(v_q)$ for each query node v_q, using a "lazy" strategy (as will be discussed) (line 4). It then conducts a fixpoint computation, by checking if there exists a query edge (v_{q_1}, v_{q_2}), such that there is a node $v_{o_1} \in \mathsf{can}(v_{q_1})$ which has no child in $\mathsf{can}(v_{q_2})$. If so, v_{q_1} (and all the data nodes contained in it) is no longer a candidate for v_q. Gview thus removes v_{q_1} from $\mathsf{can}(v_{q_1})$. If $\mathsf{can}(v_{q_1})$ is empty, then query node q_1 has no valid candidate in some concept graph, and Gview returns $\emptyset$. Otherwise, $\mathsf{mat}(v_{q_1})$ is refined by $\mathsf{can}(v_{q_1})$: if $\mathsf{mat}(v_{q_1})$ is empty, it is initialized with $\mathsf{can}(v_{q_1})$; otherwise, $\mathsf{mat}(v_{q_1})$ only keeps those candidates that are in $\mathsf{can}(v_{q_1})$. If $\mathsf{mat}(v_{q_1})$ becomes empty, no candidate can be find in G for v_{q_1}, and Gview returns $\emptyset$.

G_v construction. After all the concept graphs in $\mathcal{I}$ are processed, Gview constructs G_v with a node set V_{q_v}, which contains a node for each match set, and a corresponding edge set E_{q_v}. G_v is then returned.

It is costly to identify the candidates for the query nodes in Q by accessing the ontology graph O and G, which may take up to $O(|\mathsf{Q}||G|)$ time. Instead of identifying the candidates for a query node v_q and the user-defined similarity threshold θ, a "lazy" strategy only identifies a set of nodes (as $\mathsf{can}(v_q)$) in the concept graph G_o, such that the candidates of v_q are contained in these nodes. To this end, it simply selects the nodes in G_o labeled with the concept labels l, where the distance of l and the label of v_q in O is less than $\mathsf{sim}^{-1}(\theta) + \mathsf{sim}^{-1}(\beta)$. Here β is the similarity threshold adopted to generate G_o. One may verify that each candidate of v_q *w.r.t.* the similarity threshold θ is in one of such nodes, since the similarity function $\mathsf{sim}()$ is a monotonically decreasing function of the label distances in O. Moreover, the total candidate selection time is reduced to as $O(|\mathsf{Q}||O|)$. Note that $|\mathsf{Q}|$ and $|O|$ are typically small comparing to $|G|$.

Example 21: Recall the query Q in Example. 15. Using the ontology index $\mathcal{I} = \{\ G_{o_1}\ G_{o_2}\ \}$ (Figure 4.5), Gview extracts G_v as follows. (1) Using G_{o_1}, Gview initializes can(moonlight) with the node `moonlight` in G_{o_1}, and similarly initializes can(museum) and can(tourists) (line 4). For *e.g.,* query edge {`tourist`,`moonlight`}, Gview refines can(tourists) by checking if every node in can(tourists) has a child in can(moonlight) (line 5-10). After the refinement, mat(moonlight) = {`HC, riverside`}, mat(museum) = {`Disneyland, RG`} and mat(tourists) = {`HT, CT`}. (2) Using G_{o_2}, Gview identifies that can(tourists) = {`CT`}, can(museum) = {`RG`}, and can(moonlight) = {`riverside`, `RP`}. (3) Putting these together, the final match sets mat(*moonlight*) = {`riverside`}, mat(tourists) = {`CT`} ∩{`HT, CT`} = `CT`, and mat(museum) = {`RG`} ∩{`Disneyland, RG`} = {`RG`}. G_v (as shown in Figure 4.7) is then constructed as the subgraph of G induced by the nodes `riverside`, `CT`, and `RG`. □

Correctness and complexity. The algorithm Gview correctly computes a subgraph G_v. To see this, observe that (1) G_v is initialized using the lazy strategy contains all the possible matches; (2) for each query edge (v_{q_1}, v_{q_2}), Gview uses can(v_{q_2}) to refine can(v_{q_1}) in each concept graph, and only removes those nodes that are not matches (non-matches) for v_{q_1}; and (3) if can(v_q) is empty when processing a concept graph, then there is no match in G for v_q. Since if there indeed exists a data node v that can match v_q, then for every query edge (v_q, v'_q), there must exist a corresponding edge (v_o, v'_o) $(v \in v_o)$ in *every* concept graph. Thus, Gview only removes non-matches of Q from the initialized G_v. The correctness of Gview thus follows.

For the complexity, (1) it takes $O(|V_q||C|)$ to identify the candidates for the query nodes using lazy strategy, where $|C|$ is the total number of concept labels . The filtering process can be implemented in time $O(|E_q||\mathcal{I}|)$. The construction of G_v is in time $O(|\mathcal{I}|)$. Putting these together, the total time of Gview is in $O(|E_q||\mathcal{I}|)$. In practice $|E_q|$ is typically small, and the complexity of Gview can be considered as near-linear *w.r.t.* $|\mathcal{I}|$.

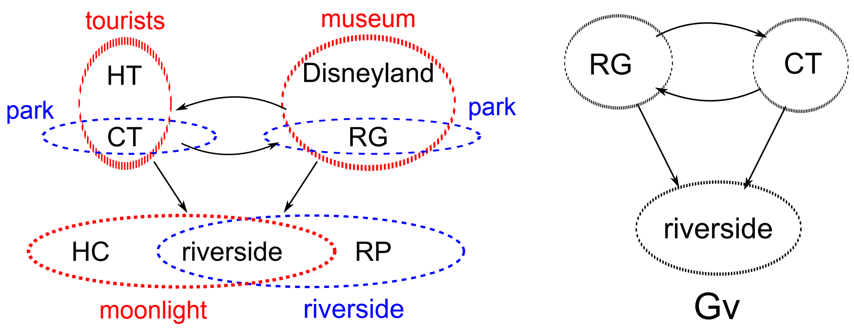


Figure 4.7: Generating subgraph G_v.

4.5 Subgraph Query Processing

In the verification phase, the framework performs the subgraph isomorphism tests over the subgraph extracted from the ontology index. We provide a global ontology-based subgraph querying algorithm for the top K matches problem. The algorithm, denoted as KMatch.

Algorithm. Upon receiving a query Q, the algorithm KMatch first extracts the subgraph G_s by invoking the procedure Gview (see Section 4.4), For each query node v_q, it constructs a candidate list $\mathcal{L}(v_q)$, sorted in the descending order of the similarity. KMatch then iteratively constructs a subgraph G_s using the candidates with the largest similarity from the candidate lists, and if G_s is a match, it inserts G_s to a heap H. The above process repeats until all such G_s is processed, or H contains the top K matches with maximum similarity scores.

It takes $O(|\mathsf{Q}||\mathcal{I}|)$ time to compute G_s, as remarked earlier. The total time of KMatch is thus in $(|\mathsf{Q}||\mathcal{I}| + |G_s|^{|V_q|})$. As verified in our experiment, in practice G_s is significantly smaller than G (see Section 4.6).

Remarks. The ontology-based subgraph querying framework can be easily adapted to support traditional subgraph isomorphism. Indeed, when the user-defined similarity threshold is 1.0, (1) the ontology index can be used to extract a subgraph G_v, which only contains the candidate nodes with identical labels for the query nodes, and (2) any match extracted from G_s is a subgraph isomorphic to Q in terms of identical subgraph isomorphism.

4.6 Experimental Evaluation

We next present an experimental study using real-life graph data. We conducted three sets of experiments to evaluate: (1) the effectiveness of the ontology-based subgraph querying, (2) the efficiency of the query evaluation framework, and (3) the performance and cost of the ontology index.

Datasets. (a) CrossDomain is taken from a benchmark suite FebBench [127], which consists of (i) an RDF data graph with 1.07M nodes and 3.86M edges where nodes represent entities from different domains (*e.g.,* Wikipedia, locations, biology, music, newspapers), and edges represent the relationship between the entities (*e.g.,* born in, locate at, favors); and (ii) an ontology graph with 1.44M concepts and 5.30M relations. The data graph takes in total 150Mb physical memory. (b) Flickr contains a data graph taken from `http://press.liacs.nl/mirflickr/` with 1.3M nodes and 6.42M edges, where the nodes represent images, tags, users or locations, and edges represent their relationship. It also contains an ontology graph from DBpedia (`http://dbpedia.org`) with more than 3.64 million entities. The data graph takes in total 194Mb physical memory. In our experiments, we employ the ontology graph to describe the tags in Flickr.

Following [80], we set the similarity function as $\mathsf{sim}(l', l) = 0.9^{\mathsf{dist}(l', l)}$ for all the ontology graphs O, where $\mathsf{dist}(l', l)$ is the distance between two nodes l' and l in O. For example, if a label l is 2 hops away from l' in O, $\mathsf{sim}(l', l) = 0.81$.

Implementation. We implemented the following algorithms in C++: (1) algorithm OntoIdx; (2) algorithm KMatch; (3) SubIso, the subgraph isomorphism algorithm in [147], which identifies the matches using identical label matching; (4) $\mathsf{SubIso_r}$, which, as a comparison to KMatch, is revised from [147] that rewrites the query graph, and directly computes all the matches and select the best ones; (5) VF2, which computes the minimum weighted matches, by exploiting a *similarity matrix* between the query label and all the labels in the data nodes;

To favor VF2, we precomputed a similarity matrix, where each entry records $\mathsf{sim}(u, v)$ as the similarity between a query node u and a data node v *w.r.t.* the ontology graph O. We also optimized VF2 such that it terminates as soon as the top K matches are identified. The time cost of computing the similarity matrix is not counted for VF2.

We used a machine powered by an Intel(R) Core 2.8GHz CPU and 8GB of RAM, using Ubuntu 10.10. Each experiment was run 5 times and the average is reported here.

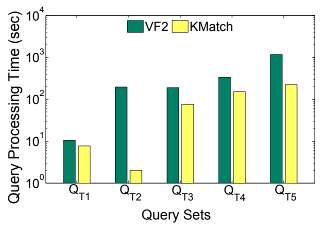
Experimental results. We next present our findings.

Exp-1: Effectiveness and flexibility. In this set of experiments, we first evaluated the effectiveness of KMatch and SubIso. We generated 5 query templates for CrossDomain, and 4 query templates for Flickr. We use $(|V_p|, |E_p|, |L_p|)$ to denote the size of a query $\mathsf{Q}(V_p, E_p, L_p)$. For CrossDomain, (1) Q_{T_1} is a tree of size $(4, 3, 3)$ searching for movies, directors and distributors, and Q_{T_2} of size $(4, 4, 3)$ is a cycle obtained by inserting an edge to Q_{T_1}; (2) Q_{T_3} of size $(4, 6, 4)$ is to search pop stars, record companies, albums and songs, and Q_4 is obtained by only "generalizing" the query label of Q_{T_3}, *e.g.,* from "Green Record Company" to "Record Company"; and (3) Q_{T_5} of size $(5, 6, 4)$ is to identify the soccer stars, clubs and their teammates. Similarly, for Flickr the 4 queries Q_{T_6} to Q_{T_9} are to identify photos of animals taken at specified locations. Each template Q_{T_i} is populated as a query set of 100 queries (also denoted as Q_{T_i}) by varying the node labels only. For ontology index, we employ the graph partitioning algorithm in [126] to generate concept labels with similarity threshold $\beta = 0.8$, unless otherwise specified.

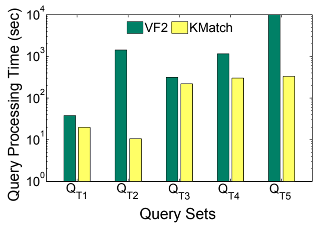
Query	CrossDomain		
	θ=1	θ=0.9	θ=0.8
Q_{T_1}	1	2,687	9,099
Q_{T_2}	0	24	271
Q_{T_3}	1	170	342
Q_{T_4}	0	405	991
Q_{T_5}	0	30,854	48,225

Query	Flickr		
	θ=1	θ=0.9	θ=0.8
Q_{T_6}	2	6	307
Q_{T_7}	0	177	2,160
Q_{T_8}	0	448	6,028
Q_{T_9}	0	799	15,052

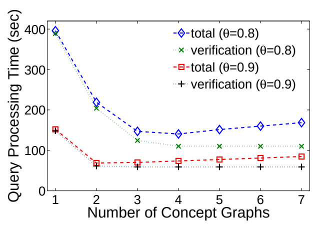
Table 4.1: Effectiveness of querying real-life graphs.



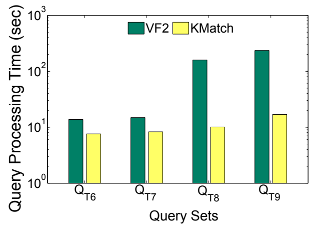
(a) Efficiency: CrossDomain (θ=0.9)



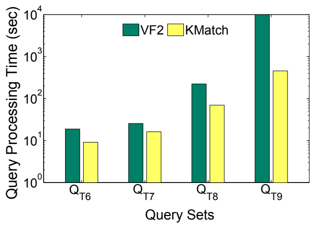
(b) Efficiency: CrossDomain (θ=0.8)



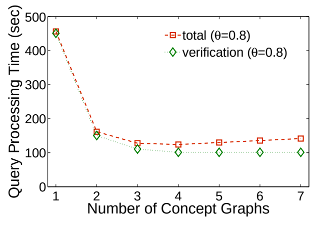
(c) Varying card ($\mathcal{I}$)



(d) Efficiency: Flickr (θ=0.9)



(e) Efficiency: Flickr (θ=0.8)



(f) Varying card ($\mathcal{I}$)

Figure 4.8: Evaluation on ontology-based subgraph querying.

Effectiveness. We first compared the number of matches found by SubIso and KMatch over CrossDomain and Flickr, as shown in Table 4.1. Fixing card $\mathcal{I} = 1$, *i.e.*, the ontology index $\mathcal{I}$ contains a single concept graph, we varied the similarity threshold of the queries from 1.0 to 0.8, and identify *all* the matches. For all the queries over CrossDomain, SubIso only finds in average 1 exact match for query set Q_{T_1} and Q_{T_3}, and no match for all other queries. In contrast, KMatch identifies much more matches that are semantically close to the query according to our observation. It also finds more meaningful matches than SubIso over Flickr.

Two sample patterns and their closest matches are shown in Figure 4.9. (1) Query Q_2 in Q_{T_2} (Figure 4.9(a)) over CrossDomain is to find two movies distributed by `Walt Disney` and directed by `James Cameron`, where one is screened out of competition at `Cannes Film Festival`, and the other is related with `Aliens`. The closest match is shown in Figure 4.9(b) where `Aliens` is matched to the movie `Aliens of the Deep`, and `Cannes Film Festival` has a match `Ghosts of the Abyss`. (2) Query Q_3 (Figure 4.9(c)) of Flickr is to identify two photos both related with `‘‘Flamingo’’` with color `‘‘Pink’’`, and one is taken in `San Diego` while the other in `Miami`. The closest match is given in Figure 4.9(d) where `Miami` is matched to “Seaworld” in Florida.

The algorithm VF2, via carefully processed similarity matrix, identifies the same set of matches as KMatch (thus is not shown) with much more running time, as will be shown.

Query flexibility. As shown in Table 4.1, (a) for all the queries, the match number increases when the similarity threshold θ decreases, since more data nodes become candidates and more subgraphs become matches; (b) fixing node number and labels, the insertion of edges increases the topological complexity of the query, *e.g.,* from Q_1 to Q_2, and thus, reduces the number of matches; and (c) fixing the structure, the query label generalization (from *e.g.,* Q_3 to Q_4) increases the candidates of the query nodes, which in turn increases the match number.

Exp-2: Efficiency and scalability. We evaluated the performance of KMatch, $\mathsf{SubIso_r}$ and VF2 using real-life datasets and synthetic data, and their scalability using synthetic data. In these experiments, the indexes were precomputed, and thus their construction time were not counted.

Real-life graphs. Figure 4.8(a) and Figure 4.8(b) (both in log scale) show the running time of KMatch and VF2 for evaluating Q_{T_1} to Q_{T_5} over CrossDomain in Table 4.1. The results tells us the following. (1) KMatch *always* outperform VF2. For example, KMatch takes only 1% of the running time of VF2 to evaluate Q_{T_2}. When $\theta = 0.9$ (resp. $\theta = 0.8$), KMatch takes 30% (resp. 22%) of the running time of VF2 in average for all the queries. (2) When θ decreases, both algorithms takes more time due to more candidates. In addition, KMatch improves the efficiency of VF2 better for larger θ due to the filtering power of the ontology index even with only a single concept graph.

To evaluate the scalability with $\mathsf{card}(\mathcal{I})$, *i.e.,* the number of concept graphs, we used CrossDomain, varied $\mathsf{card}(\mathcal{I})$ from 1 to 7, and tested the cases where θ is 0.9 and 0.8, respectively. The results, shown in Figure 4.8(c), tells us that the running time of KMatch, decreases while $\mathsf{card}(\mathcal{I})$ increases. Specifically, when θ = 0.8, the verification (resp. filtering) time decreases (resp. increases) from 396 (resp. 2) seconds to 110 (resp. 30) seconds when $\mathsf{card}(\mathcal{I})$ increases from 1 to 4, and the total time decreases from 398 seconds to 168 seconds. The total time increases when card($\mathcal{I}$) is increased from 4 to 7. This is because (a) more concept graphs effectively filter more candidates, and reduce the verification time, and (b) when card($\mathcal{I}$) > 4, while the index spends more time in filtering phase, it cannot further reduce the verification time, thus the total time increases. Similarly, the running time of KMatch decreases when θ= 0.9 and card($\mathcal{I}$) < 3.

The efficiency of KMatch and VF2 over Flickr is given in Figure 4.8(d), Figure 4.8(e), and Figure 4.8(f), which verify the results of their CrossDomain counterparts Figure 4.8(a), Figure 4.8(b), and Figure 4.8(c), respectively. In average, the running time of KMatch is 30% of that of VF2 over Flickr when $\theta = 0.9$. When $\theta = 0.8$, VF2 does not run to complete for Q_{T_4}.

Exp-3: Effectiveness of ontology index. We next investigate (1) ctime, *i.e.,* the running time of algorithm OntoIdx; (2) the compression rate cr $= \frac{|E_o|}{|E|}$, where $|E_o|$ is the average edge size in $\mathcal{I}$, and $|E|$ is the edge size of the data graph, (3) the memory reduction mr $= \frac{|M_o|}{|M|}$, where $|M_o|$ and M are the physical memory cost of $\mathcal{I}$ and the data graph, respectively; and (4) the filtering rate fr $= \frac{|G_v|}{|G_{sub}|}$, where $|G_v|$ is the average size of the induced subgraphs G_v in filtering phase, and $|G_{sub}|$ is the size of all the nodes and edges visited by VF2. We fixed card $\mathcal{I} = 1$, and β = 0.8. The result is shown below in Table 4.2

The above results tell us the following. (1) For both data sets, the efficiency of OntoIdx is comparable to that of VF2 for processing a single query (see Exp-2).

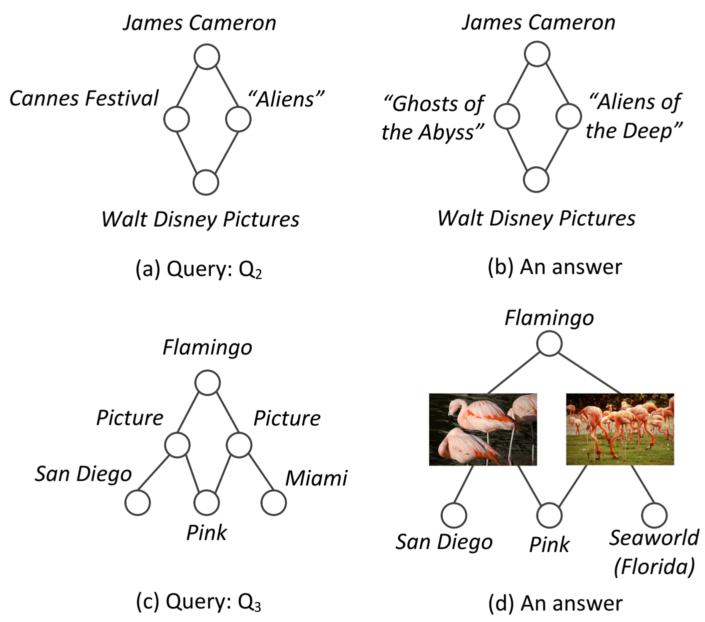


Figure 4.9: Case study: queries and the matches.

(2) $\mathcal{I}$ contains much less nodes and edges over the data graph, and takes only half of of its physical memory cost. (3) Even when only a single concept graph is used, the index effectively filters the search space. Indeed, the size of G_v for verification is only 6% and 24% of $|G_{sub}|$ over CrossDomain and Flickr, respectively.

Table 4.2: Effectiveness of indexing

Dataset	ctime	cr	mr	fr
CrossDomain	$694s$	0.43	0.51	0.06
Flickr	$383.83s$	0.71	0.52	0.24

4.7 Related Work

Ontology-based graph queries. The ontology information has been used for pattern mining [20], keyword searching [65] and the semantic Web [33, 88]. The Ontogator [97] exploits an ontology-based multi-facet search paradigm, which links keyword queries to a set of entities in multiple distinct ontology views, created via ontology projection. [20] proposes techniques to mine the frequent patterns over graphs with generalized labels in the input taxonomies. Classes hierarchy are used to evaluate queries specified by a SPARQL-style language over RDF graphs in [33], where approximate answers are identified, measured by a distance metric. Our work differs from theirs in the following. (a) We consider general ontology graphs rather than hierarchical taxonomies or class lattice. (b) We find matches for a given query graph, instead of discovering frequent patterns in graphs as in [20]. (c) The queries in [33] are defined in terms of a query language specified for semantic Web. In contrast, we study general subgraph queries with node and edge labels. Moreover, the queries in [33] are posed over RDF graphs with predefined schema, where we consider subgraph queries over general data graphs without any schema. (d) The query evaluation is not discussed in [33, 88].

Closer to our work is [88], which extends template graph searching by interpolating ontologies to data graphs. The data graphs are recursively extended by a set of ontologies from ontology queries, and are then queried by a template graph. Our work differs from theirs in (a) instead of merging ontology graphs with data graphs, we leverage ontology graph to develop filtering strategies to identify matches, and (b) we provide query evaluation and indexing techniques, while [88] focus on data fusion techniques. The incremental querying techniques are also not addressed in [88].

Graph abstraction. The concepts of bisimulation [114] and regular equivalence [14] are proposed to define the equivalent graph nodes, which can be grouped to form abstracted graphs as indexes [103]. In this work we use the similar idea to construct the ontology index, by abstracting data graphs as a set of concept graphs for efficient subgraph filtering and querying. However, while the notions in [14, 114] are based on label equality, a concept graph groups nodes in a data graph in terms of an external ontology graph, thus unifies the ontology similarity and graph abstraction, as discussed in Section 4.4.

4.8 Summary

We have proposed the ontology-based subgraph querying, based on a quantitative metric for the matches. These notions support finding matches that are semantically close to the query graphs. We have proposed a framework for finding the top-k closest matches, via a filtering and verification strategy using ontology index. In addition, we have proposed an incremental algorithm to update indexes upon data graph changes. Our experimental study have verified that the framework is able to efficiently identify the matches, which cannot be found by conventional subgraph isomorphism and its extensions.

Chapter 5

Summarizing Answer Graphs

Graph querying might generate an excessive number of matches, referred to as "answer graphs", that could include different relationships among keywords. An ignored yet important task is to group and summarize answer graphs that share similar structures and contents for better query interpretation and result understanding. This chapter studies the summarization problem for the answer graphs induced by a keyword query Q. (1) A notion of *summary graph* is proposed to characterize the summarization of answer graphs. Given Q and a set of answer graphs $\mathcal{G}$, a summary graph preserves the relation of the keywords in Q by summarizing the paths connecting the keywords nodes in $\mathcal{G}$. (2) A quality metric of summary graphs, called *coverage ratio*, is developed to measure information loss of summarization. (3) Based on the metric, a set of summarization problems are formulated, which aim to find minimized summary graphs with certain coverage ratio. (a) We show that the complexity of these summarization problems ranges from PTIME to NP-complete. (b) We provide exact and heuristic summarization algorithms. (4) Using real-life and synthetic graphs, we experimentally verify the effectiveness and the efficiency of our techniques.

5.1 Introduction

Keyword queries have been widely used for querying graph data, such as information networks, knowledge graphs, and social networks [152]. A keyword query Q is a set of keywords $\{k_1, \ldots, k_n\}$. The evaluation of Q over graphs is to extract data related with the keywords in Q [26, 152].

Various methods were developed to process keyword queries. In practice, these methods typically generate a set of graphs $\mathcal{G}$ *induced by* Q. Generally speaking, (a) the keywords in Q correspond to a set of nodes in these graphs, and (b) a path connecting two nodes related with keywords k_1, k_2 in Q suggests how the keywords are connected, *i.e.,* the relationship between the keyword pair (k_1, k_2). We refer to these graphs as *answer graphs* induced by Q. For example, (1) a host of work on keyword querying [57, 60, 72, 86, 115, 152] defines the query results as answer graphs; (2) keyword query interpretation [22, 145] transforms a keyword query into graph structured queries via the answer graphs extracted for the keyword;

(3) result summarization [68, 91] generates answer graphs as *e.g.,* "snippets" for keyword query results.

Nevertheless, keyword queries usually generate a great number of answer graphs (as intermediate or final results) that are too many to inspect, due to the sheer volume of data. This calls for effective techniques to summarize answer graphs with representative structures and contents. Better still, the summarization of answer graphs can be further used for a range of important keyword search applications.

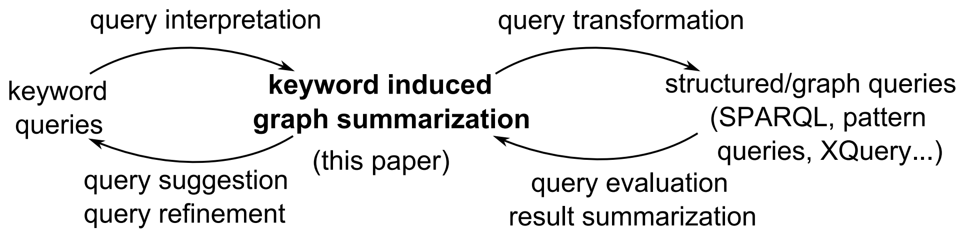


Figure 5.1: Graph summarization.

Enhance Search with Structure. It is known that there is an usability-expressivity tradeoff between keyword query and graph query [138] (as illustrated in Figure 5.1). For searching graph data, keyword queries are easy to formulate; however, they might be ambiguous due to the lack of structure support. In contrast, graph queries are more accurate and selective, but difficult to describe. Query interpretation targets the trade-off by constructing graph queries, *e.g.,* SPARQL [130], to find more accurate results. Nevertheless, there may exist many interpretations as answer graphs for a single keyword query [46]. A summarization technique may generate a small set of summaries, from which graph queries can be induced. That is, a user can first submit keyword queries and then pick up the desired graph queries, thus taking advantage of both keyword query and graph query.

Improve Result Understanding and Query Refinement. Due to query ambiguity and the sheer volume of data, keyword query evaluation often generates a large number of results [68, 82]. This calls for effective result summarization, such that users may easily understand the results without checking them one by one. Moreover, users may come up with better queries that are less ambiguous, by inspecting the connection of the keywords reflected in the summary. Based on the summarization result, efficient query refinement and suggestion [92, 125] may also be proposed.

Example 22: Consider a keyword query $Q = \{$ `Jaguar`, `America`, `history` $\}$ issued over a knowledge graph. Suppose there are three graphs G_1, G_2 and G_3 induced by the keywords in Q as *e.g.,* query results [86,115], as shown in Figure 5.1. Each node in an answer graph has a type, as well as its unique id. It is either

(a) a keyword node marked with '$*$' (*e.g.,* `Jaguar XK`*) which corresponds to a keyword (*e.g.,* `Jaguar`), or (b) a node connecting two keyword nodes.

The induced graphs for Q illustrate different relations among the same set of keywords. For example, G_1 suggests that “Jaguar” is a brand of cars with multiple offers in many cities of USA, while G_3 suggests that “Jaguar” is a kind of animals found in America. To find out the answers the users need, reasonable graph structured queries are required for more accurate searching [22]. To this end, one may construct a *summarization* over the answer graphs. Two such summaries can be constructed as G_{s_1} and G_{s_2}, which suggest two graph queries where “Jaguar” refers to a brand of car, and a kind of animal, respectively. Better still, by summarizing the relation between two keywords, more useful information can be provided to the users. For example, G_{s_1} suggests that users may search for “offers” and “company” of “Jaguar,” as well as their locations.

Assume that the user wants to find out how “Jaguar” and “America” are related in the search results. This requires a summarization that only considers the connection between the nodes containing the keywords. Graph G_s depicts such a summarization: it shows that (1) “Jaguar” relates to “America” as a type of car produced and sold in cities of USA, or (2) it is a kind of animal living in the continents of America. Moreover, in practice one often place a budget for summarizations [52,137]. This calls for quality metrics and techniques for concise summaries that illustrate the connection information between keywords as much as possible. □

The above example suggests that we may summarize answer graphs $\mathcal{G}$ induced by a keyword query Q, to help keyword query processing. We ask the following questions. (1) How to define "query-aware" summaries of $\mathcal{G}$ in terms of Q? (2) How to characterize the quality of a summary for Q? (3) How to efficiently identify good summaries under a budget?

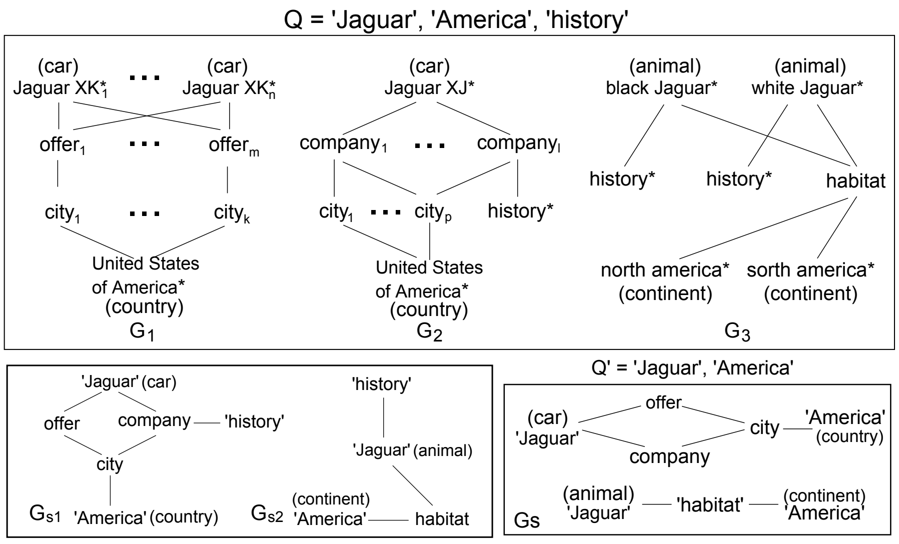


Figure 5.2: Keyword query on a knowledge graph.

We study the above problems for summarizing keyword induced answer graphs.

(1) We formulate the concept of answer graphs for a keyword query Q (Sec-

tion 5.2). To characterize the summarization for answer graphs, we propose a notion of *summary graph*. Given Q and $\mathcal{G}$, a summary graph captures the relationship among the keywords from Q in $\mathcal{G}$.

(2) We introduce quality metrics for summary graphs (Section 5.3). One is defined as the size of a summary graph, and the other is based on *coverage ratio* α, which measures the number of keyword pairs a summary graph can *cover* by summarizing pairwise relationships in $\mathcal{G}$.

Based on the quality metrics, we introduce two *summarization* problems. Given Q and $\mathcal{G}$, (a) the α-summarization problem is to find a minimum summary graph with a certain coverage ratio α; we consider 1-summarization problem as its special case where $\alpha = 1$; (b) the K summarization problem is to identify K summary graphs for $\mathcal{G}$, where each one summarizes a subset of answer graphs in $\mathcal{G}$. We show that the complexity of these problems ranges from PTIME to NP-complete. For the NP-hard problems, they are also hard to approximate.

(3) We propose exact and heuristic algorithms for the summarization problems. (a) For 1-summarization, we present an exact, quadratic-time algorithm to find a minimum 1-summary (Section 5.4). For a given keyword query, it is to identify a set of "redundant" (resp. "equivalent") nodes in $\mathcal{G}$ for Q, and construct the

summary by removing (resp. mergeing) these nodes. (b) We provide heuristic algorithms for the α-summarization (Section 5.4) and k summarization problems (Section 5.5), respectively. These algorithms greedily select and summarize answer graphs with the minimum estimated cost in terms of size and coverage.

(4) We experimentally verify the effectiveness and efficiency of our summarization techniques using both synthetic data and real-life datasets. We find that our algorithms effectively summarize the answer graphs. For example, they generate summary graphs that cover every pair of keywords with size in average 24% of the answer graphs. They also scale well with the size of the answer graphs. These effectively support summarization over answer graphs.

5.2 Preliminary

In this section, we formulate the concept of answer graphs induced by keyword queries, and their summarizations.

5.2.1 Keyword Induced Answer Graphs

Answer graphs. Given a keyword query Q as a set of keywords $\{k_1, \ldots, k_n\}$ [152], an *answer graph* induced by Q is a connected undirected graph $G = (V, E, L)$, where V is a node set, $E \subseteq V \times V$ is an edge set, and L is a labeling function which assigns, for each node v, a label $L(v)$ and a *unique* identity. In practice, the node labels may represent the type information in *e.g.,* RDF [115], or node attributes [165]. The node identity may represent a name, a property value, a *URI*, *e.g.,* "*dbpedia.org/resource/Jaguar*," and so on. Each node $v \in V$ is either a *keyword node* that corresponds to a keyword k in Q, or an *intermediate* node on a path between keyword nodes. We denote as v_k a keyword node of k. The keyword nodes and intermediate nodes are typically specified by the process that generates the answer graphs, *e.g.,* keyword query evaluation algorithms [152]. A path connecting two keyword nodes usually suggests a relation, or "connection pattern," as observed in *e.g.,* [45].

We shall use the following notations. (1) A path from keyword nodes v_k to v'_k is a nonempty *simple* node sequence $\{v_k, v_1 \ldots, v_n, v'_k\}$, where v_i $(i \in [1, n])$ are intermediate nodes. The label of a path ρ from v_k to v'_k, denoted as $L(\rho)$, is the concatenation of all the node labels on ρ. (2) The union of a set of answer graphs $G_i = (V_i, E_i, L_i)$ is a graph $G = (V, E, L)$, where $V = \bigcup V_i$, $E = \bigcup E_i$, and each node in V has a unique node id. (3) Given a set of answer graphs $\mathcal{G}$, we denote as $\mathsf{card}(\mathcal{G})$ the number of the answer graphs $\mathcal{G}$ contains, and $|\mathcal{G}|$ the total number of its nodes and edges. Note that an answer graph does not necessarily contain keyword nodes for all the keywords in Q, as common found in *e.g.,* keyword querying [152].

Example 23: Figure 5.2 illustrates a keyword query Q and a set of answer graphs $\mathcal{G} = \{G_1, G_2, G_3\}$ induced by Q. Each node in an answer graph has a label as its type (*e.g.,* `car`), and a unique string as its id (*e.g.,* `Jaguar XK`$_1$).

Consider the answer graph G_1. (a) The keyword nodes for the keyword `Jaguar` are `Jaguar`$_{XK_i}$ ($i \in [1, n]$), and the node `United States of America` is a keyword node for `America`. (b) The nodes `offer`$_i$ ($i \in [1, m]$) and `city`$_j$ ($j \in [1, k]$) are the intermediate nodes connecting the keyword nodes of `Jaguar` and `America`. (c) A path from `Jaguar` to `USA` passing the nodes `offer`$_1$ and `city`$_1$ has a label {`car`,`offer`, `city`,`country`}. Note that (1) nodes with different labels (*e.g.,* `Jaguar`$_{XK_1}$ labeled by "car" and `black jaguar` by "animal") may correspond to the same keyword (*e.g.,* `Jaguar`), and (2) a node (*e.g.,* `city`$_1$) may appear in different answer graphs (*e.g.,* G_1 and G_2). $\square$

5.2.2 Answer Graph Summarization

Summary graph. A summary graph of $\mathcal{G}$ for Q is an undirected graph $G_s = (V_s, E_s, L_s)$, where V_s and E_s are the node and edge set, and L_s is a labeling function. Moreover, (1) each node $v_s \in V_s$ labeled with $L_s(v_s)$ represents a node set $[v_s]$ from $\mathcal{G}$, such that (a) $[v_s]$ is either a keyword node set, or an intermediate node set from $\mathcal{G}$, and (b) the nodes v in $[v_s]$ have the same label $L(v) = L_s(v_s)$. We say v_{s_k} is a *keyword node* for a keyword k, if $[v_{s_k}]$ is a set of keyword nodes of k; (2) For any path ρ_s between keyword nodes v_{s_1} and v_{s_2} of G_s, there exists a path ρ with the same label of ρ_s from v_1 to v_2 in the *union* of the answer graphs in $\mathcal{G}$, where $v_1 \in [v_{s_1}]$, $v_2 \in [v_{s_2}]$. Here the path label in G_s is similarly defined as its counterpart in an answer graph.

Hence, a summary graph G_s never introduces "false" paths by definition: if v_{s_1} and v_{s_2} are connected via a path ρ_s in G_s, it suggests that there is a path ρ of the same label connecting two keyword nodes in $[v_{s_1}]$ and $[v_{s_2}]$, respectively, in the union of the answer graphs. It might, however, "lose" information, *i.e.*, not all the labels of the paths connecting two keyword nodes are preserved in G_s.

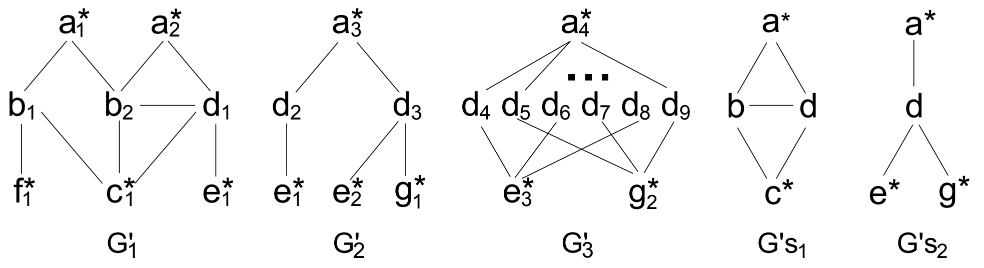


Figure 5.3: Answer graphs and summary graphs.

Example 24: Consider Q and $\mathcal{G}$ from Figure 5.2. One may verify that G_{s_1}, G_{s_2} and G_s are summary graphs of $\mathcal{G}$ for Q. Specifically, (1) the nodes `Jaguar`, `history` and `America` are three keyword nodes in G_{s_1}, and the rest nodes are intermediate ones; (2) G_{s_2} contains a keyword node `Jaguar` which corresponds to keyword nodes {`black jaguar`, `white jaguar`} of the same label `animal` in $\mathcal{G}$. (3) For any path connecting two keyword nodes (*e.g.,* {`Jaguar`, `offer`, `city`, `America`}) in G_{s_1}, there is a path with the same label in the union of G_1 and G_2 (*e.g.,* {$\texttt{Jaguar}_{XK_1}$, $\texttt{offer}_1$, $\texttt{city}_1$, `United States of America`}).

As another example, consider the answer graphs G'_1, G'_2 and G'_3 induced by a keyword query $Q' = \{\texttt{a, c, e, f, g}\}$ in Figure 5.3. Each node a_i (marked with $*$ if it is a keyword node) in an answer graph has a label a and an id a_i, similarly for the rest nodes. One may verify the following. (1) Both G'_{s_1} and G'_{s_2} are summary graphs for the answer graph set $\{G'_1, G'_2\}$; while G'_{s_1} (resp. G'_{s_2}) only preserves the labels of the paths connecting keywords `a` and `c` (resp. `a`, `e` and `g`). (2) G'_{s_2} is not a summary graph for G'_3. Although it correctly suggests the relation between keywords (a, e) and (a, g), it contains a "false" path labeled (e, d, g), while there is no path in G'_3 with the same label between e_3 and g_2. □

Remarks. One can readily extend summary graphs to support directed, edge labeled answer graphs by incorporating edge directions and labels into the path label. We can also extend summary graphs for preserving path labels for each answer graph, instead of for the union of answer graphs, by reassigning node identification to answer graphs.

5.3 Quality Measurement

We next introduce two metrics to measure the quality of summary graphs, based on information coverage and summarization conciseness, respectively. We then introduce a set of summarization problems. To simplify the discussion, we assume that the union of the answer graphs contains keyword nodes for each keyword in Q.

5.3.1 Coverage Measurement

It is recognized that a summarization should summarize as much information as possible, *i.e.,* to maximize the information coverage [52]. In this context, a summary graph should capture the relationship among the query keywords as much as possible. To capture this, we first present a notion of *keywords coverage.*

Keywords coverage. Given a keyword pair (k_i, k_j) (k_i, $k_j \in Q$ and $k_i \neq k_j$) and answer graphs $\mathcal{G}$ induced by Q, a summary graph G_s *covers* (k_i, k_j) if for any path ρ from keyword nodes v_{k_i} to v_{k_j} in the union of the answer graphs in $\mathcal{G}$, there is a path ρ_s in G_s from v_{s_i} to v_{s_j} with the same label of ρ, where $v_{k_i} \in [v_{s_i}]$, $v_{k_j} \in [v_{s_j}]$.

Note that the coverage of a keyword pair is "symmetric" over undirected answer graphs. Given Q and $\mathcal{G}$, if G_s covers a keyword pair (k_i, k_j), it also covers (k_j, k_i).

Coverage ratio. Given a keyword query Q and $\mathcal{G}$, we define the *coverage ratio* α of a summary graph G_s of $\mathcal{G}$ as

$$\alpha = \frac{2 \cdot M}{|Q| \cdot (|Q| - 1)}$$

where M is the total number of the keyword pairs (k, k') covered by G_s. Note that there are in total $\frac{|Q||Q|-1}{2}$ pairs of keywords from Q. Thus, α measures the information coverage of G_s based on the coverage of the keywords.

We refer to as *α-summary graph* the summary graph for $\mathcal{G}$ induced by Q with coverage ratio α. The coverage ratio measurement favors a summary graph that covers more keyword pairs, *i.e.,* with larger α.

Example 25: Consider Q and $\mathcal{G}$ from Figure 5.2. Treating G_{s_1} and G_{s_2} as a single graph G_{s_0}, one may verify that G_{s_0} is a 1-summary graph: for any keyword pair from Q and any path between the keyword nodes in $\mathcal{G}$, there is a path of the same label in G_{s_0}. On the other hand, G_s is a $\frac{1}{3}$ summary graph for Q: it only covers the keyword pairs $(\texttt{Jaguar}, \texttt{America})$. Similarly, one may verify that G'_{s_1} (resp. G'_{s_2}) in Figure 5.3 is a 0.1-summary graph (resp. 0.3-summary graph), for answer graphs $\{G'_1, G'_2, G'_3\}$ and $Q = \{a, c, e, f, g\}$. □

5.3.2 Conciseness Measurement

A summary graph should also be concise, without introducing too much detail of answer graphs, as commonly used in information summarization [52, 137].

Summarization size. We define the size of a summary graph G_s, (denoted as $|G_s|$) as the total number of the nodes and edges it has. For example, the summary graph G_{s_1} and G_{s_2} (Figure 5.2) are of size 12 and 7, respectively. The smaller a summary graph is, the more concise it is.

Putting the information coverage and conciseness measurements together, We say a summary graph G_s is a *minimum* α-summary graph, if for any other α-summary graph G'_s of $\mathcal{G}$ for Q, $|G_s| \leq |G'_s|$.

Example 26: Bisimulation relation [49] constraints the node equivalence via a recursively defined neighborhood label equivalence, which is an overkill for concise summaries over keyword relations. For example, the nodes b_1 and b_2 cannot be represented by a single node as in G_{s_1} via bisimulation (Figure 5.3), due to different neighborhood. One the other hand, error-tolerant [110], structure-based summaries [142] and schema extraction [145] may generate summary graphs with "false paths," such as G'_{s_2} for G'_3. To prevent this, auxiliary structures and parameters are required. In contrast in our work, a summary graph preserves path labels for keywords without any auxiliary structures. □

5.3.3 Summarization Problems

Based on the quality metrics, we next introduce two summarization problems for keyword induced answer graphs. These problems are to find summary graphs with high quality, in terms of information coverage and conciseness.

Minimum α-Summarization. Given a keyword query Q and its induced answer graphs $\mathcal{G}$, and a user-specified coverage ratio α, the *minimum α-summarization* problem, denoted as MSUM, is to find an α-summary graph of $\mathcal{G}$ with minimum size. Intuitively, the problem aims to find the smallest summary graph [137] which can cover the keyword pairs no less than user-specified coverage requirement.

Theorem 9: MSUM *is NP-complete (for decision version) and APX-hard (as an optimization problem).* □

The APX-hard class consists of all problems that cannot be approximated in polynomial time within arbitrary small approximation ratio [149]. We prove the complexity result and provide a heuristic algorithm for MSUM in Section 5.4.

Minimum 1-*summarization.* We also consider the problem of finding a summary graph that covers every pair of keywords (k_i, k_j) ($k_i, k_j \in Q$ and $i \neq j$) as concise as possible, *i.e.,* the *minimum* 1-*summarization problem* (denoted as PSUM). Note that PSUM is a special case of MSUM, by setting $\alpha = 1$. In contrast to MSUM, PSUM is in PTIME.

Theorem 10: *Given Q and $\mathcal{G}$,* PSUM *is in $O(|Q|^2|\mathcal{G}| + |\mathcal{G}|^2)$ time,* i.e., *it takes $O(|Q|^2|\mathcal{G}| + |\mathcal{G}|^2)$ time to find a minimum 1-summary graph, where $|\mathcal{G}|$ is the size of $\mathcal{G}$.* □

We will prove the above result in Section 5.4.

K **Summarization**. In practice, users may expect a set of summary graphs instead of a single one, where each summary graph captures the keyword relationships for a set of "similar" answer graphs in terms of path labels. Indeed, as observed in text summarization (*e.g.,* [52]), a summarization should be able to cluster a set of similar objects.

Given Q, $\mathcal{G}$, and an integer K, the K *summarization* problem (denoted as KSUM) is to find a summary graph set G_S, such that (1) each summary graph $G_{s_i} \in G_S$ is a 1-summary graph of a group of answer graphs $G_{p_i} \subseteq \mathcal{G}$, (2) the answer graph sets G_{p_i} form a K-partition of $\mathcal{G}$, *i.e.,* $\mathcal{G} = \bigcup G_{p_i}$, and $G_{p_i} \cap G_{p_j} = \emptyset$ ($i, j \in [1, K]$, $i \neq j$); and (3) the total size of G_S, *i.e.,* $\sum_{G_{s_i} \in G_S} |G_{s_i}|$ is minimized. The KSUM problem can also be extended to support α-summarization.

The following result tells us that the problem is hard to approximate. We will prove the result in Section 5.5, and provide a heuristic algorithm for KSUM.

Theorem 11: KSUM *is NP-complete and APX-hard.* □

Remarks. The techniques for MSUM and KSUM can be used in a host of applications. (a) The α-summaries from MSUM can be used to suggest (structured) keyword queries [22], as well as graph (pattern) queries [42, 130, 145]. The intermediate nodes in the summaries also benefit reasonable query expansion [125]. (b) In practice the answer graphs can be too many to inspect. The techniques for KSUM naturally serve as post-processing for result summarizations [68]. Better still, KSUM also provides a reasonable clustering for answer graphs [52]. The generated K summaries can further be used for query expansion based on clustered results [92].

While determining the optimal value of α and K remain to be open issues, α can be usually set according to *e.g.*, "budget" of comprehansion [137], and K can be determined following empirical rules [100] or information theory.

5.4 Computing α-Summarization

In this section we investigate the α-summarization problem. We first investigate PSUM in Section 5.4.1, as a special case of MSUM. We then discuss MSUM in Section 5.4.2.

5.4.1 Computing 1-Summary Graphs

To show Theorem 10, we characterize the 1-summary graph with a sufficient and necessary condition. We then provide an algorithm to check the condition in polynomial time. We first introduce the notion of dominance relation.

Dominance relation. The *dominance relation* $R_{\preceq}(k, k')$ for keyword pair (k, k') over an answer graph $G = (V, E, L)$ is a binary relation over the intermediate nodes of G, such that for each node pair $(v_1, v_2) \in R_{\preceq}(k, k')$, (1) $L(v_1) = L(v_2)$, and (2) for any path ρ_1 between keyword node pair v_{k_1} of k and v_{k_2} of k' passing v_1, there is a path ρ_2 with the same label between two keyword nodes v'_{k_1} of k and $v_{k'_2}$ of k' passing v_2. We say v_2 *dominates* v_1 *w.r.t.* (k, k'); moreover, v_1 is *equivalent* to v_2 if they dominate each other. In addition, two keyword nodes are equivalent if they have the same label, and correspond to the same keyword.

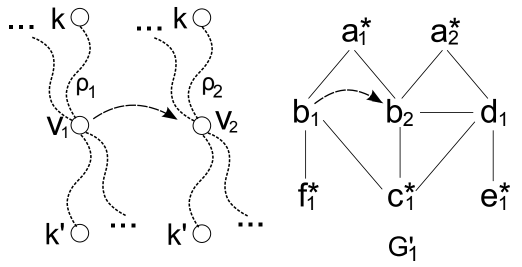


Figure 5.4: Dominance relation.

The dominance relation is as illustrated in Figure 5.4. Intuitively, (1) $R_{\preceq}(k, k')$ captures the nodes that are "redundant" in describing the relationship between a keyword pair (k, k') in G; (2) moreover, if two nodes are equivalent, they play the same "role" in connecting keywords k and k', *i.e.,* they cannot be distinguished in terms of path labels. For example, when the keyword pair (a, c) is considered in G'_1, the node b_1 is dominated by b_2, as illustrated in Figure 5.4.

Remarks. The relation $R_{\preceq}$ is similar to the *simulation* relation [19, 64], which computes node similarity over the entire graph by neighborhood similarity. In contrast to simulation, $R_{\preceq}$ captures dominance relation induced by the paths connecting keyword nodes only, and only consider intermediate nodes. For example, the node b_1 and b_2 is not in a simulation relation in G'_1, unless the keyword pair (a, c) is considered (Figure 5.4). We shall see that this leads to effective summarizations for specified keyword pairs.

Sufficient and necessary condition. We now present the sufficient and necessary condition, which shows the connection between $R_{\preceq}$ and a 1-summary graph.

Proposition 12: *Given Q and $\mathcal{G}$, a summary graph G_s is a minimum 1-summary graph for $\mathcal{G}$ and Q, if and only if for each keyword pair (k, k') from Q, (a) for each intermediate node v_s in G_s, there is a node v_i in $[v_s]$, such that for any other node v_j in $[v_s]$, $(v_j, v_i) \in R_{\preceq}(k, k')$; and (b) for any intermediate nodes v_{s_1} and*

v_{s_2} in G_s with same label and any nodes $v_1 \in [v_{s_1}]$, $v_2 \in [v_{s_2}]$, $(v_2, v_1) \notin R_{\preceq}(k, k')$.

□

Proof sketch: We prove Proposition 12 as follows.

(1) We first proof by contradiction that G_s is a 1-summary graph if and only if Condition (a) holds. Assume G_s is a 1-summary graph while Condition (a) does not hold. Then there exists an intermediate node v_s, and two nodes v_i and v_j that cannot dominate each other. Thus, there must exist two paths in the union of answer graphs as $\rho = \{v_1, \ldots, v_i, v_{i+1}, \ldots, v_m\}$ and $\rho' = \{v'_1, \ldots, v_j, v_{j+1}, \ldots, v_n\}$ with different labels, for a keyword pair (k, k'). Since v_i, v_j is merged as v_s in G_s, there exists, *w.l.o.g.*, a false path in G_s as ρ'' with label $L(v_1) \ldots L(v_i) L(v_{j+1}) \ldots L(v_m)$, which contradicts the assumption that G_s is a 1-summary graph. Now assume Condition (a) holds while G_s is not a 1-summary graph. Then there at least exists a path from keywords k to k' that is not in G_s. Thus, there exists at least an intermediate node v_s on the path with $[v_s]$ in G_s which contains two nodes that cannot dominate each other. This contradicts the assumption that Condition (a) holds.

(2) For the summary minimization, we show that Conditions (a) and (b) together guarantee if there exists a 1-summary G'_s where $|G'_s| \leq |G_s|$, there exists a one to

one function mapping each node (resp. edge) in G'_s to a node (resp. edge) in G_s, *i.e.*, $|G_s| = |G'_s|$. Hence, G_s is a minimum 1-summary graph by definition. $\square$

We next present an algorithm for PSUM following the sufficient and necessary condition, in polynomial time.

Algorithm. pSum has the following two steps.

Initialization. pSum first initializes an empty summary graph G_s. For each keyword pair (k, k') from Q, pSum computes a "connection" graph of (k, k') induced from $\mathcal{G}$. Let G be the union of the answer graphs in $\mathcal{G}$. A connection graph of (k, k') is a subgraph of G induced by (1) the keyword nodes of k and k', and (2) the intermediate nodes on the paths between the keyword nodes of k and those of k'. Once $G_{(k,k')}$ is computed, pSum sets G_s as the union graph of G_s and $G_{(k,k')}$.

Reducing. pSum then constructs a summary graph by removing nodes and edges from G_s. It computes the dominance relation $R_{\preceq}$ by invoking a procedure DomR, which removes the nodes v as well as the edges connected to them, if they are dominated by some other nodes. It next merges the nodes in G_s that have dominate relation (as defined in 12(a)), into a set $[v_s]$, until no more nodes in G_s can be merged. For each set $[v_s]$, a new node v_s as well as its edges connected to other nodes are created. G_s is then updated with the new nodes and edges, and is returned as a minimum 1-summary graph.

Procedure DomR. Similar as the process to compute a simulation [64], DomR extends the process to undirected graphs. For each node v in G_s, DomR initializes a dominant set $[v]$, as $\{v'|L(v') = L(v)\}$. For each edge $(u, v) \in G_s$, it identifies the neighborhood set of u (resp. v) as $N(u)$ (resp. $N(v)$), and removes the nodes that are not in $N(v)$ (resp. $N(u)$) from $[u]$ (resp. $[v]$) (lines 4). Indeed, a node $u' \in [u]$ cannot dominant u if $u' \notin N(v)$, since a path connecting two keyword nodes passing edge (u, v) contains "$L(u)L(v)$" in its label, while for u', such path does not exist. The process repeats until no changes can be made. $R_{\preceq}$ is then collected from the dominant sets and returned.

Analysis. pSum correctly returns a summary graph G_s. Indeed, G_s is initialized as the union of the connection graphs, which is a summary graph. Each time G_s is updated, pSum keeps the invariants that G_s remains to be a summary graph. When pSum terminates, one may verify that the sufficient and necessary condition as in Proposition 12 is satisfied. Thus, the correctness of pSum follows.

It takes $O(|Q|^2|\mathcal{G}|)$ to construct G_s as the union of the connection graphs for each keyword pairs. It takes DomR in total $O(|\mathcal{G}|^2)$ time to compute $R_{\preceq}$. To see this, observe that (a) it takes $O(|\mathcal{G}|^2)$ time to initialize the dominant sets, (b) during each iteration, once a node is removed from $[u]$, it will no longer be put back, *i.e.,* there are in total $|G_s|^2$ iterations, and (c) the checking at line 4 can be done in constant time, by looking up a dynamically maintained map recording

$|[u] \setminus N(v)|$ for each edge (u, v), leveraging the techniques in [64]. Thus, the total time complexity of pSum is in $O(|Q|^2|\mathcal{G}| + |\mathcal{G}|^2)$. Hence, Theorem 10 follows.

Table 5.1: Summarization examples.

Nodes in G_s	dominance sets
`offer`	$\{\texttt{offer}_i\}(i \in [1, m])$
`city`	$\{\texttt{city}_i\}(i \in [1, k]), \{\texttt{city}_j\}(j \in [k+1, p])$
`company`	$\{\texttt{company}_i\}(i \in [1, l-1]), \{\texttt{company}_l\}$

Example 27: Recall the query Q and the answer graph set $\mathcal{G}$ in Figure 5.2. The algorithm pSum constructs a minimum 1-summary graph G_s for $\mathcal{G}$ as follows. It initializes G_s as the union of the connection graphs for the keyword pairs in Q, which is the union graph of G_1, G_2 and G_3. It then invokes procedure DomR, which computes dominance sets for each intermediate node in G_s, partly shown as follows (k <p). pSum then reduces G_s by removing dominated nodes and merging equivalent nodes until no change can be made. For example, (1) $\texttt{company}_x$ ($x \in [1, l-1]$) are removed, as all are dominated by $\texttt{company}_l$; (2) all the `offer` nodes are merged as a single node, as they dominate each other. G_s is then updated as the union of G_{s_1} and G_{s_2} (Figure 5.2). □

From Theorem 10, the result below immediately follows.

Corollary 13: *It is in $O(|S||\mathcal{G}| + |\mathcal{G}|^2)$ to find a minimum 1-summary graph of $\mathcal{G}$ for a given keyword pair set S.* □

Indeed, pSum can be readily adapted for specified keyword pair set S, by specifying G_s as the union of the connection graphs induced by S (line 4). The need to find 1-summary graphs for specified keyword pairs is evident in the context of *e.g.,* relation discovery [45], where users may propose specified keyword pairs to find their relationships in graph data.

5.4.2 Minimum α-Summarization

We next investigate the MSUM problem: finding the minimum α-summarization. We first prove Theorem 9, *i.e.,* the decision problem for MSUM is NP-complete. Given Q, a set of answer graphs $\mathcal{G}$ induced by Q, a coverage ratio α, and a size bound B, the decision problem of MSUM is to determine if there exists a α-summary graph G_s with size no more than B. Observe that MSUM is equivalent to the following problem (denoted as MSUM*): find an m-element set $S_m \subseteq S$ from a set of keyword pairs S, such that $|G_s| \leq B$, where (a) $m = \alpha \cdot \frac{|Q||Q-1|}{2}$, (b) $S = \{(k, k')|k, k' \in Q,\ k \neq k'\}$, and (c) G_s is the minimum 1-summary graph for $\mathcal{G}$ and S_m. It then suffices to show MSUM* is NP-complete.

Complexity. We show that MSUM* is NP-complete as follows. (1) MSUM* is in NP, since there exists a polynomial time algorithm to compute G_s for a keyword pair set S, and determine if $|G_s| \leq B$ (Corollary 13). (2) To show the lower bound, we construct a reduction from the maximum coverage problem, a known

NP-complete problem [48]. Given a set X and a set T of its subsets $\{T_1, \ldots, T_n\}$, as well as integers K and N, the problem is to find a set $T' \subseteq T$ with no more than K subsets, where $|\bigcup T' \cap X| \geq N$. Given an instance of maximum coverage, we construct an instance of MSUM^* as follows. (a) For each element $x_i \in X$, we construct an intermediate node v_i. (b) For each set $T_j \in T$, we introduce a keyword pair (k_{T_j}, k'_{T_j}), and construct an answer graph G_{T_j} which consists of edges (k_{T_j}, v_i) and (v_i, k'_{T_j}), for each v_i corresponding to $x_i \in T_j$. We set S as all such (k_{T_j}, k'_{T_j}) pairs. (c) We set $m = |T|$-K, and $B = |X|$-N. One may verify that there exists at most K subsets that covers at least N elements in X, if and only if there exists a 1-summary graph that covers at least $|S|$-K keyword pairs, with size at most $2 * (|X|$-$N+ m)$. Thus, MSUM^* is NP-hard. Putting (1) and (2) together, MSUM^* is NP-complete.

The APX-hardness can be proved by constructing an approximation ratio-preserving reduction [149] from the weighted maximum coverage problem, a known APX-hard problem, via a similar transformation as discussed above.

The above analysis completes the proof of Theorem 9.

The APX-hardness of MSUM indicates that it is unlikely to find a polynomial-time algorithm for MSUM with every fixed approximation ratio [149]. Instead, we resort to an efficient heuristic algorithm, mSum.

A greedy heuristic algorithm. Given Q and $\mathcal{G}$, mSum (1) dynamically maintains a set of connection graphs $\mathcal{G}_C$, and (2) greedily selects a keyword pair (k, k') and its connection graph G_c, such that the following "merge cost" is minimized:

$$\delta_{r(\mathcal{G}_C, G_c)} = |G_{s(\mathcal{G}_C \cup \{G_c\})}| - |G_{s(\mathcal{G}_C)}|$$

where $G_{s(\mathcal{G}_C \cup \{G_c\})}$ (resp. $G_{s(\mathcal{G}_C)}$) is the 1-summary graph of the answer graph set $\mathcal{G}_C \cup \{G_c\}$ (resp. $(\mathcal{G}_C)$). Intuitively, the strategy always chooses a keyword pair with a connection graph that "minimally" introduces new nodes and edges to the dynamically maintained 1-summary graph.

The algorithm mSum is shown in Alg. 8. It first initializes a summary graph G_s (as empty), as well as an empty answer graph set $\mathcal{G}_C$ to maintain the answer graphs to be selected for summarizing (line 1). For each keyword pair (k, k'), it computes the connection graph $G_{c(k,k')}$ from the union of the answer graphs in $\mathcal{G}$, and puts $G_{c(k,k')}$ to $\mathcal{G}_C$ (line 2-3). This yields a set $\mathcal{G}_C$ which contains in total $O(\frac{|Q|(|Q|-1)}{2})$ connection graphs. It then identifies a subset of connection graphs in $\mathcal{G}$ by greedily choosing a connection graph G_c that minimizes a dynamically updated merge cost $\delta_{r(\mathcal{G}_C, G_c)}$, as remarked earlier (line 5). In particular, we use an efficiently estimated merge cost, instead of the accurate cost via summarizing computation (as will be discussed). Next, it either computes G_s as a 1-summary graph for $G_{c(k,k')}$ if G_s is $\emptyset$, by invoking pSum (line 6), or updates G_s with the newly selected G_c, by invoking a procedure merge (line 7). G_c is then removed from G_S

(line 8), and the merge cost of all the rest connection graphs in $\mathcal{G}_C$ are updated according to the selected connection graphs (line 10-11). The process repeats until $m = \lceil \frac{\alpha|Q|(|Q|-1)}{2} \rceil$ pairs of keywords are covered by G_s, *i.e.,* m connection graphs are processed (line 9). The updated G_s is returned (line 12).

Procedure. The procedure merge is invoked to update G_s upon new connection graphs. It takes as input a summary graph G_s and a connection graph G_c. It also keeps track of the union of the connection graphs G_s corresponds to. It then updates G_s via the following actions: (1) it removes all the nodes in G_c that are dominated by the nodes in itself or the union graph; (2) it identifies equivalent nodes from the union graph and G_c (or have the same identification); (3) it then splits node v_s in G_s if $[v_s]$ contains two nodes that cannot dominate each other, or merge all the nodes in G_s that have dominance relation. G_s is then returned if no more nodes in G_s can be further updated.

Optimization. The merge cost (line 5) of mSum takes in total $O(|\mathcal{G}|^2)$ time. To reduce the merging time, we efficiently *estimate* the merge cost. Given $\mathcal{G}$, a neighborhood containment relation R_r captures the *containment* of the label sets from the neighborhood of two nodes in the union of the graphs in $\mathcal{G}$. Formally, R_r is a binary relation over the nodes in $\mathcal{G}$, such that a pair of nodes $(u, v) \in R_r$ if and only if they have the same label, and for each neighbor u' of u, there is a neighbor v' of v with the same label of u'. Denoting as $D(R_r)$ the union of the edges attached to the node u, for all $(u, v) \in R_r$, we have the following result.

Lemma 14: *For a set of answer graphs* $\mathcal{G}$ *and its* 1*-summary* G_s, $|\mathcal{G}| \geq |G_s| \geq |\mathcal{G}|$ - $|R_r(\mathcal{G})|$ - $|D(R_r)|$. □

To see this, observe the following. (1) $|\mathcal{G}|$ is clearly no less than $|G_s|$. (2) Denote G as the union of the answer graphs in $\mathcal{G}$, we have $|G_s| \geq |\mathcal{G}|$ - $|R_{\prec}(G)|$ - $|D(R_{\prec}|)$, where $R_{\prec}(G)$ is the dominance relation over G, and $D(R_{\prec})$ is similarly defined as $D(R_r)$. (3) For any $(u, v) \in R_{\prec}(G)$, (u, v) is in $R_r(\mathcal{G})$. In other words, $|R_{\prec}(G)| \leq |R_r(\mathcal{G})|$, and $|D(R_{\prec})| \leq |D(R_r)|$. Putting these together, the result follows.

The above result tells us that $|\mathcal{G}|$ - $|R_r(\mathcal{G})|$ - $|D(R_r)|$ is a lower bound for G_s of $\mathcal{G}$. We define the merge cost $\delta_{r(\mathcal{G}_C, G_c)}$ as $|\mathcal{G}|$ - $|R_r(\mathcal{G})|$ - $|D(R_r)|$ - $|G_{s(\mathcal{G}_C)}|$. Using an index structure that keeps track of the neighborhood labels of a node in $\mathcal{G}$, $\delta_{r(\mathcal{G}_C, G_c)}$ can be evaluated in $O(|\mathcal{G}|)$ time.

Analysis. The algorithm mSum correctly outputs an α-summary graph, by preserving the following invariants. (1) During each operation in merge, G_s is correctly maintained as a minimum summary graph for a selected keyword pair set. (2) Each time a new connection graph is selected, G_s is updated to a summary graph that covers one more pair of keywords, until m pairs of keywords are covered by G_s.

For complexity, (1) it takes in total $O(m \cdot |\mathcal{G}|)$ time to induce the connection graphs (line 1-3); (2) the While loop is conducted m times (line 4); In each loop, it takes $O((|\mathcal{G}|^2)$ time to select a G_c with minimum merge cost, and to update G_s (line 7). Thus, the total time complexity is $O(m|\mathcal{G}|^2)$. Note that in practice m is typically small.

Example 28: Recall the query $Q' = \{a, c, e, f, g\}$ and the answer graph set $\mathcal{G} = \{G'_1, G'_2\}$ in Figure 5.3. There are in total 10 keyword pairs. To find a minimum 0.3-summary graph, MSUM starts with a smallest connection graph induced by *e.g.,* (a, g), and computes a 1-summary graph as G''_{s_1} shown in Figure 5.5. It then identifies the connection graph G_c induced by (e, g), with least merge cost. Thus, G_{s_1} is updated to G_{s_2} by merging G_c, with one more node e_2 and edge (d_3, e_2) inserted. It then updates the merge cost, and merges the connection graph of (a, e) to G''_{s_2} to form G''_{s_3}, by invoking merge. merge identifies that in G''_{s_3} (1) a_1 is dominated by a_2, (2) the two e^*_1 nodes refer to the same node. Thus, it removes a_1 and merges e^*_1, updating G''_{s_3} to G''_s, and returns G''_s as a minimum 0.3-summary graph. □

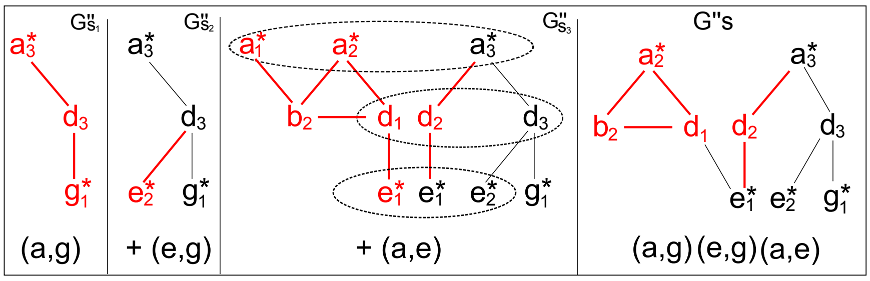


Figure 5.5: Computing minimum α-summary graph.

5.5 Computing K Summarizations

In this section we study how to construct K summary graphs for answer graphs, *i.e.,* the KSUM problem.

Complexity. We start by proving Theorem 11 (Section 5.2). Given Q, $\mathcal{G}$, an integer K and a size bound B, the decision problem of KSUM asks if there exists a K-partition of $\mathcal{G}$, such that the sum of the 1-summary graph for each partition is no more than B. The problem is in NP. To show the lower bound, we construct a reduction from the graph decomposition problem shown to be NP-hard [118]. Given a complete graph G where each edge is assigned with an integer weight, the problem is to identify K' partitions of edges, such that the sum of the maximum edge weight in each partition is no greater than a bound W. Given an instance of the problem, (a) We identify the maximum edge weight w_m in G, and construct w_m intermediate nodes $V_I = \{v_1, \dots, v_{w_m}\}$, where each intermediate node has a distinct label. (b) For each edge in G with weight w_i, we construct an answer graph with two fixed keyword nodes k_1, k_2 and edges (k_1, v_j) and (v_j, k_2), where $v_j \in V_I$, and $j \in [1, w_i]$. (c) We set $K = K'$, and $B = W$. One may verify that if a K'-partition of edges in G has a total weight within W, then there exists a K-partition of $\mathcal{G}$ with total summary size within $3W + 2K$, and vice versa. Thus, KSUM is NP-hard. This verifies that KSUM is NP-complete.

The APX-hardness of the K summarization problem can be shown similarly, by conducting an approximation preserving reduction from the graph decomposition problem, which is shown to be APX-hard [118]. The above analysis completes the proof of Theorem 11.

We next present a heuristic algorithm for KSUM. We first introduce a distance measure for answer graphs.

Graph distance metric. Given two answer graphs G_1 and G_2, we introduce a similarity function $F(G_1, G_2)$ as $\frac{|R_r(G_{1,2})|+|D(R_r)|}{|G_1|+|G_2|}$, where $G_{1,2}$ is the union of G_1 and G_2, and $R_r(G_{1,2})$ and $D(R_r)$ are as defined in Section 5.4. Intuitively, the similarity function F captures the similarity of two answer graphs, by measuring "how well" a summary graph may compress the union of the two graphs [52]. Thus a distance function $\delta(G_1, G_2)$ can be defined as 1 - $F(G_1, G_2)$.

Based on the distance measure, we propose an algorithm, kSum, which partitions $\mathcal{G}$ into K clusters G_P, such that the total set distance $F(G_{p_i})$ in each cluster G_{p_i} is minimized. This intuitively leads to K small summary graphs.

Algorithm. The algorithm kSum works similarly as a K-center clustering process [25]. It first initializes a set G_P to maintain the partition of $\mathcal{G}$, an answer graph set $\mathcal{G}_K$ with randomly selected K answer graphs from $\mathcal{G}$ as K "centers," and a summary set $\mathcal{G}_S$ to keep record of K 1-summary graphs, each corresponds to a cluster G_{p_i} in G_P; in addition, the total difference θ is initialized as a large number, *e.g.,* $K\ |\mathcal{G}|^2$. It then iteratively refines the partition G_P as follows. (1) For each answer graph $G \in \mathcal{G}$, it selects the "center" graph G_{c_j} which minimizes $\delta(G, G_{c_j})$, *i.e.,* is the closest one to G, and extends the cluster G_{p_j} with G. (2) The updated clusters G_P forms a partition of $\mathcal{G}$. For each cluster $G_{p_i} \in G_P$, a new "center" graph G'_{c_i} is selected, which minimizes the sum of the distance from G'_{c_i} to all the rest graphs in G_{p_i}. The newly identified K graphs replace the original graphs in $\mathcal{G}_K$. (3) The overall distance $\theta{=}\sum_i \sum_{G\in G_{p_i}} \delta(G, G_{c_i})$ is recomputed for G_P. kSum repeats the above process until θ converges. It then computes and returns K 1-summary graph by invoking the algorithm pSum for each cluster $G_{p_i} \in G_P$.

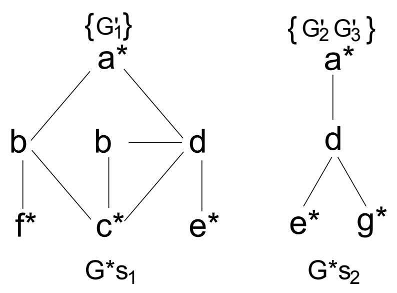


Figure 5.6: Summary graphs for a 2-partition.

Example 29: Recall the answer graph set $\mathcal{G}$ =$\{G'_1, G'_2, G'_3\}$ in Figure 5.3. Let K= 2. The algorithm pSum first selects two graphs as "center" graphs, *e.g.,* G'_1 and G'_3, and computes the distance between the graphs. One may verify that $\delta(G'_1, G'_2) > \delta(G'_2, G'_3)$. Thus, G'_2 and G'_3 are much "closer," and are grouped together to form a cluster. This produces a 2-partition of $\mathcal{G}$ as $\{\{G'_1\}, \{G'_2, G'_3\}\}$. The 1-summary graphs are then computed for each cluster. pSum finally returns G'_{s_1} and G'_{s_2} as the minimized 2 1-summary graphs, with total size 22 (Figure 5.6). □

Analysis. The algorithm kSum correctly computes K 1-summary graphs for a K-partition of $\mathcal{G}$. It heuristically identifies K clusters with minimized total distance of each answer graph in the cluster to its "center" graph. kSum can also be used to compute K α-summary graphs.

For complexity, (1) it takes kSum $O(\mathcal{G})$ time for initialization; (2) the clustering phase takes in total $O(I \cdot K \cdot |G_m|^2)$ time, where I is the number of iterations, and G_m is the largest answer graph in $\mathcal{G}$; and (3) the total time of summarization is in $O(|Q|^2||\mathcal{G}| + |\mathcal{G}|^2)$. In our experiments, we found that I is typically small (no more than 3).

5.6 Experimental Evaluation

5.6.1 Experimental Settings

Datasets. We use the following three real-life datasets in our tests. *DBLP* (`http://dblp.uni-trier.de/xml/`), a bibliographic dataset with in total 2.47 million nodes and edges, where (a) each node has a type from in total 24 types (*e.g.,*'paper', 'book', 'author'), and a set of attribute values (*e.g.,*'network', 'database', etc), and (b) each edge denotes *e.g.,* authorship or citation. We also employ *DBpedia* and *YAGO* as described in Section 2.8.

Keyword queries. (1) For *DBLP*, we select 5 common queries as shown in Table 5.2. The keyword queries are for searching information related with various topics or authors. For example, Q_1 is to search the mining techniques for temporal graphs. (2) For *DBpedia* and *YAGO*, we design 6 query templates Q_{T_1} to Q_{T_6},

Query	Keywords	card($\mathcal{G}$)	$\overline{\lvert V\rvert}, \overline{\lvert E\rvert}$
Q_1	mining temporal graphs	355	(5,6)
Q_2	david parallel computing ACM	1222	(5,4)
Q_3	distributed graphs meta-data integration	563	(5,5)
Q_4	improving query uncertain database conference	1617	(9,14)
Q_5	keyword search algorithm evaluation XML conference	7635	(7,8)

Table 5.2: Keywords queries for *DBLP*.

each consists of *type* keywords and *value* keywords. The *type* keywords are taken from the type information in *DBpedia* (*resp. YAGO*), *e.g.,* country in Q_{T_5}, and the value keywords are from the attribute values of a node, *e.g.,United States* in Q_{T_2}. Each query template Q_{T_i} is then extended to a set of keyword queries (simply denoted as Q_{T_i}), by keeping all the value keywords, and by replacing some type keywords (*e.g.,*place) with a corresponding value (*e.g.,America*). Table 5.3 shows the query templates Q_T and the total number of its corresponding queries $|Q_T|$. For example, for Q_{T_1}, 136 keyword queries are generated for *DBpedia*. One such query is {'Jaguar', 'America'}.

Answer graph generator. We generate a set of answer graphs $\mathcal{G}$ for each keyword query, leveraging [72, 86]. Specifically, (1) the keyword search algorithm in [72] is used to produce a set of trees connecting all the keywords, and (2)

Query	Keywords template	$\|Q_T\|$	$\overline{\mathsf{card}(\mathcal{G})}$	$\overline{\|V\|}, \overline{\|E\|}$
Q_{T_1}	*Jaguar* place	136	75	(5,7)
Q_{T_2}	*united_states* politician award	235	177	(6,7)
Q_{T_3}	album music genre *american_music_awards*	168	550	(11,25)
Q_{T_4}	fish bird mammal protected_area *north_american*	217	1351	(12,24)
Q_{T_5}	player club manager league city country	52	1231	(17,28)
Q_{T_6}	actor film award company *hollywood*	214	1777	(12,27)

Table 5.3: Keywords queries and the answer graphs for *DBpedia* and *YAGO*.

the trees are expanded to a graph containing all the keywords, with a bounded diameter 5, using the techniques in [86]. Table 5.2 and Table 5.3 report the average number of the generated answer graphs $\mathsf{card}(\mathcal{G})$ and their average size, for *DBLP* and *DBpedia*, respectively. For example, for Q_{T_3}, an answer graph has 11 nodes and 25 edges (denoted as $(11, 25)$) on average. For *YAGO*, $\mathsf{card}(\mathcal{G})$ ranges from 200 to 2000, with answer graph size from $(5, 7)$ to $(10, 20)$. On the other hand, various methods exist *e.g.,* top-k graph selection [145], to reduce possibly large answer graphs.

Implementation. We implemented the following algorithms in Java: (1) pSum, mSum and kSum for answer graph summarization; (2) SNAP [142] to compare with pSum, which generates a summarized graph for a single graph, by grouping nodes such that the pairwise group connectivity strength is maximized; (3) kSum $_{td}$, a revised kSum using a top-down strategy: (a) it randomly selects two answer graphs G_1 and G_2, and constructs 2 clusters by grouping the graphs that are close to G_1 (resp. G_2) together; (b) it then iteratively splits the cluster with larger total inter-cluster distance to two clusters by performing (a), until K clusters are constructed, and the K summary graphs are computed.

All experiments were run on a machine with an Intel Core2 Duo 3.0GHz CPU and 4GB RAM, using Linux. Each experiment was run 5 times and the average is reported here.

5.6.2 Performance on Real-life Datasets

Exp-1: Effectiveness of pSum. We first evaluate the effectiveness of pSum. To compare the effectiveness, we define the *compression ratio* cr of a summarization algorithm as $\frac{|G_s|}{|\mathcal{G}|}$, where $|G_s|$ and $|\mathcal{G}|$ are the size of the summary and answer graphs. For pSum, G_s refers to the 1-summary graph for $\mathcal{G}$ and Q. Since SNAP is not designed to summarize a set of graphs, we first union all the answer graphs in $\mathcal{G}$ to produce a single graph, and then use SNAP to produce a summarized graph G_s. To guarantee that SNAP preserves path information between keywords, we carefully selected parameters *e.g.,* participation ratio [142]. We verify the effectiveness by comparing cr of pSum with that of SNAP.

Fixing the query set as in Table 5.2, we compared cr of pSum and SNAP over *DBLP*. Figure 5.7(a) tells us the following. (a) pSum generates summary graphs much smaller than the original answer graph set. For example, cr of pSum is only 7% for Q_2, and is on average 23%. (b) pSum generates much smaller summary graphs than SNAP. For example, for Q_2 over *DBLP*, the G_s generated by pSum reduces the size of its counterparts from SNAP by 67%. On average, pSum outperforms SNAP by 50% over all the datasets. While SNAP may guarantee path preserving via carefully set parameters, it cannot identify dominated nodes, thus produces larger G_s.

Using Q_{T_i} ($i \in [1, 6]$), we comapred cr of pSum and SNAP over *DBpedia* (Figure 5.7(b)) and *YAGO* (Figure 5.7(c)). (1) pSum produces summaries on average 50% (resp. 80%) smaller of the answer graphs, and on average 62% (resp. 72%) smaller than their counterparts generated by SNAP over *DBpedia* (resp. *YAGO*). (2) For both algorithms, cr is highest over *DBpedia*. The reason is that *DBpedia* has more node labels than *DBLP*, and the answer graphs from *DBpedia* are denser (Table 5.3). Hence, fewer nodes can be removed or grouped in the answer graphs for *DBpedia*, leading to larger summaries. To further increase the compression ratio, one can resort to α-summarization with information loss.

Exp-2: Effectiveness of mSum. In this set of experiments, we verify the effectiveness of mSum. We compare the average size of α-summary graphs by mSum (denoted as $|G_s^\alpha|$) with that of 1-summary graphs by pSum (denoted as $|G_s|$). Using real-life datasets, we evaluated $\frac{|G_s^\alpha|}{|G_s|}$ by varying α.

Fixing the keyword query set as $\{Q_3, Q_4, Q_5\}$, we show the results over *DBLP* in Figure 5.7(d). (1) $|G_s^\alpha|$ increases for larger α. Indeed, the smaller coverage ratio a summary graph has, the fewer keyword pair nodes and the paths are summarized, which usually reduce $|G_s^\alpha|$ and make it more compact. (2) The growth of $|G_s^\alpha|$ is slower for larger α. This is because new keyword pairs are more likely to have already been covered with the increment of α. Figure 5.7(e) and Figure 5.7(f) illustrate the results over *DBpedia* and *YAGO* using the query templates $\{Q_{T_4}, Q_{T_5}, Q_{T_6}\}$ (Table 5.3). The results are consistent with Figure 5.7(d).

We also evaluated the *recall* merit of mSum as follows. Given a keyword query Q, we denote the recall of mSum as $\frac{|P'|}{|P|}$, where P (resp. P') is the set of path labels between the keyword nodes of k and k' in $\mathcal{G}$ (resp. α-summary graph by mSum), for all $(k, k') \in Q$. Figures 5.7(g), 5.7(h) and 5.7(i) illustrate the results over the three real-life datasets. The recall increases with larger α, since more path labels are preserved in summary graphs, as expected. Moreover, we found that mSum covers on average more than 85% path labels for all keyword pairs over *DBLP*, even when $\alpha = 0.6$.

In addition, we compared the performance of mSum with an algorithm that identifies the minimum summary graph by exhaust searching. Using *DBpedia* and its query templates, and varying α from 0.1 to 1 (we used pSum when $\alpha = 1.0$), we found that mSum always identifies summary graphs with size no larger than 1.07 times of the minimum size.

Exp-3: Effectiveness of kSum. We next evaluate the effectiveness of kSum, by evaluating the *average compression ratio*, $\mathsf{cr}_K = \frac{1}{K}\sum_{i=1}^{K} \frac{|G_{s_i}|}{|G_{p_i}|}$ for each cluster G_{p_i} and its corresponding 1-summary graph G_{s_i}.

Fixing the query set $\{Q_3, Q_4, Q_5\}$ and varying K, we tested cr_K over *DBLP*. Figure 5.7(j) tells us the following. (1) For all queries, cr_K first decreases and then increases with the increase of K. This is because a too small K induces large clusters that contain many intermediate nodes that are not dominated by any node, while a too large K leads to many small clusters that "split" similar intermediate nodes. Both cases increase cr_K. (2) cr_K is always no more than 0.3, and is also smaller than its counterpart of pSum in Figure 5.7(a). By using kSum, each cluster G_{p_i} contains a set of similar answer graphs that can be better summarized.

The results in Figure 5.7(k) and 5.7(l) are consistent with their counterparts in Figure 5.7(a). In addition, cr_K is in general higher in *DBpedia* than its counterparts over *DBLP* and *YAGO*. This is also consistent with the observation in Exp-1.

The space cost of the algorithms is mainly on storing answer graphs and dominance relations. In general, pSum takes at most 100M over *DBLP* and *YAGO*, which is less than 1% of the cost for storing the original data graphs. The space cost of mSum and kSum are similar to that of pSum.

5.7 Related Work

Graph compression and summarization. Graph summarization is to (approximately) describe graph data with small amount of information. (1) Graph compression [110] uses MDL principle to compress graphs with bounded error. However, the goal is to reduce space cost while the original graph can be restored, by using auxiliary structures as "corrections." (2) Summarization techniques are proposed based on (a) bisimulation equivalence relation [104], or (b) relaxed bisimulation relation that preserves paths with length up to K [76, 104]. Simulation based minimization [19] reduces a transition system based on simulation equivalence relation. These work preserve paths for every pair of nodes, *i.e.,* all-pair connectivity, which can be too restrictive to generate concise summaries for keyword queries. (3) Summary techniques in [142, 165] enable flexible summarization over graphs with multiple node and edge attributes, while the path information is approximately preserved, controlled by additional parameters, *e.g.,* participation ratio [142].

In contrast to these work, we find concise summaries that preserve relationships among keywords, rather than all-pair connectivity [76,104] or entire original graph [110]. Moreover, in contrast to [110,142], these summaries require no auxiliary structure for preserving the relationships.

Graph clustering. A number of graph clustering approaches have also been proposed to group similar graphs [4]. As remarked earlier, these techniques are not query-aware, and may not be directly applied for summarizing query results as graphs [90]. In contrast, we propose algorithms to (1) group answer graphs in terms of a set of keywords, and (2) find best summaries for each group.

Result Summarization. Result summarization over relational databases and XML are proposed to help users understand the query results. [68] generates summaries for XML results as trees, where a snippet is produced for each result tree. This may produce snippets with similar structures that should be grouped for better understanding [90]. To address this issue, [91] clusters the query results based on the classification of their search predicates. Our work differs in that (1) we generate summaries for and as general graphs rather than trees [68], (2) we study how to summarize connections “induced” by keywords, while the main focus of [91] is to identify proper return nodes.

5.8 Summary

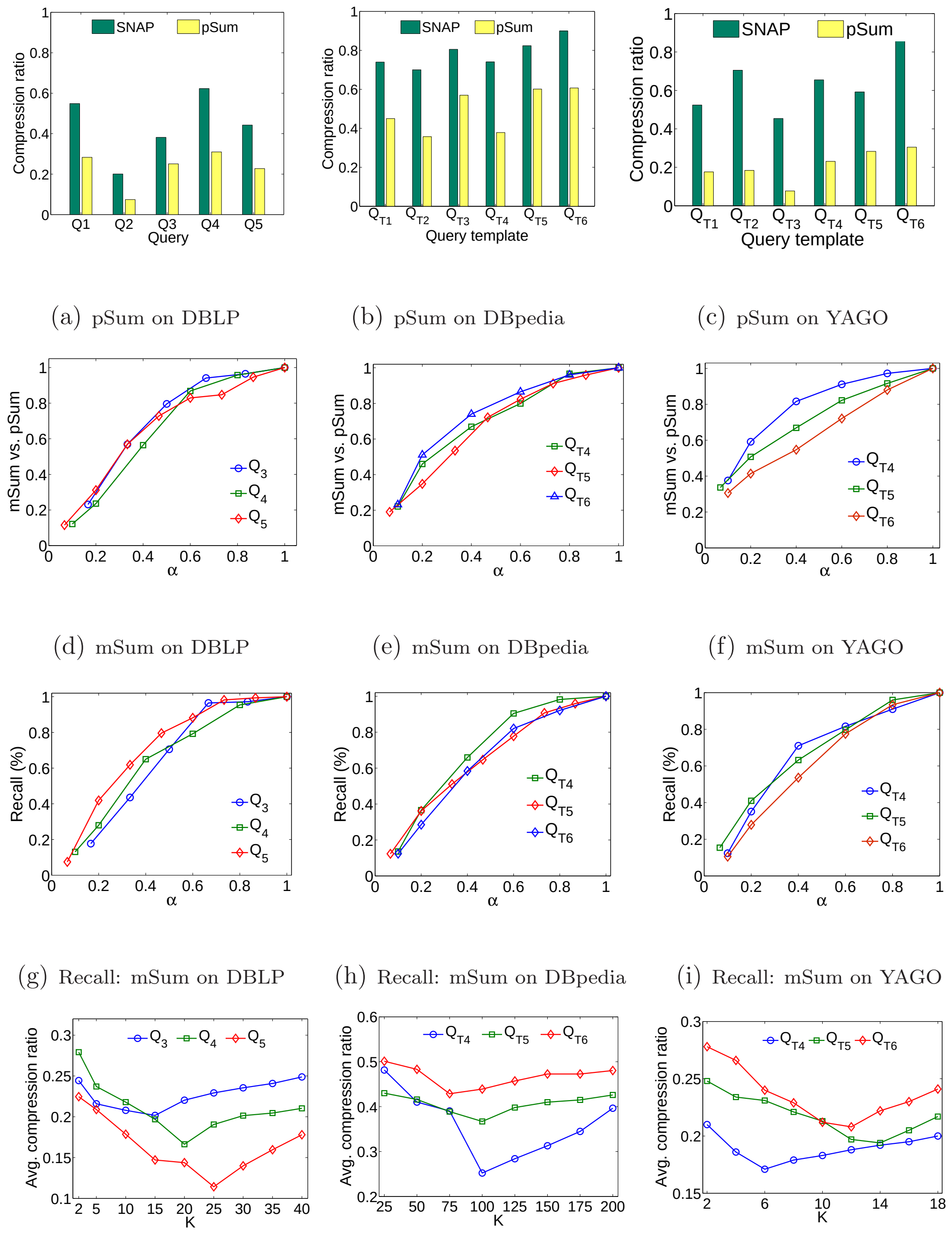
We have developed summarization techniques for keyword search in graph data. By providing a succinct summary of answer graphs induced by keyword queries, these techniques can improve query interpretation and result understanding. We have proposed a new concept of summary graphs and their quality metrics. Three summarization problems were introduced to find the best summarizations with minimum size. We established the complexity of these problems, which range from PTIME to NP-complete. We proposed exact and heuristic algorithms to find the best summarizations. As experimentally verified, the proposed summarization methods effectively compute small summary graphs for capturing keyword relationships in answer graphs.

Algorithm 8 Algorithm mSum

Input: Q, answer graphs $\mathcal{G}$, coverage ratio α;

Output: α-summary graph G_s;

```
1: initialize $G_s$; Set $\mathcal{G}_C := \emptyset$;
2: for each pair $(k, k')$ where $k$, $k' \in Q$ do
3:     compute connection graph $G_{c(k,k')}$; $\mathcal{G}_C := \mathcal{G}_C \cup \{G_{c(k,k')}\}$;
4: end for
5: while $G_S \neq \emptyset$ do
6:     for each $G_{c(k,k')} \in \mathcal{G}_C$ with minimum merge cost do
7:         if $G_s = \emptyset$ then $G_s :=$ pSum$((k,k'), \mathcal{G})$;
8:         else merge$(G_s, G_{c(k,k')})$;
9:         $\mathcal{G}_C := \mathcal{G}_C \setminus \{G_{c(k,k')}\}$;
10:        if $m$ connection graphs are merged then break;
11:        for each $G_c \in \mathcal{G}_C$ do update merge cost of $G_c$;
12:    end for
13: end while
```



(a) pSum on DBLP (b) pSum on DBpedia (c) pSum on YAGO

(d) mSum on DBLP (e) mSum on DBpedia (f) mSum on YAGO

(g) Recall: mSum on DBLP (h) Recall: mSum on DBpedia (i) Recall: mSum on YAGO

(j) kSum on DBLP (k) kSum on DBpedia (l) kSum on YAGO

Figure 5.7: Evaluation on summarization.

Chapter 6

Distributed Graph Processing

Scalable processing of large graphs requires careful partitioning and distribution of graphs across clusters. In this chapter, we investigate the problem of managing large-scale graphs in clusters and study access characteristics of local graph queries such as breadth-first search, random walk, and SPARQL queries, which are popular in real applications. These queries exhibit strong access locality, and therefore require specific data partitioning strategies.

In this work, we propose a Self Evolving Distributed Graph Management Environment (Sedge), to minimize inter-machine communication during graph query processing in multiple machines. In order to improve query response time and throughput, Sedge introduces a two-level partition management architecture with complimentary primary partitions and dynamic secondary partitions. These two kinds of partitions are able to adapt in real time to changes in query workload. Sedge also includes a set of workload analyzing algorithms whose time complexity is linear or sublinear to graph size. Empirical results show that it significantly improves distributed graph processing on today's commodity clusters.

6.1 Introduction

The graphs of interest are often massive with millions, even billions of vertices, making common graph operations computationally intensive. In the presence of data objects associated with vertices, it is clear that graph data can easily scale up to terabytes in size. Moreover, with the advance of the *Semantic Web*, efficient management of massive *RDF* data is becoming increasingly important as Semantic Web technology is applied to real-world applications [3,9]. The recent *Linked Open Data* project has published more then 20 billion RDF triples [61]. Although the RDF data is generally represented in triples, the data inherently presents graph structure and is therefore interlinked. Not surprisingly, the scale and the flexibility rise to the major challenges to the RDF graph management.

The massive scale of graph data easily overwhelms memory and computation resources on commodity servers. Yet online services must answer user queries on these graphs in near real time. In these cases, achieving fast query response time and high throughput requires partitioning/distributing and parallel processing of graph data across large clusters of servers. An appealing solution is to divide a graph into smaller partitions that have minimum connections between them, as adopted by Pregel [98] and SPAR [120]. As long as the graph is clustered to similar-size partitions, the workload of machines holding these partitions will be quite balanced. However, the assumption becomes invalid for *local graph queries* when they are concentrated on a subset of vertices (hotspots), e.g., find/aggregate the attributes of h-hop neighbors around a vertex, calculate personalized PageRank [71], perform a random walk starting at a vertex, and calculate hitting time. When these queries are not uniformly distributed or hitting partition boundaries, we will either have an imbalance of workload or intensive inter-machine communications. A good graph partition management policy should consider these situations and adapt dynamically to changing workload.

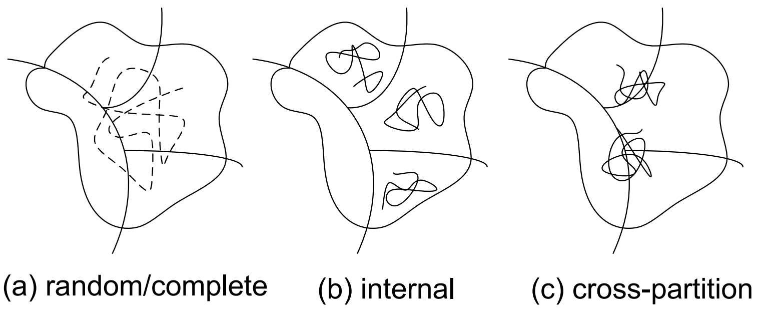


Figure 6.1: Distributed query access pattern.

There could be three kinds of query workload in graphs. For random access or complete traversal of an entire graph shown in Figure 6.1(a), a static balanced partition scheme might be the best solution. For queries whose access is bounded by partition boundaries, as shown in Figure 6.1(b), they shall be served efficiently by the balanced partition scheme too. However, if there are many graph queries crossing the partition boundaries shown in Figure 6.1(c), the static partition scheme might fail due to inter-machine communications. *One partition scheme cannot fit all.* Instead, one shall generate multiple partitions with complementary boundaries or new partitions on-the-fly so that these queries can be answered efficiently.

Graph partitioning is a hard and old problem, which has been extensively studied in various communities since 1970s [74, 78]. Graph partitioning is also widely used in parallel computing (e.g., [62]). The best approaches often depend on the properties of the graphs and the structure of the access patterns. Much of the previous work has focused on graphs arising from scientific applications (meshes [50], etc) that have a different structure than social networks and RDFs focused in this study, where well-defined partitions often do not exist [84]. In this study, our focus is not to design new graph partition algorithms, but to adjust partitions to serve queries efficiently. We design a **S**elf **E**volving Distributed **Gra**ph **M**anagement **E**nvironment (Sedge). While Sedge adopts the same computation model and programming APIs of Pregel [98], it emphasizes graph partition management, which is the key to query performance. It adds important functions to support overlapping partitions, with the goal of minimizing inter-machine communication and increasing parallelism by dynamically adapting graph partitions to query workload change.

Our contributions. A major contribution of this study is an examination of an increasingly important data management problem in large-scale graphs and the proposal of a graph partition management strategy that supports overlapping partitions and replicates for fast graph query processing. Dynamic graph partitioning and overlap graph partitioning were widely investigated before (e.g., [151]). However, few methods study how to adapt partitions to satisfy dynamic query workload in social and information networks. We addressed this issue and proposed Sedge, a workload driven method to manage partitions in large graphs. We eliminate a constraint in Pregel [98] that does not allow duplicate vertices in partitions. This constraint makes it difficult to handle skewed query workload. It is able to replicate some regions of a graph and distribute them in multiple machines to serve queries in parallel. For this goal, we develop three techniques in Sedge: (1) Complementary Partitioning; (2) Partition Replication; and (3) Dynamic Partitioning. Complementary Partitioning is to find multiple partition schemes such that their partition boundaries are different from one another. Partition replication is to replicate the same partitions in multiple machines to share the workload on these partitions. Dynamic Partitioning is to construct new partitions to serve cross-partition queries locally. In order to perform dynamic partitioning efficiently, we propose an innovative technique to profile graph queries. As manifested later, it is too expensive to log all of the vertices accessed by each query. We introduced the concept of color-blocks and coverage envelope to bound the portion of a graph that has been accessed by a query. An efficient algorithm to merging these envelopes to formulate new partitions is thus developed. The partition replication and dynamic partitioning are together termed on-demand partitioning since the two techniques are primarily employed during the runtime of the system to adapt evolving queries. Additionally, a two-level partition architecture is developed to connect newly generated partitions with primary partitions.

We implement Sedge based on Pregel. However, the concepts proposed and verified in this work are also valid to other systems. The performance of Sedge is validated with several large graph datasets as well as a public *SPARQL* performance benchmark. The experimental results show that the proposed partitioning approaches significantly outperform the existing approach and demonstrate superior scaling properties.

6.2 System Design

Many applications [35, 120] employ graph partitioning methods for distributed processing. Unfortunately, real-life networks such as social networks might not

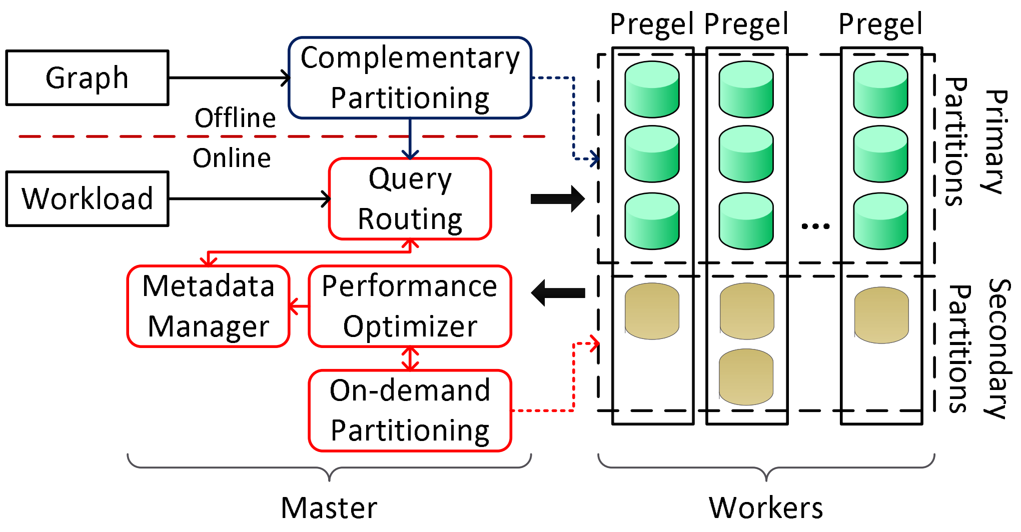


Figure 6.2: Sedge: system design.

have well-defined clusters [84], indicating that many cross-partition edges could exist for any kind of balanced partitions. For queries that visit these edges, the inter-machine communication latency will affect query response time significantly. To alleviate this problem, we propose Sedge, which is based on multi partition sets (Figure 6.2).

Sedge is designed to eliminate the inter-machine communication as much as possible. As shown in Figure 6.2, the offline part first partitions the input graph in a distributed manner and distributes them to multiple workers. It creates multiple partition sets so that each set runs independently. Pregel [98] is a scalable distributed graph processing framework that works in a bulk synchronous mode. Pregel is used as a computing platform that is able to execute local graph queries. There are various kinds of local graph queries including breadth-first search, random walk, and SPARQL queries. Unlike many graph algorithms, a local query usually starts at one vertex and only involves a limited number of vertices (termed *active vertice*). In each iteration, a Pregel instance only accesses active vertices, thus eliminating many synchronous steps. Section 6.5 will discuss synchronization for the queries with writes and updates.

The online part collects statistical information from workers and actively generates and removes partitions to accommodate the changing workload. Therefore the set of online techniques built in Sedge must be very efficient to minimize overhead. Our study is focused on partition management. For fault-tolerance and live partition migration with ACID properties, detailed explorations of these issues are given in [39, 98] and similar techniques can be applied here. In the following discussion, we overview major components including complementary partitioning, on-demand partitioning, the mechanism to connect primary and secondary partitions, the meta-data to facilitate query routing and performance optimizer.

6.2.1 Graph Partitioning

Definition 1:[Graph Partitioning] Given a graph $G = (V, E)$, graph partitioning, C, is to divide V into partitions $\{P_1, P_2, \ldots, P_n\}$ such that $\cup_i P_i = V$, and $P_i \cap P_j = \emptyset$ for any $i \neq j$. The edge cut set E_c is the set of edges whose vertices belong to different partitions. □

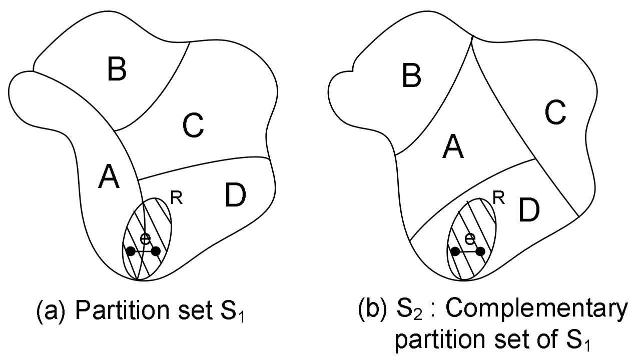


Figure 6.3: Complementary partitioning.

Graph partitioning needs to achieve dual goals. On the one hand, in order to achieve the minimum response time, the best partitioning strategy is to split the graph using the minimum cut. On the other hand, taking the system throughput into consideration, the partitions should be as balanced as possible. This is exactly what the normalized cut algorithm can do [74]. Techniques derived from graph compression, e.g., [12] can also be applied here. However, partitioning a graph using a random hash function might not work very well.

Complementary Partitioning is to repartition a graph such that the original cross-partition edges become internal ones. Figure 6.3(b) shows an example of complementary graph partitions of Figure 6.3(a). In the new partition set, the queries (shaded area R) on original cross-partition edge, e, will be served within the same partition. Therefore, the new partition set can handle graph queries that have trouble in the original partition set. If there is room to hold both S_1 and S_2 in clusters, for a query Q visiting the shaded area R in S_1, the system shall route it to S_2 to eliminate communication cost. Meanwhile, the new partition set can also share the workload with original partition set. This complementary partitioning idea can be applied multiple times to generate a series of partition sets. We call each partition set a "*primary partition set.*" Each primary partition set is self complete, where a Pregel instance can run independently.

Primary partition set can serve queries that are uniformly distributed in the graph. However, they are not good at dealing with unbalanced query workload: queries that are concentrated in one part of the graph. It will be necessary to either create a replicated partition (Figure 6.4(a)) or generate a new overlapping partition (Figure 6.4(b)) in an idle machine so that the workload can be shared appropriately. This strategy, called **On-demand Partitioning**, will generate new partitions online. These add-on partitions, called "*secondary partitions*," could last until their corresponding workload diminishes.

6.2.2 Two-Level Partition Management

Given many primary/secondary partitions, it is natural to inquire how to manage these partitions. Here we propose the concept of **Two-Level Partition Management**. Figure 6.4 depicts one example, where there are intensive workloads on two shaded areas. Based on a primary partition set, $\{A, B, C, D\}$, two secondary partitions, B' and E, are created to share the unbalanced workload on

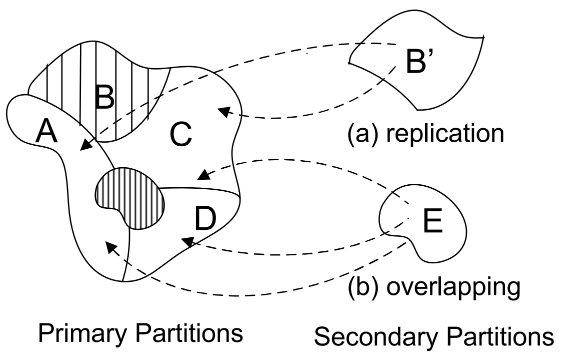


Figure 6.4: Two-level partition architecture.

primary partitions. Since the vertices in secondary partitions are the duplicates of vertices in primary partitions, some of the vertices might connect to the vertices in primary partitions. Therefore it is necessary to maintain the linkage between vertices in secondary partitions and those in primary partitions. In our design, the linkage is only recorded in secondary partitions. It is not necessary to maintain such links in primary partitions. For example, for partition B', it has to maintain the linkage to A and C. While for A and C, they only maintain links to B, but not to B'.

During the runtime, each primary partition set and the corresponding secondary partitions are maintained by a Pregel instance that is running on a set of worker machines as indicated in Figure 6.2. Multiple isolated independent Pregel instances are coordinated by meta-data management.

6.2.3 Meta-data Management

Meta-data is maintained by both the master and the Pregel instances. As in Figure 6.2, the **meta-data manager** in the master node maintains the information about each live Pregel instance and a fine-grained table mapping vertices to the Pregel instances. An index mapping vertices to partitions is also maintained by each live Pregel instance. This two-level indexing strategy is used to facilitate fast **query routing**. Specifically, when a query is issued to the system, the routing component first checks the vertex table maintained by the master. The index entry maps the vertex id to the Pregel instance which can most efficiently execute the query. After the query is routed to a particular Pregel instance, it is the duty of the vertex index maintained by the Pregel instance to decide to which partition the query should be forwarded. The detailed techniques of indexing vertices and routing queries will be discussed in Section 6.5.

In order to facilitate different kinds of queries, in addition to vertex index, it is desirable to design indices for the attributes of vertices and edges. Efficient decentralized/distributed indexing techniques, such as [133], have come to the fore in recent years. However, this topic is beyond the scope of this work.

6.2.4 Performance Optimizer

The **Performance Optimizer** continuously collects runtime information from all the Pregel instances via daemon processes and characterizes the execution of the query workload, such as vertex access times of each partition, and the number of cross-machine messages/queries. The optimizer can update the meta-data maintained by the master and evoke on-demand partitioning routine as the workload varies. It is notable that although we depict the on-demand partitioning as a component on the master side in Figure 6.2, the routine is actually executed by the Pregel instance on the worker side in a distributed manner. Therefore the overhead of on-demand partitioning will be isolated and not affect the performance of other Pregel instances.

6.3 Complementary Partitioning

Complementary partitioning is to find multiple partition sets such that their partition boundaries do not overlap. Formally, we define the problem as:

Given a partition set $\{P_1, P_2, ..., P_k\}$ *on* G *and the cut edges* $E_c = \{e_1, e_2, ..., e_i\}$. *The problem is to partition* G *into a new partition set* $\{P'_1, P'_2, ..., P'_k\}$ *satisfying the same partitioning criteria (e.g., minimum cut) such that the new cut edges do not overlap with* E_c.

If we want to exclude more edges, E_c could be expanded to include edges near the original cut edges. Without loss of generality, we assume G is an undirected graph with unit edge weight. X is an $n \times k$ matrix, defined as follows,

$$x_{ij} = \begin{cases} 1 & v_i \in V(P_j), \\ 0 & \textit{otherwise.} \end{cases}$$

X gives a k-partition set of G. Furthermore, we define the following constraints on X: (1) *full coverage and disjoint*: $X\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is a all-ones vector with appropriate size; (2) *balance*: $X^T\mathbf{1} \leq \mathbf{m}$, where $m_i = (1+\sigma)\lceil \frac{n}{k} \rceil$. m_i is a rough bound of partition size; σ controls the size balance. (3) *edge constraint*: $tr\ X^T\mathcal{W}X = \mathbf{0}$, where $\mathcal{W} = (w_{ij})$ is defined as an edge restrictive $n \times n$ Laplacian matrix. Given the edge set E_c, if $e_{ij} \in E_c$, $w_{ij} = -1$, otherwise $w_{ij} = 0$. Additionally, $w_{ii} = -\sum_{j \neq i} w_{ij}$. The complementary partitioning problem can be described below:

$$\begin{aligned} \textit{minimize} \quad & \frac{1}{2} tr\ \ X^T\mathcal{L}X \\ \textit{s.t.} \quad & X \textit{ is binary} \\ & X\mathbf{1} = \mathbf{1}, X^T\mathbf{1} \leq \mathbf{m} \\ & tr\ X^T\mathcal{W}X = \mathbf{0} \end{aligned} \tag{6.1}$$

where $\mathcal{L} = (l_{ij})$ is a $n \times n$ Laplacian matrix. By definition, if $e_{ij} \in E(G)$, $l_{ij} = -1$, otherwise $l_{ij} = 0$ and $l_{ii} = -\sum_{j \neq i} l_{ij}$. The objective function gives the overall cost of the cut edges with respect to a particular assignment of X.

The above problem is a nonconvex quadratically constrained quadratic integer program (*QCQIP*). We rewrite the problem formulation so that we can reuse the existing balanced partitioning algorithms:

$$\begin{aligned} &\textit{minimize} && tr\ \ X^T(\mathcal{L} + \lambda\mathcal{W})X \\ &s.t. && X \textit{ is binary} \\ & && X\mathbf{1} = \mathbf{1}, X^T\mathbf{1} \leq \mathbf{m} \end{aligned} \tag{6.2}$$

This new definition drops edge constraint in (6.1) and incorporate it into the objective function using a weighting factor λ on the cut edges. By changing the value of λ, we are able to control the overlap of the existing edge cut and the new edge cut generated by the complementary partition set. It also provides a scalable solution: Given the cut edges of the existing partition sets, we increase their weight by λ and then run balanced partitioning algorithms such as METIS [74] to perform graph partitioning.

The value of λ plays a critical role. Let the edge cut of the complementary partition set be E'_c. If its value is small, the partitioning algorithm can not distinct the cut edges with the others. On the other hand, if the value is too large, the algorithm might have to cut significantly more edges in order to completely avoid the existing edge cut. That is, E'_c might be much larger than E_c, which is not good too. In our implementation, we set $\lambda = 2^k$ and experiment different k with a set of simulated graph queries. For each k, we check the ratio $\beta = \frac{|E'_c| - |E_c|}{|E_c|}$. It was observed that when $k = 4$ and $\beta \leq 0.1$, the obtained partition set can achieve good performance.

Another possible technique for complementary partitioning is to delete all the edges in E_c first and then run classic partitioning algorithm. We argue that this approach doesn't work since (1) edge deletion destroys the structure of the graph, and thus the new result may probably not reflect the real connections among the graph partitions; (2) in order to preserve a good partition schema, i.e., minimum cut, in complementary partitioning, some of the edges should be included in the edge cut repeatedly.

The heuristic algorithm can be applied multiple times to generate a series of complementary partition sets, each of which try to partition the graph such that the boundary edges in one partition set will be internal edges in another partition set. With multiple partition sets, for each vertex u, there could be several partitions $P_1, P_2, \ldots, P_l$ to handle queries submitted to u. Queries should be routed to a partition where u is far away from partition boundaries. We define such a partition as a *safe partition* for vertex u. As soon as a new complementary partition set is generated, we can obtain the safe partitions for the vertices, especially those on the boundary of the original partitions.

Remark. There are some extreme cases, e.g., complete graph, where no complementary partition schema exists. However, for large graphs with small dense substructures, we can continuously perform complementary partitioning. In reality, due to space limitation, we can only afford a few sets of complementary partitions, and resort to on-demand partitioning algorithms to handle skewed query workloads that target some hotspots.

6.4 On-demand Partitioning

In the processing of many graph queries, primary partitions could have hotspots that are frequently visited. The queries heading to these partitions will suffer longer response time. There are two kinds of query hotspots: (1) internal hotspots that are located in one partition; (2) cross-partition hotspots that are on the boundary of multiple partitions. We developed two partitioning techniques, *partition replication* and *dynamic partitioning*, to generate secondary partitions on demand to handle hotspots.

6.4.1 Partition Replication

Definition 2:[Partition Workload] Given a graph G, a partition $P \subseteq G$, and a query set $Q = \{q_1, q_2, \ldots, q_m\}$, the query set of P, written $W(P)$, is the queries that have accessed at least one vertex in P. The internal query set of P, written $W_{int}(P)$, is the set of queries that only accessed vertices in P. The external (cross-partition) query set of P, written $W_{ext}(P)$, is equal to $W(P) - W_{int}(P)$. □

Given a partition P, when its internal workload ($W_{int}(P)$) becomes intensive, it will saturate the CPU cycles of the machine that holds P. One natural solution is to replicate P to P'. If there is an idle machine with free memory space, Sedge will send P' to that machine. Otherwise, it will find a slack partition and replace it with P'. A slack partition is a secondary partition with low query workload on it. By routing queries to P', the workload on P could be reduced.

6.4.2 Cross-partition Hotspots

When cross-partition hotspots exist, primary partitions have to communicate with each other frequently to answer cross-partition queries. Instead of replicating multiple partitions, it is better to generate new partitions that only cover cross-partition hotspots. The new partitions will not only share heavy workload, but also reduce communication overhead, thus improving query response time.

Hotspot Analysis. Before assembling a new partition, we need to find cross-partition hotspots first. Given a partition, we calculate a ratio $r = \frac{|W_{ext}(P)|}{|W_{int}(P)|+|W_{ext}(P)|}$ and resort to a hypothesis testing method to detect abnormal cross-partition query workload.

If a query is uniformly and randomly distributed over a partition P, we can calculate the probability of observing a cross-partition query in P by either doing a simulation or approximating it using the following external edge ratio, $p = \frac{|E_{ext}(P)|}{|E_{int}(P)|+|E_{ext}(P)|}$, where $|E_{ext}(P)|$ is the number of cross-partition edges between P and other partitions, and $|E_{int}(P)|$ is the number of internal edges. If r is significantly higher than p, it could be reasonably assumed that there are cross-partition hotspots in P. Let $n = |W_{int}(P)| + |W_{ext}(P)|$ and $k = |W_{ext}(P)|$. The chance to have $\geq k$ cross-partition queries is

$$Pr(x \geq k) = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}.$$

When $Pr(x \geq k)$ is very small (e.g., 0.01), it means there is an abnormal large number of cross-partition queries in P.

6.4.3 Track Cross-partition Queries

Besides detecting cross-partition hotspots, we need a method to track the trail of cross-partition queries and pack them to form a new partition. It is intuitive to record each query in the form of its exact search path. However, it is not only space and time consuming for profiling, but also difficult to generalize. Instead we mark the search path of a cross-partition query with coarse-granularity units, *color-blocks*.

A **color-block** is a set of vertices $V_i \subset V$ where they are assigned with a unique color c_i. For any vertex $v \in V$, it has one and only one color. Using color-blocks, we are able to coarsen a graph with a much smaller number of units. To form color-blocks, we experimented on several algorithms, i.e., *nearest-k neighbors*, *neighbors within k-hops*, etc, and found that neighbors within 1-hop outperforms the others. Disjointed 1-hop color-blocks could be generated as follows: (1) randomly select one vertex, find its 1-hop neighbors, and form a color-block; (2) delete the vertices of this color-block; (3) repeat (1) and (2) until no vertex is left.

6.4.4 Dynamic Partitioning

[**Query Profiling**] Given a set $C = \{c_1, c_2, ..., c_n\}$ of color-blocks, we track the trail of a query with a subset of color-blocks, $L_j = \{c_{j_1}, c_{j_2}, ..., c_{j_l}\}$. Since these color-blocks will be grouped together later, it is not necessary to record the visiting order of color-blocks. L_j is termed an **envelope** of the query.

By tracking cross-partition queries using color-blocks, each query can be profiled as an *envelope*. Figure 6.5 shows the relation among partitions, color-blocks

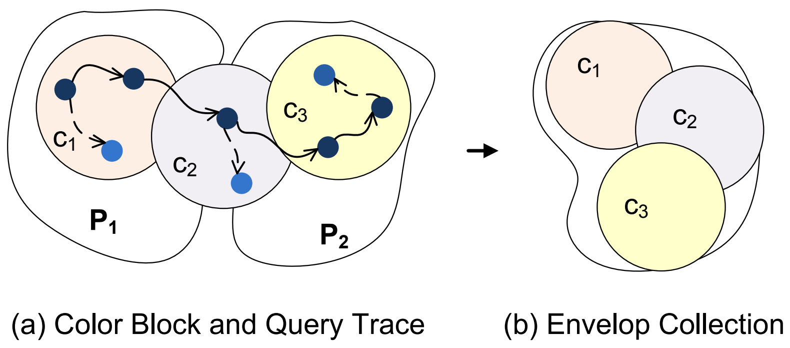


Figure 6.5: Color-block and envelop collection.

and envelopes. Given a set of candidate envelopes, a partition cannot assemble all of them due to its space constraint. Herein we formulate the problem as an *envelopes collection* problem.

[**Envelopes Collection**] Given a partition with the storage capacity M, there are a set $L = \{L_1, L_2, ..., L_n\}$ of envelopes and a set $\bigcup_{j=1}^{n} L_j$ of m colors, each envelope L_j encapsulates a set $L_j = \{c_{i_1}, c_{i_2}, ..., c_{i_l}\}$ of colors and the size of color c_k is w_k. If $D \subseteq L$ and $R = \bigcup_{L_j \in D} L_j$, the objective is to find such a set D that maximizes $|D|$ with the constraint $\sum_{c_k \in R} w_k \leq M$, where M is the default partition size.

Envelopes collection is reminiscent of the *Set-Union Knapsack Problem*, which is a classic NP-complete problem. We propose a greedy algorithm based on the intuition that combining similar envelopes consumes less space than combining non-similar ones. Given two envelopes L_i and L_j, the overlap of their color-block sets is measured as the *Jaccard coefficient* $Sim(L_i, L_j) = \frac{|L_i \cap L_j|}{|L_i \cup L_j|}$. Given n

envelopes, performing pair-wise similarity comparison is a procedure running in $O(n^2)$. To cope with this challenge, we employ a hash-based algorithm, called *Locality Sensitive Hashing* (LSH) [51] to perform similarity search in a provably sublinear time.

LSH is a probabilistic method that hashes items so that similar items can be mapped to the same buckets with high probability [51]. In our case, we adopt a LSH scheme called *Min-Hash* [32]. The basic idea of Min-Hash is to randomly permute the involved set of color-blocks and for each envelope L_i we compute its hash value $h(L_i)$ as the index of the *first* color-block under the permutation that belongs to L_i. It has been shown in [32] that if we randomly choose a permutation that is uniformly distributed, the probability that two envelopes will be mapped to the same cluster is exactly equal to their similarity. We use Min-Hash as a probabilistic clustering method that assigns a pair of envelopes to the same bucket with a probability proportional to the similarity between them. Each bucket is considered as a cluster and the envelopes within the same bucket are combined together.

[**Partition Generation**] After obtaining a set of independent clusters, each cluster is assigned with a benefit score, $\rho = \frac{|W(C)|}{|C|}$, to measure the quality of the cluster. Here $|W(C)|$ is the number of cross-partition queries denoted by all the envelopes in the cluster C (more accurately, the times of the color-blocks in C

are accessed) and $|C|$ is the size of the cluster. We create an empty partition and iteratively assemble the cluster with the highest ρ at each step as long as the total size is no greater than the default partition size M.

Algorithm 9 Similarity-Based Greedy Clustering Algorithm

```
Input: Envelope set $L = \{L_i\}$;
Output: New partition $P$;
1: Initialize hash functions;
2: for each $L_i \in L$ do
3:     hash_value = $h(L_i)$;
4:     add $L_i$ to $C_{hash\_value}$;
5: end for
6: $C = \{C_{hash\_value}\}$ for each $C_{hash\_value} \neq \emptyset$;
7: for each cluster $C_i$ in $C$ do
8:     $\rho[i] = |W(C_i)|/|C_i|$;
9: end for
10: Sort clusters on $\rho$ in descending order;
11: cluster set $P = \emptyset$;
12: Add clusters to $P$ as many as possible, s.t., $size(P) \leq M$;
```

Scalability issues. The greedy algorithm is outlined in Algorithm 9. For n envelopes, the complexity of Min-Hash clustering is $O(n)$ (lines 1-5) and the sorting runs in $O(mlog(m))$ (line 9) where m is the number of the clusters generated (line 6). In the worst case, combining the clusters needs $O(nm)$ (line 12). In total, the complexity of this greedy algorithm is $O(nm)$. There is still a concern that if n and m are large, this algorithm would lead to poor scalability. To cope with this challenge, we limit the growth of n and m in the following way. On one hand, we use a *sampling* method to constrain the size of n. For example, when the dynamic partitioning procedure is triggered, among a set of cross-partition queries we randomly select a number of queries as a sample to generate the new partition. On the other hand, we could coarsen the size of color-blocks by increasing the number of vertices included in these blocks. This will result in a color set much smaller than the vertex set. In the experiment, we show that these two methods collectively guarantee that the dynamic partitioning method works in an efficient way.

Discussion: Duplicate Sensitive Graph Query. As a design principle, primary partitions are disjointed: each vertex only has one copy in the partitions. However, when secondary partitions exist, it is often the case that there are two copies v and v' for the same vertex. It might cause a potential issue, as illustrated in Figure 6.6. Figure 6.6(a) shows the original graph. In Figure 6.6(b), secondary partition P_2 is added and v' is a duplicate vertex v. Suppose we run the following algorithm to calculate the number of v's 2-hop friends :

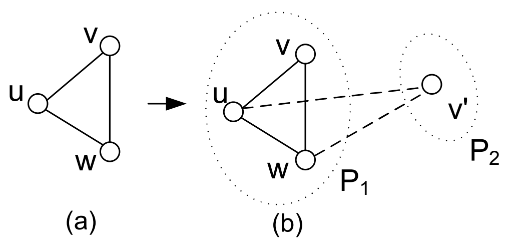


Figure 6.6: Duplicate vertex.

[Method 1] *Starting at v', we send a message to its* 1*-hop friends and these friends send another message to their* 1*-hop friends. Each partition reports the number of vertices who received messages. Sum up the numbers.*

The above algorithm works correctly in primary partitions. However, for Figure 6.6(b), it will produce a wrong answer. Due to this complication, it is not straightforward to run queries correctly in secondary partitions. Fortunately, for many local graph queries, there are implementations that are not sensitive to overlapping partitions. If we change Method 1 slightly, it will work correctly.

[Method 2] *Starting at v', we send a message to its* 1*-hop friends and these friends send another message to their* 1*-hop friends. Each partition reports the vertices who received messages. Union the results by removing duplicates.*

Other graph queries such as random walk, personalized PageRank, hitting time and neighborhood intersection have implementations that are not sensitive to duplications. We call queries that can be correctly answered on overlapping partitions *Duplicate Insensitive Graph Queries.* If a duplicate sensitive graph query running on a secondary partition exceeds the boundary of the partition, the query will be terminated and restarted in a primary partition. In Sedge, the query routing component (described in the next section) maintains a vertex-partition fitness list for the start vertex of a query. It helps route the query to a partition that can serve it locally with high probability.

6.5 Runtime Optimization

6.5.1 Query Routing

An incoming query arrives with at least one initial vertex. The master node dispatches the query to a Pregel instance according to the initiated vertex. As shown in Figure 6.3, if possible, a query shall be routed to a Pregel instance (PI for short) where its initiated vertex is in the safe region. Here, we devise a data structure in the master node to coordinate query routing:

- Instance Workload Table (IWT): $I \rightarrow W(I)$, where I is the ID of a PI and $W(I)$ is the workload of the PI.
- Vertex-Instance Fitness List (VFL): $v \rightarrow L_v\{I\}$, where $L_v\{I\}$ is an id list of the PIs.

Given a vertex v, the PIs where v is in safe region are ranked higher in *VFL*. Since some vertices, such as those with very high degree, might not be in any safe region, we assign a random order of PIs to their *VFL*s. During the runtime, the IWT is updated by the monitoring routine. Given a query, the algorithm routes the query to the first PI in its *VFL* that is not *busy* with respect to the IWT. Once the query is finished, if the query cannot be served locally in its assigned PI, the query fitness list will shift the PI to the end of the list. Since the number of Pregel instances is small, *VFL* is implemented using *bitset*. Bitset is an array optimized for space allocation: each element occupies only one bit. For example, it uses only 3 bits to represent up to 8 PIs. Our experimental results show that the simple greedy routing strategy can outperform random query routing significantly.

Vertex-Partition Mapping. In order to process queries, each Pregel instance needs to maintain the following tables to map vertices to partitions. All partitions are mapped onto unique IDs.

- Partition Workload Table (PWT): $P \rightarrow W(P)$, where P is the ID of a partition and $W(P)$ is the workload.
- Vertex-Primary Partition Table (*VPT*): $v \rightarrow P$, where P is a primary partition. Each vertex is mapped to one and only one primary partition.

- Partition-Replicates Table (PRT): $P \rightarrow \{S_R\}$, where $\{S_R\}$ are the identical replicates of P. For $\forall v \in P$, it may associate with several S_R.
- Vertex-Dynamic Partitions Table (VDT): $v \rightarrow \{S_D | v \in S_D\}$, where $\{S_D\}$ are the new partitions generated by the dynamic partitioning method.

Space complexity. Due to the limited number of partitions in practice, the size of the PWT and the PRT is negligible. VPT is $O(n)$, where n is the number of vertices in G. It only takes several gigabytes to store a VPT table for billions of vertices. The size of VDT depends on the number of vertices covered by the secondary partitions. Usually, the size is far smaller than $O(n)$.

In particular, each secondary partition is associated with one primary partition set from which it is created. When a secondary partition is generated or deleted, an entry in PRT or VDT needs to be updated accordingly. For K Pregel instances, we maintain their tables separately. That is, we will have K sets of PWT, VPT, PRT and VDT. These tables are stored in main memory.

6.5.2 Partition Workload Monitoring

The workload monitoring component in Sedge is built in the *optimizer* module (Figure 6.2). Report messages from all Pregel instances are sent to the master at the end of each period. Typically a report message from a Pregel instance I

includes the number of the queries served in I (i.e., $W_{int}(I)$ and $W_{ext}(I)$), the total access times of the vertices ($\sum_{q \in W(I)} |V(q)|$), and the CPU run time of the machines holding I. These messages encode the workload information of Pregel instances. The master updates the *IWT* accordingly. Analogously, each Pregel instance collects runtime information of their partitions and calculates the ratio between the total access times of the vertices and the size of the partition and sorts the partitions based on the ratio. Then with respect to the threshold ratio, a partition can be marked as a *hot* or a *slack* one. The information is maintained in the *PWT*.

6.5.3 Partition Replacement

As discussed in Section 6.4, secondary partitions are generated to deal with query *hotspots*. In practice, the space that can be used to accommodate additional partitions is often limited. Therefore, it is unlikely to create as many secondary partitions as possible. At the same time, in real-world applications, query hotspots may become "*slack*" ones after a period. This practical issue motivates a partition replacement scheme that replaces a slack secondary partition with a newly generated one. In Sedge, when a replacement is needed, we simply select the slackest secondary partition and replace it with the one newly generated.

6.5.4 Dynamic Update and Synchronization

Real-world graphs usually change over time in terms of insertion and deletion of nodes and edges. Sedge can adapt to these dynamic changes. Here we take the update on one Pregel instance as an example. Since the information of a vertex can be obtained by referring to the vertex-partition map, edge insertion and deletion can be accomplished directly. For the insertion/deletion of edge (u, v), find the primary and secondary partitions of u and v, insert or delete the edge. To delete vertex v, one can retrieve all of its edges and delete them, and then retrieve all of partitions containing v and delete v. For insertion of vertex v and its edges, one can first locate a primary partition P where the majority of v's neighbors are located, and then add v to that partition. Meanwhile, update all of the replicates of P and then submit edge insertion requests. For vertex insertion and deletion, we also need to update the vertex-partition map, i.e., VFL, VPT and VDT. Note that the update should be applied to all the Pregel instances. When the insertion of vertices and the following edge insertions make a primary partition too big, we need to redo the partitioning from scratch. Additionally, when a query changes vertex values during its execution, the cost of keeping the vertex values in sync is usually quite high especially when there are many duplicates. In Sedge, we adopt a simple strategy: when a query changes a vertex value, a new update query is issued to all the corresponding partitions. An experiment in Section 6.6.2 demonstrates the efficiency of dynamic update in Sedge.

6.6 Experimental Evaluation

The system is programmed in Java. We use a distributed version of METIS [74] to generate primary partitions. To evaluate Sedge on a diversified set of graphs and queries, we test datasets in two categories: RDF benchmarks and real graph datasets using different sets of graph queries. Our experiments are going to demonstrate that (1) Sedge is efficient and scalable, in comparison with the situation without partition management, and (2) the design of each component including complementary partitioning and on-demand partitioning is effective for performance improvement.

The experiments are conducted on a cluster with 31 computing nodes: each has 4 GB RAM, two quad-core 2.60GHz Xeon Processors and a 160 GB hard drive. Among these nodes, one serves as the master and the rest as workers. The cluster is connected by a gigabit ethernet. In each experiment, we perform three cold runs over the same experimental setting and report the average performance.

For each graph in the following experiments, we generate 5 complementary partition sets beforehand. We use CP_1 to denote the performance when only using the first primary partition set while CP_2, CP_3, CP_4 and CP_5 to denote the performance when using 2, 3, 4 and 5 partition sets, respectively. Each primary partition set consists of 12 primary partitions, which fill in 6 workers.

6.6.1 Evaluation with a SPARQL Benchmark

We first evaluate the system performance of Sedge on a SPARQL benchmark graph. SPARQL is an emerging standard for RDF. Efficient storage techniques for large-scale RDF data and evaluation strategies for SPARQL are currently under exploration in the database community [9,128]. In this experiment, we will illustrate that our partitioning techniques can improve SPARQL query execution significantly.

The SP^{2}Bench Benchmark [128] chooses the *DBLP* library as its simulation basis. It can generate arbitrary large RDF test data which mirrors vital real-world distributions found in the original DBLP data. Using the generator provided by [128], we create an RDF graph with $100M$ edges (11.24GB). It is a heterogenous graph with the subjects/objects as the vertices and the predicates as the links.

SP^{2}Bench provides 12 query templates, $Q1, Q2, \ldots, Q12$ that are delicately designed to capture all key features of the SPARQL query language. In this work, we select five categories in which the existing SPARQL engines have difficulties. These queries are listed in the *Appendix*. From the view of query operation, $Q6$ and

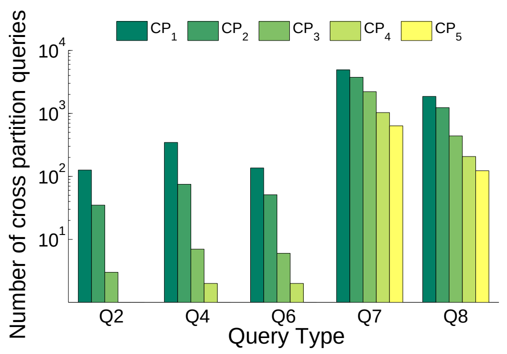


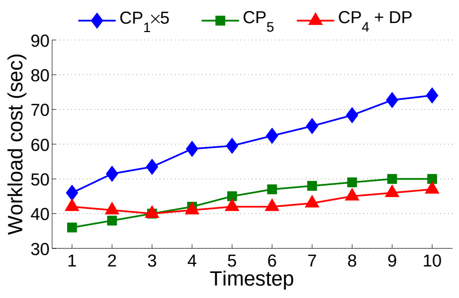
Figure 6.7: Number of cross-partition queries.

The missing bars for the CP_4 and CP_5 of $Q2$, the CP_5 of $Q4$ and the CP_5 of $Q6$ correspond to the value of 0, i.e., the cross-partition query vanishes.

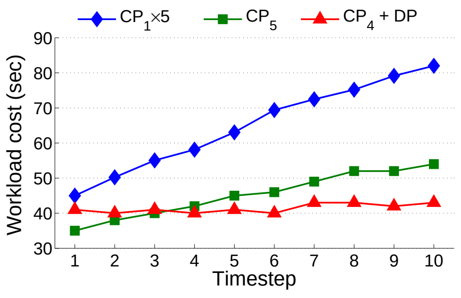
$Q7$ encode the operations of OPTIONAL (akin to left outer joins) with FILTER and BOUND; from the view of access pattern, $Q2$ and $Q4$ contain two distinctive graph patterns, "*long path chains*" and "*bushy patterns*" [128]; $Q8$, extracting the *Erdös Number* of the authors, showcases the queries that concentrate on a "*hotspot*." We map the queries against specific vertices as the query starts and thereafter match the variables to the nodes or edges during the query execution.

In order to validate the complementary partitioning approach, we generate a workload with $10,000$ queries, which are the equal mixture of the 5 query types with randomly selected starts. The queries are routed automatically to the corresponding partitions with the assistance of the query routing module. We compare the performance by varying the number of the used primary partition sets. Figure 6.7 shows the effect of the approach. Note that the Y-axis is plotted in logarithmic scale to accommodate the significant differences in the number of queries that access at least two partitions. It is observed that by adding more complementary partition sets, the number of cross-partition queries can be dramatically reduced. It vanishes for Queries $Q2$, $Q4$ and $Q6$ when 4 or more complementary partition sets are used. A close look at the difference in the performance between the variants of query types reveals that $Q2$, $Q4$ and $Q6$ exhibit high locality. In contrast, $Q7$ and $Q8$ exhibit more complex access pattern. Figure 6.7 shows for the queries of $Q7$ and $Q8$, CP_5 outperforms CP_1 by up to almost one order of magnitude. The result suggests that our complementary partitioning is an effective way in response to cross-partition queries of various types. Figure 6.7 also shows, with respect to different queries, how the percentage of vertices in safe partitions changes when the number of complementary partition sets increases. For example, for $Q7$, the percentage of vertices in safe partitions increases from 50.9% (1 partition set) to 94.7% (5 complementary partition sets); and for $Q8$, it increases from 81.4% to 97.6%.

To demonstrate how Sedge responds to skewed workloads, we generate a synthetic evolving workload which contains 10 timesteps. In each timestep, the workload consists of $10,000$ queries which are the mixture of the 5 query types with equal number. To control the evolution of the workload, each query is assigned with a lifetime value. If the query is internal (finished within a partition), it



(a) Lifetime$_c$ = 2



(b) Lifetime$_c$ = 5

Figure 6.8: Performance of complementary and on-demand partitioning.

has lifetime, $lifetime_I$; otherwise, it has lifetime, $lifetime_C$. When a query expires, it will restart in the next timestep with a new lifetime and a randomly selected start. Since random internal queries do not contribute to a skewed workload, we set $lifetime_I = 1$ for simplicity and vary the value of $lifetime_C$ in the following experiments. Note that when $lifetime_C > lifetime_I$, the number of cross-partition queries will increase gradually because more internal queries will become cross-partition queries than the reverse along the time.

We compare the approaches from two perspectives: complementary partitioning and on-demand partitioning. $CP_1 \times 5$ uses 5 static replicates of the first partition set (i.e., run five Pregel's independently, each with 1/5 workload), and CP_5 uses all the 5 complementary partition sets. Both of the two approaches use up 30 worker space. Note that we run these two settings only using Pregel instances where no query profiling (on-demand partitioning) is applied. $CP_4 + DP$

uses 4 complementary partition sets and employs the rest worker space for on-demand partitioning. To maintain a fair comparison, the number of secondary partitions can not exceed 12, the size of one partition set in our experiments.

Figure 6.8 reports the accumulated time cost of the query workload at each timestep with respect to the three approaches. The overhead of on-demand partitioning is also included in the workload cost. Figure 6.8(a) shows the performance of these approaches when $lifetime_C = 2$. The curve of CP_5 illustrates that the complementary partitioning technique significantly outperforms the static replication ($CP_1 \times 5$). The advantage becomes more obvious along with the accumulation of the cross-partition queries. It can also be seen that due to the generation of new secondary partitions, $CP_4 + DP$ outperforms CP_5 after timestep 3. When $lifetime_C = 5$, Figure 6.8(b) shows a similar result of the comparison between $CP_1 \times 5$ and CP_5 as in Figure 6.8(a). However, in Figure 6.8(a), $CP_4 + DP$ outperforms CP_5 noticeably after timestep 3 and the time cost almost remains steady. This is because when $lifetime_C = 2$, due to the dynamics of the queries, the system invokes on-demand partitioning more frequently (6 times) than that when $lifetime_C = 5$ (3 times).

6.6.2 Evaluation with Real Graph Datasets

Next, we evaluate the design of Sedge by testing the effectiveness of each component. We use another set of graphs and queries to show the broad usage of Sedge. Nevertheless, the same test can be conducted on SP^{2}Bench and similar results will be observed.

Datasets. We use both real-world graphs and the synthetic graph in the test. **Web graph.** It is a uk-2007-05 web graph data from http://webgraph.dsi.unimi.it [12], which is a collection of UK websites. It contains 30M vertices and 956M edges. **Twitter graph.** The Twitter graph is crawled from *Twitter*, consisting of 40.1M users. There are 1.4B edges (including multi-edges) in this dataset. For simplicity, we aggregated the multi-edges and the associated attributes as one edge which represents several messages sent from one user to another at different time. **Bio graph.** The Bio graph is a *de Bruijn Graph* built from a sample of mRNA. In this graph, vertices represent sub-sequences of DNA symbols with length of twenty one (a.k.a. *k-mer length*) and edges represent the adjacent relationships between vertices: the two vertices differ by a single symbol [164]. We collect 50M vertices and construct 68M edges. The resulting de Bruijn graph is like a tree.

Graph	Size (GB)	Partition (s)	*VFL* (MB)	*VPT* (MB)
Web	14.8	120	81.5	35.3
Twitter	24	180	109.0	45.4
Bio	13	40	135.9	55.3
Syn.	17	800	543.7	205

Table 6.1: Graph datasets.

Synthetic scale-free graph. The graph is generated based on R-MAT [21]. It consists of 0.2 billion vertices and 1.0 billion edges. The graph matches "*pow-law*" behaviors and naturally exhibits "*community*" structure.

Table 6.1 summarizes the size of the graphs, the time cost of building one primary (complementary) partition set, the size of the vertex-instance fitness list (*VFL*), and the size of the vertex-partition table (*VPT*). It can be seen that the auxiliary meta-data is much smaller than the graph it serves, only 0.5% – 5% of its size.

We use three classic local graph queries to experiment the performance: (1) *h-hop Neighbor Search (h-NS)*: the query starts from a vertex v and does a breath-first search for all the vertices within h hops of v; (2) *h-step Random Walk (h-RW)*: the query starts at a vertex and at each following step jumps to one of its neighbors with equal probability. The query consists of h steps; (3) *h-step Random Walk with Restart (h-RWR)*: it is a *h-step random walk* query; but at each step it may return to its start vertex with p probability. We set $p = 10\%$ by default.

For global graph algorithms like single-source shortest distance, Sedge could also support them. However, they are not the focus of this work.

We test the effectiveness of our proposed algorithms: *complementary partitioning*, *partition replication* and *dynamic partitioning*. Due to the space limitation, we first show the experiments on the *Web graph* with different test settings. For the other datasets, we get quite similar results. We will then give an evaluation of the system on the scalability, using all of the four graphs.

Complementary Partitioning

Figure 6.9 shows the effect of complementary partitioning in reducing the communication cost. In this experiment, we use CP_1 as the baseline (the result will not change if we replicate CP_1 five times) and test $10,000$ *h-RWR* queries using different number of complementary partition sets. By varying the step of the *h-RWR*, it can be seen that the complementary partitioning method can reduce the inter-machine messages. As to queries with longer random walk, the performance of Sedge degrades. However, with more complementary partitions, e.g., CP_4 and CP_5, Sedge can still achieve good performance in message reduction.

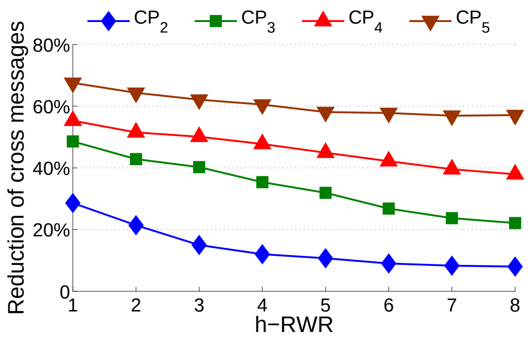


Figure 6.9: Complementary partitioning.

Partition Replication

To evaluate the performance of partition replication on unbalanced workload, we randomly generate a workload with mixed queries, i.e., *3-NS*, *5-RW*, *5-RWR*, on a specific graph partition (denoted as P_1) and continuously increase the number of queries from $10,000$ to $50,000$. We run this changing workload under 3 different settings: (1) CP_1 (the baseline); (2) CP_1 and 1 replicate of P_1 (ref. as $CP_1 + P_S$); (3) CP_1 and 2 replicates of P_1 (ref. as $CP_1 + P_S \times 2$). Figure 6.10 shows the number of queries can be served per second (throughput) for each setting. It is observed that the throughput by using partition replication significantly outperforms that of no replication one. This is because the query workload on $P1$ is distributed and processed in parallel among the primary partition and its replicates.

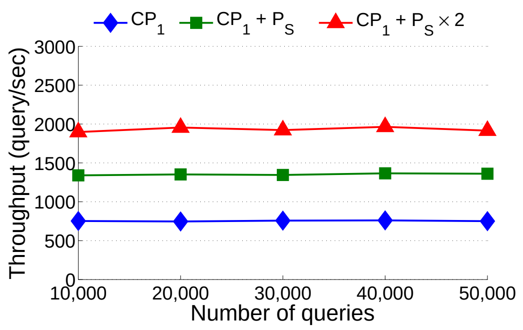


Figure 6.10: Partition replication.

Dynamic Partitioning

To test the performance of dynamic partitioning, we focus on queries that access multiple partitions. We randomly generate mixed cross-partition queries (*3-NS*, *5-RW* and *5-RWR*) and test the system performance by varying the number of queries from $10,000$ to $50,000$. We run Sedge with only one primary partition set (CP_1) as well as with one primary partition set and on-demand generated secondary partitions ($CP_1 + DP$), respectively.

Figure 6.11 shows the runtime cost of dynamic partitioning. It measures the run time of each stage to finish a dynamic partitioning process: *query profiling*, *envelopes collection* and *new partition generation*. The figure shows the cost per query by varying the number of cross-partition queries. For all the three stages, it is observed that the cost remains almost constant. Therefore the dynamic partitioning method is scalable with respect to the number of cross-partition queries.

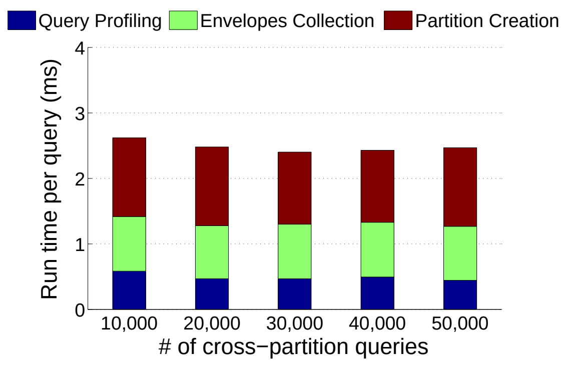


Figure 6.11: Dynamic Partitioning: runtime cost.

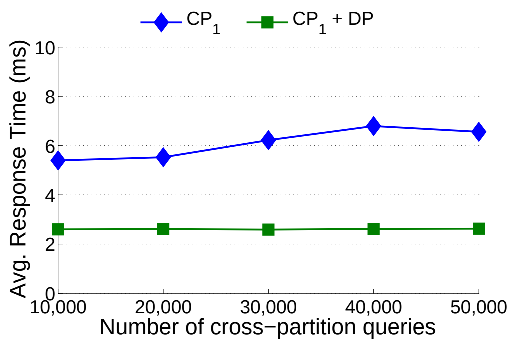


Figure 6.12: Dynamic partitioning: response time.

We next use the same query workload to test the effect of dynamic partitioning. Figure 6.12 shows the average response time by varying the number of cross-partition queries. Note that the response time here only indicates the query answering time. From the figure, we can observe the query response time is significantly improved compared to the static partitioning method. This also explains that our algorithms are effective for serving cross-partition queries. In the above experiments, Sedge uses slightly larger space with secondary partitions.

Scalability Evaluation

Additionally, we test the capability of Sedge to handle intensive cross-partition queries. We generate five sets of query workload, each of which contains $100,000$ random queries and set the percentage of the cross-partition queries as 0%, 25%, 50%, 75% and 100%, respectively. For this experiment, we use CP_1 as the baseline and demonstrate the performance of $CP_1 + DP$, where DP denotes secondary partitions generated by dynamic partitioning on demand. We employ 6 machines to hold CP_1 and assign additional machines gradually to accommodate the new partitions.

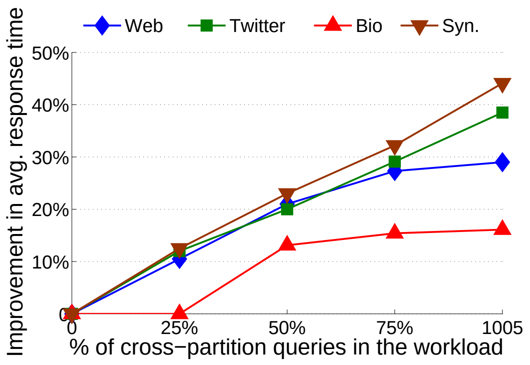


Figure 6.13: Cross-partition queries vs. improvement ratio.

Figure 6.13 shows the improvement ratio in average response time. In this figure, we plot the lift of the average response time by using on-demand partitioning compared with the baseline. The response time includes both the query answering time and the overhead of on-demand partitioning. As we increase the percentage of cross-partition queries, it can be seen that for all the four datasets, there is a significant improvement in average response time. In detail, however, we observe different improvement performance with respect to the changing workload. For the *Twitter graph* and *Synthetic graph*, the ratio increases constantly. This can be explained as follows. In these two graphs, there are many tightly connected substructures (*communities*). If these substructures are divided among multiple partitions, the cross-partition queries on them will visit these partitions frequently and as a result produce much inter-machine communication. In this case, by collecting the hot substructures together, our system can dramatically improve the efficiency. As for the *Bio graph*, it is a tree-like structure. Hence, the cross-partition query does not visit many partitions and the improvement in query response time is not remarkable when compared with the baseline method. The characteristics of the *Web graph* are between these two types.

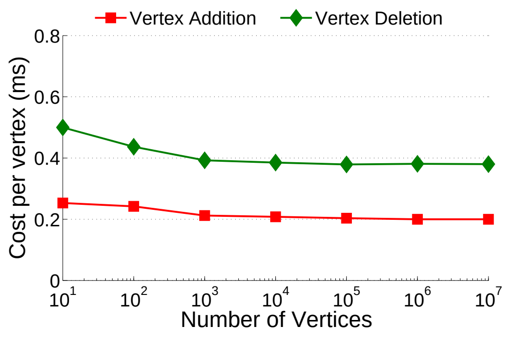


Figure 6.14: Dynamic update and synchronization cost.

Dynamic Updates and Synchronization

To test the performance of dynamic update/synchronization, we experiment on vertex addition and deletion on the large *Synthetic graph*. To assure updates are indeed executed globally, 5 primary (complementary) partition sets are initially loaded and runs in parallel. In the experiment of vertex addition, we generate new vertices with respect to the degree distribution of the graph, which is a "power-law" distribution with $\gamma = 2.43$ (a.k.a *scaling parameter*, [6]). New edges are constructed according to preferential attachment. As to the experiment of vertex deletion, we randomly select vertices in the graph to delete. Figure 6.14 shows the average run time for each vertex addition/deletion operation by varying the number of vertices. It is observed that the addition and deletion operation per vertex can be accomplished in about 0.2ms and 0.4ms respectively and the time is almost constant with respect to the number of updated vertices.

6.7 Related Work

Graph partitioning is an important problem with extensive applications in many areas, including circuit placement, parallel computing and scientific simulation. Large-scale graph partitioning tools are available, e.g. METIS [74], Chaco [63], and SCOTCH [116], just to name a few. This study is not to propose a new graph partitioning algorithm. Instead, it is focused on a workload driven method to manage partitions in large graphs.

Distributed memory systems in super-computing is able to process large-scale linked data, e.g., [77, 108]. These systems could map shared data into the address space of multiple processors. They are usually very general, supporting random memory access that has less locality than the graph queries introduced in this work, thus could not benefit from query locality. Malewicz et al. [98] introduced Pregel, which could run graph algorithms in a distributed and fault-tolerant manner. Logothetis et al. [93] introduced a generalized architecture for continuous bulk processing (CBP) that is good for building incremental applications in large datasets including graphs. Najork proposed the scalable hyperlink store, SHS [108]. SHS studied several key issues in large graph processing: real-time response, graph compression, fault tolerance, etc. Our study touches another aspect on managing partitions to fit workload changes. Kang et al. [73] developed a peta-scale graph mining system, *PEGASUS*, built on the top of the Hadoop platform. PEGASUS proposed and optimized iterative matrix-vector multiplication operators. The difference between Pregel and MapReduce can be referred to [98]. In this work, we implement and leverage the computing environment provided by Pregel, but focus on graph partition management, not optimization techniques for specific algorithms. COSI [15] is a framework that is able to partition very large social networks according to query history. Such work is optimized for static query workload and hence cannot be readily applied to dynamic query workload. Pujol et al. [120] developed a social partitioning and replication middle-ware, *SPAR*, to achieve data locality while minimizing replication. SPAR aims to optimize performance based on social network structures, e.g., communities, while our system develops partitioning techniques that adapt to query workload change. As discussed before, network structures might not reflect actual query workload. In addition to in-memory solutions, Nodine et al. [111] considered the problem of using disk blocks efficiently in searching graphs that are too large to fit in memory. The idea of using redundant blocks is related to complementary partitioning proposed in Sedge.

Distributed query processing has also been studied on semistructured data [18, 135], relational data [35] and RDF [7]. The key technique is minimizing data movement by partial evaluation, hybrid shipping, two-phase optimization and replication (see [81] for a survey). Additionally, as the emerging of Semantic Web, more and more data sources on the Web are organized in the RDF model and linked together. With the observation of the heterogeneity and scalability challenges existing in the management of RDF data, innovative data schemas have been proposed. One of the widely used techniques has been termed the *property table* [16, 155]. The technique is to cluster subjects sharing similar properties/predicates. Another technique, *vertical table* [3], is to vertically partition the schemas on property value. Efficient RDF data management is still an open problem and has not been addressed thoroughly.

6.8 Summary

We introduced an emerging data management problem in large-scale social and information networks. In order to process graph queries in parallel, these networks need to be partitioned and distributed across clusters. How to generate and manage partitions becomes an important issue. We illustrated that, for graph queries which have strong locality and skewed workload, static partition scheme does not work well. Thus, we proposed two partitioning techniques, *complementary partitioning* and *on-demand partitioning*. Based on these techniques, we introduced an architecture with a two-level partition structure, *primary* and *secondary partitions*, to handle graph queries with changing workload. The experiments demonstrated the developed system can effectively minimize inter-machine communication during distributed graph query processing. For future work, it is interesting to explore efficient RDF storage mechanisms and distributed metadata indexing solutions.

Chapter 7

The Architecture of SLQ

7.1 System Design

The system SLQ consists of back-end and front-end modules, as illustrated in Figure 7.1. For the back-end, SLQ integrates (1) a full fledged indexing module (the lower left part in Figure 7.1) which implements all the indices in Section 2.6, and (2) an offline learning module for the ranking model (the lower right part in Figure 7.1). Each module in the back-end can be maintained dynamically in response to data updates, without affecting the front-end, *i.e.,* online query processing. We also separate the information for graph structure and its node and edge content, and store the latter in the database. This enables SLQ to perform fast graph traversal by only loading much smaller graph structure into the memory, while accessing richer content, *e.g.,* attribute values, documents, pictures in the database.

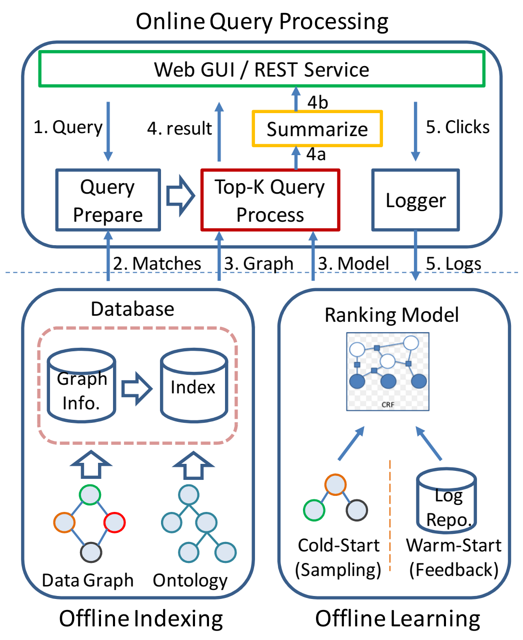


Figure 7.1: SLQ: architecture.

The front-end modules reside in the application layer (online query processing), as shown in the upper level in Figure 7.1. Upon receiving a query, (1) the "`query prepare`" module first interprets the query to an internal format, and prepares the match candidates by looking up the indices; (2) SLQ next invokes "`top-k query process`" using the ranking model to find accurate matches; (3) once the top-k results are generated, SLQ directly renders the results to the user through our "`GUI/REST service`"; (4) SLQ can also provide the users with the summarized views of the results via "`summarize`"; and (5) the user queries and result preferences are memorized by "`Logger`" to improve the ranking model.

SLQ is designed with elasticity. It can be scaled up easily by duplicating a module without affecting the others. For example, when there are intensive query requests, we can duplicate the application layer in multiple servers and then evenly distribute the queries among the servers.

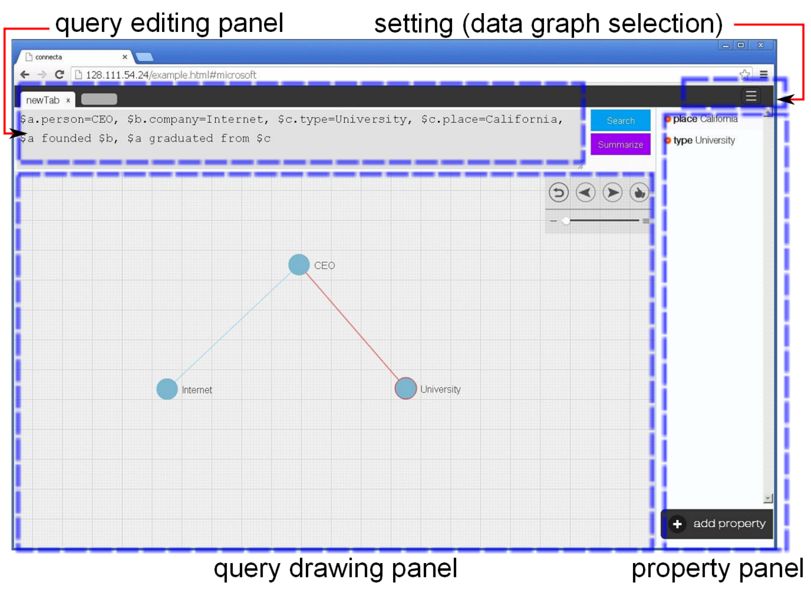
7.2 Demonstration

Setup. We demonstrate SLQ over the knowledge graphs in Table 1.1. The system is implemented in Java and is deployed on an Intel Core i7 2.8GHz, 32GB server.

Demo Scenario. SLQ provides user interfaces for *query formalization* and *query result exploration* (Figure 7.2). We invite users to experience (1) how to easily form a query without prior training in query languages or graph databases, (2) how the results, summaries and even the data graph can be conveniently navigated and viewed.

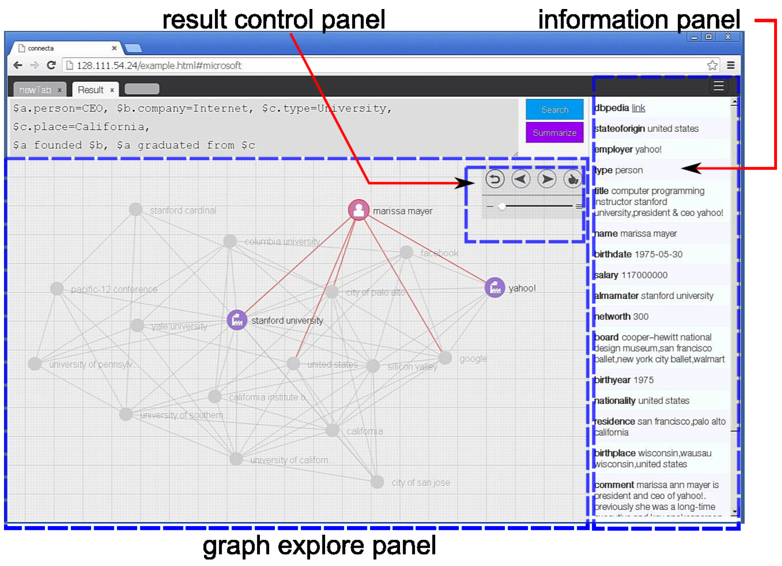
As shown in Fig 7.2(a), the query formalization interface renders the users with two major *query panels* to form queries. (1) Users can conveniently draw a graph query in the `query drawing panel`, and can freely add and edit the properties and values in the the `property panel` for each query node or edge. (2) Alternatively, users are invited to use our built-in query language SLQL in the `query editing panel`. SLQL is designed for simplicity. It consists of two types of statements: (1) the *node statement*, "$\$x.property = value$," where "$\$x$" denotes a query node with a label constraint "*property* = *value*," and (2) the *edge statement*, "$\$x$ *predicate* $\$y$," where "$\$x$" and "$\$y$" are the query nodes as in (1) and "*predicate*" indicates the relationship between the two nodes. A graph query and its SLQL representation is shown in Figure 7.2(a).

The second demonstration scenario invites users to run their queries and inspect the results. A user can also run the query on various knowledge graphs, *e.g.,*Freebase, chosen from `setting`. The results of the query in Figure 7.2(a) are shown in the graph exploration interface (Fig 7.2(b)). The `graph explore panel` renders top-1 result (the highlighted part) as well as its peripheral structure (the dim part) in the data graph. The user can then inspect the detail information of a node/edge shown in the `information panel`. By double-clicking a node, users are able to explore the one-hop neighbors of the node from the data graph. Moreover, the `result control panel` enables the user to navigate the next/previous result or return to the top-1 result. The user feedback will be recorded if they click the `like button` on specific results. We also invite users to try the `summarize` button to view the summarized results from a large collection of returned matches, and drill down to find details.



```
$a.person=CEO, $b.company=Internet, $c.type=University, $c.place=California,
$a founded $b, $a graduated from $c
```

(a) Query formalization interface



```
$a.person=CEO, $b.company=Internet, $c.type=University,
$c.place=California,
$a founded $b, $a graduated from $c
```

dbpedia link
stateoforigin united states
employer yahoo!
type person
title computer programming instructor stanford university,president & ceo yahoo!
name marissa mayer
birthdate 1975-05-30
salary 117000000
almamater stanford university
networth 300
board cooper–hewitt national design museum,san francisco ballet,new york city ballet,walmart
birthyear 1975
nationality united states
residence san francisco,palo alto california
birthplace wisconsin,wausau wisconsin,united states
comment marissa ann mayer is president and ceo of yahoo!. previously she was a long-time

(b) Graph exploration interface

Figure 7.2: SLQ: user interface.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

Knowledge graph and other complex graph data have been emerging as a foundation for many real-world applications, including search, recommendation, advertisement, Question&Answering and other intelligent systems. However, the current techniques cannot satisfy the practical requirements by leveraging the knowledge graphs. On one hand, due to the various data providers, knowledge graphs are usually quite heterogeneous, with a complex schema defined on the entities and the relationships. This fact exposes great challenges to the end users who do not possess any understanding of the data and the schema but still post their great information need. This paradox calls for effective techniques that can bridge the gap between the users and the underlying knowledge graphs. On the other hand, the volume of the graph data increases dramatically. To retrieve the information *w.r.t.* the user's request requires great computation effort and time cost. This contrasts the requirement for online applications where the users are impatient. This challenge suggests the efficient techniques that can be executed in query time.

In response to the many challenges, we propose SLQ, which is a uniform system for querying knowledge graphs. It intends to effectively process the request from users in arbitrary format, *e.g.,* keywords, nature language questions and graph queries. Overall, SLQ incorporates the following novel techniques.

Schemaless graph querying. It matches user's queries by a set of transformation functions and then adopts a learning-to-rank strategy to efficiently retrieve the best results. The ranking model considers the syntax/semantic matching signals and thus provides high-quality results. Moreover, we design a strategy to automatically generate training instances for the ranking model from the data graph. This is especially helpful in the cold-start stage where no manual effort is required for instances labeling. According to the validation on several real-life large knowledge graphs, the technique outperforms traditional keyword and approximate graph searching algorithms in terms of quality and efficiency: (a) it is able to find matches that are semantically meaningful to the queries which cannot be answered by the existing keyword or graph query methods; (b) it is 2-4 times faster than the baseline algorithm, and is orders of magnitude faster than a naive top-k algorithm that inspects every match; (c) for ranking quality, it achieves up to 50% improvement in the mean average precision, compared with its previous counterparts.

Fast top-k search. Given a query, it is important to extract and return the best answers from a large number of results. The top-k search technique is designed for this purpose and has been required in many real-world systems. Thus, we propose STAR, a top-k subgraph matching framework that deals with these new challenges. It develops fast query processing algorithms for popular star-shaped queries and makes use of this engine to solve more complex graph queries. Extensive experiments have been conducted on STAR and demonstrated its superior performance over the traditional approaches. For star queries, we found that STAR is 5-10 times faster than TA and 10-100 times faster than BP. While for general queries, our query optimization technique can achieve up to 45% runtime improvement over the baseline algorithms.

Semantic matching and indexing. Since the users' queries are often ambiguous and expressed in different vocabulary from that in the knowledge graph, this is a great need to identify answers that are not only identical or similar, but also semantically close to the query's words. To this end, SLQ also integrates a technique based on both the data graph and the ontology graph. This gives rise to the ontology-based graph querying technique. We introduce an ontology index, which is built together with the data graph. Based on the index, we propose a filtering-and-verification framework for computing the top-k results. By verifying the methods on several real graphs, the ontology-based graph querying can identify much more meaningful matches than traditional subgraph querying methods. The framework is efficient and scales well with the size of the data graphs, queries, and ontology graphs while comes with little indexing overhead.

Result summarization. Result summarization is an effective approach to render a large number of answers to the users. We formulate several interesting summarization problems for graph querying, α-summarization and K summarization. The complexity of these problems ranges from PTIME to NP-complete. We propose exact and heuristic algorithms for these summarization problems. As shown in the experimental study, our algorithms effectively summarize the answer graphs: they generate summary graphs that cover every pair of keywords with size in average 24% of the answer graphs. They also scale well with the size of the answer graphs.

Distributed graph process. Due to the excessive volume of the graph data, we introduce the distributed solutions for graph management and query processing. Based on a careful study of the query workload on large graphs, we propose two partitioning techniques, *i.e.,* complementary partitioning and dynamic partitioning. Complementary partitioning is designed to serve queries locally which need cross partition boundaries previously, thus can reduce the communication cost. Dynamic Partitioning can further reduce this cost by constructing new partitions. The balance problem is also investigated by employing partition replication. We compare our technique with several state-of-the-art frameworks on large graphs and find it is extremely efficient for graph traversal queries.

To the best of our knowledge, SLQ is among the first efforts of developing a unified system for querying large knowledge graphs. SLQ provides a set of solutions that help non-professional users access complex graph data in a much easier manner.

8.2 Future Directions

Since SLQ is designed for flexibility and being capable of incorporating any new components in need, we plan to extend it to the following directions in the future.

Nature language querying. Nature language is a user-friendly way of querying knowledge graphs. To answer a natural language question, the traditional techniques first transform it into graph format which serves as an intermediate representation that can be effectively processed by existing techniques, such as SLQ. However, this transformation poses great challenges. (1) It is difficult to infer proper entities and relationships from a short question, without the full context of the user's search intent. (2) The natural language question could be quite ambiguous, leading to many possible graph query transformations.

Distributed top-k search. Although there are many distributed top-k search techniques, they cannot be leveraged for top-k graph querying. With the increasing scale of the graph data and the emerging of many mature distributed frameworks, it is urgent to explore the top-k graph querying in a distributed setting. This implies several challenges: (1) graph querying exhibits non-locality, which usually results in a large amount of communication in the distributed environment. (2) The traditional join-then-bound strategy cannot work effectively since the bound is usually quite loose. Hence it is likely to extract an excessive number of results for the top-k search, even when k is small; (3) The query optimization problem, such as graph query decomposition, will dramatically affect the efficiency.

Some other interesting but important directions, such as graph querying benchmark and better ranking technique with user feedbacks, are also worth exploring in the future.

Bibliography

[1] Dbpedia. *dbpedia.org*.

[2] Freebase. *freebase.com*.

[3] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[4] C. C. Aggarwal and H. Wang. A survey of clustering algorithms for graph data. In *Managing and Mining Graph Data*, pages 275–301. 2010.

[5] N. Aizenbud-Reshef, A. Barger, I. Guy, Y. Dubinsky, and S. Kremer-Davidson. Bon voyage: social travel planning in the enterprise. In *CSCW*, pages 819–828, 2012.

[6] R. Albert and A.-L. Barabasi. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[7] J. H. an D. J. Abadi and K. Ren. Scalable sparql querying of large rdf graphs. In *VLDB*, 2011.

[8] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, 2008.

[9] M. Arenas and J. Pérez. Querying semantic web data with sparql. In *PODS*, 2011.

[10] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *PODS*, 2011.

[11] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461.

[12] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[13] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. SIGMOD, 2008.

[14] S. Borgatti and M. Everett. The class of all regular equivalences: Algebraic structure and computation. *Social Networks*, 11(1):65 – 88, 1989.

[15] M. Bröcheler, A. Pugliese, V. P. Bucci, and V. S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *ASONAM*, pages 248–255, 2010.

[16] J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68, 2002.

[17] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 2002.

[18] P. Buneman, G. Cong, and W. Fan. Using partial evaluation in distributed query evaluation. In *VLDB*, pages 211–222, 2006.

[19] D. Bustan and O. Grumberg. Simulation-based minimization. *TOCL*, 4(2):181–206, 2003.

[20] A. Cakmak and G. Ozsoyoglu. Taxonomy-superimposed graph mining. In *EDBT*, 2008.

[21] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[22] S. Chakrabarti, S. Sarawagi, and S. Sudarshan. Enhancing search with structure. *IEEE Data Eng. Bull.*, 33(1):3–24, 2010.

[23] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.

[24] L. Chang, X. Lin, W. Zhang, J. X. Yu, Y. Zhang, and L. Qin. Optimal enumeration: Efficient top-k tree matching. In *Proc. VLDB Endow.*, volume 8, pages 533–544, 2015.

[25] M. Charikar, S. Guha, É. Tardos, and D. Shmoys. A constant-factor approximation algorithm for the k-median problem. In *STOC*, pages 1–10, 1999.

[26] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD*, 2009.

[27] G. Cheng and Y. Qu. Term dependence on the semantic web. In *International Semantic Web Conference*, 2008.

[28] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.

[29] J. Cheng, X. Zeng, and J. X. Yu. Top-k graph pattern matching over large graphs. In *ICDE*, 2013.

[30] C. Choi, M. Cho, J. Choi, M. Hwang, J. Park, and P. Kim. Travel ontology for intelligent recommendation system. In *AMS*, 2009.

[31] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47, 2004.

[32] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[33] O. Corby, R. Dieng-Kuntz, F. Gandon, and C. Faron-Zucker. Searching the semantic web: approximate query processing based on ontologies. *Intelligent Systems, IEEE*, 21(1):20 – 27, 2006.

[34] V. Cordì, P. Lombardi, M. Martelli, and V. Mascardi. An ontology-based similarity between sets of concepts. In *WOA*, pages 16–21, 2005.

[35] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, pages 48–57, 2010.

[36] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. Divq: diversification for keyword search over structured databases. In *SIGIR*, pages 331–338, 2010.

[37] R. Dieng-Kuntz and O. Corby. Conceptual graphs for semantic web applications. In *ICCS*, volume 3596, pages 19–50, 2005.

[38] X. Ding, J. Jia, J. Li, J. Liu, and H. Jin. Top-k similarity matching in large graphs with attributes. In *Database Systems for Advanced Applications*, pages 156–170, 2014.

[39] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, pages 301–312, 2011.

[40] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[41] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[42] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.

[43] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3, 2010.

[44] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *VLDB*, 2013.

[45] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. Rex: Explaining relationships between entity pairs. *PVLDB*, 5(3):241–252, 2011.

[46] H. Fu and K. Anyanwu. Effectively interpreting keyword queries on rdf databases with a rear view. In *ISWC*, 2011.

[47] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD workshop*, 2011.

[48] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* 1979.

[49] R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: Coarsest partition problems. *J. Automated Reasoning*, 2003.

[50] J. Gilbert, G. Miller, and S.-H. Teng. Geometric mesh partitioning: implementation and experiments. *SIAM J. Sci. Comput.*, 19:2091–2110, 1998.

[51] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[52] J. Goldstein, V. Mittal, J. Carbonell, and M. Kantrowitz. Multi-document summarization by sentence extraction. In *NAACL-ANLPWorkshop on Automatic summarization*, pages 40–48, 2000.

[53] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[54] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[55] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.

[56] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, 2003.

[57] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *SIGMOD*, 2003.

[58] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *ICDE*, 2014.

[59] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[60] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[61] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[62] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[63] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. of Supercomputing*, 1995.

[64] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[65] H. H. Hoang and A. M. Tjoa. The state of the art of ontology-based query systems: A comparison of existing approaches. In *In Proc. of ICOCI06*, 2006.

[66] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, 2011.

[67] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *VLDB*, 2011.

[68] Y. Huang, Z. Liu, and Y. Chen. Query biased snippet generation in xml search. In *SIGMOD*, pages 315–326, 2008.

[69] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[70] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[71] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[72] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[73] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[74] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359 – 392, 1999.

[75] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

[76] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[77] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA*, pages 13–21, 1992.

[78] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[79] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: fast graph search with label similarity. In *VLDB*, 2013.

[80] R. Knappe, H. Bulskov, and T. Andreasen. Perspectives on ontology-based querying. *Int. J. Intell. Syst.*, 22(7):739–761, 2007.

[81] D. Kossmann. The state of the art in distributed query processing. *ACM Trans. Database Syst.*, 32(4):422–469, 2000.

[82] G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina. Data clouds: summarizing keyword search results over structured data. In *EDBT*, pages 391–402, 2009.

[83] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. On semi-automated web taxonomy construction. In *WebDB*, 2001.

[84] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[85] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[86] G. Li, B. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[87] P. Li, Q. Wu, and C. J. Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In *NIPS*, 2007.

[88] E. Little, K. Sambhoos, and J. Llinas. Enhancing graph matching techniques with ontologies. In *Information Fusion*, pages 1–8, 2008.

[89] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[90] Z. Liu and Y. Chen. Query results ready, now what? *IEEE Data Eng. Bull.*, 33(1):46–53, 2010.

[91] Z. Liu and Y. Chen. Return specification inference and result clustering for keyword search on xml. *TODS*, 35(2):10, 2010.

[92] Z. Liu, S. Natarajan, and Y. Chen. Query expansion based on clustered results. *PVLDB*, 4(6):350–361, 2011.

[93] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental algorithms. In *SOCC*, pages 51–62, 2010.

[94] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[95] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, 2013.

[96] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[97] E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - a semantic view-based search engine service for web applications. In *ISWC*, 2006.

[98] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[99] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[100] K. V. Mardia, J. T. Kent, and J. M. Bibby. Multivariate analysis. 1980.

[101] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *ICDE*, 2005.

[102] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27:415–444, 2001.

[103] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[104] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[105] J. Mondal and A. Deshpande. EAGr: supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD*, pages 1335–1346, 2014.

[106] M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In *ISWC*, 2011.

[107] D. Mottin, M. Lissandrini, V. Yannis, and T. Palpanas. Exemplar queries: Give me an example of what you need. In *VLDB*, 2014.

[108] M. Najork. The scalable hyperlink store. In *Hypertext*, pages 89–98, 2009.

[109] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[110] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[111] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[112] N. Ntarmos, I. Patlakas, and P. Triantafillou. Rank join queries in nosql databases. *VLDB*, 2014.

[113] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[114] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SICOMP*, 16(6), 1987.

[115] K. Parthasarathy, S. Kumar, and D. Damien. Algorithm for answer graph construction for keyword queries on rdf data. In *WWW*, 2011.

[116] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN*, pages 493–498, 1996.

[117] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.

[118] J. Plesník. Complexity of decomposing graphs into factors with given diameters or radii. *Mathematica Slovaca*, 32(4):379–388, 1982.

[119] J. Pound, I. F. Ilyas, and G. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, 2010.

[120] J.-M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

[121] Y. Qi, K. S. Candan, and M. L. Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *VLDB*, pages 507–518, 2007.

[122] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* O'Reilly Media, 2013.

[123] S. B. Roy, T. Eliassi-Rad, and S. Papadimitriou. Fast best-effort search on graphs with multiple attributes. *TKDE*, 99:1, 2014.

[124] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, 2013.

[125] N. Sarkas, N. Bansal, G. Das, and N. Koudas. Measure-driven keyword-query expansion. *PVLDB*, 2(1):121–132, 2009.

[126] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Euro-Par*, pages 296–310, 2000.

[127] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *International Semantic Web Conference (1)*, 2011.

[128] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A sparql performance benchmark. In *ICDE*, pages 222–233, 2009.

[129] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[130] S. Shekarpour, S. Auer, A.-C. N. Ngomo, D. Gerber, S. Hellmann, and C. Stadler. Keyword-driven sparql query generation leveraging background knowledge. In *Web Intelligence*, pages 203–210, 2011.

[131] R. Singh, J. Xu, and B. Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Research in computational molecular biology*, pages 16–31, 2007.

[132] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. *SIGKDD*, 2011.

[133] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[134] H. Stuckenschmidt and M. Klein. Structure-based partitioning of large concept hierarchies. In *ISWC*, 2004.

[135] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1):1–62, 2002.

[136] C. Sutton and A. McCallum. An introduction to conditional random fields for relational learning. *Introduction to statistical relational learning*, 93:142–146, 2007.

[137] M. Sydow, M. Pikula, R. Schenkel, and A. Siemion. Entity summarisation with limited edge budget on knowledge graphs. In *IMCSIT*, pages 513–516, 2010.

[138] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD*, 2008.

[139] M. Taylor, J. Guiver, S. Robertson, and T. Minka. Softrank: optimizing non-smooth rank metrics. In *WSDM*, 2008.

[140] S. Tejada, C. A. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *KDD*, 2002.

[141] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top-k query processing for semistructured data. *VLDB*, 2008.

[142] Y. Tian, R. Hankins, and J. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, 2008.

[143] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[144] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, 2009.

[145] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, 2009.

[146] S. Tu, L. Tennakoon, M. O'Connor, R. Shankar, and A. Das. Using an integrated ontology and information model for querying and reasoning about phenotypes: The case of autism. In *AMIA Annual Symposium Proceedings*, volume 2008, page 727, 2008.

[147] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.

[148] V. Vassilevska and R. Williams. Finding, minimizing, and counting weighted subgraphs. In *STOC*, pages 455–464, 2009.

[149] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[150] A. Wagner, V. Bicer, and T. Tran. Pay-as-you-go approximate join top-k processing for the web of data. In *The Semantic Web: Trends and Challenges*, pages 130–145. 2014.

[151] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[152] H. Wang and C. Aggarwal. A survey of algorithms for keyword search on graph data. *Managing and Mining Graph Data*, pages 249–273, 2010.

[153] Y. Weiss and W. T. Freeman. On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. *Information Theory*, 47(2):736–744, 2001.

[154] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, 2009.

[155] K. Wilkinson and K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

[156] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan. Summarizing answer graphs induced by keyword queries. *VLDB*, 6(14), 2013.

[157] Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *ICDE*, 2013.

[158] M. Xie, L. V. Lakshmanan, and P. T. Wood. Efficient rank join with aggregation constraints. *VLDB*, 2011.

[159] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. In *GRADES*, 2013.

[160] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *VLDB*, 7(7), 2014.

[161] C. Yanover and Y. Weiss. Finding the m most probable configurations using loopy belief propagation. In *NIPS*, 2004.

[162] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[163] X. Zeng, J. Cheng, J. Yu, and S. Feng. Top-k graph pattern matching: A twig query approach. *WAIM*, 2012.

[164] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Res.*, 18(5):821–829, 2008.

[165] N. Zhang, Y. Tian, and J. M. Patel. Discovery-driven graph summarization. In *ICDE*, 2010.

[166] B. Zong, R. Raghavendra, M. Srivatsa, X. Yan, A. K. Singh, and K.-W. Lee. Cloud service placement via subgraph matching. In *ICDE*, pages 832–843, 2014.

[167] L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *Ph.D. workshop in CIKM*, 2007.

[168] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. *VLDB*, 2009.

[169] L. Zou, R. Huang, H. Wang, J. X. Yu, W. He, and D. Zhao. Natural language question answering over RDF: a graph data driven approach. In *SIGMOD*, 2014.