QUERYING GRAPH STRUCTURED RDF DATA

by

SHI QIAO

Submitted in partial fulfillment of the requirements
For the degree of Doctor of Philosophy

Dissertation Adviser: Dr. Meral Z. Özsoyoğlu

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

January, 2016
CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Shi Qiao

candidate for the degree of Doctor of Philosophy *.

Committee Chair

Meral Z. Özsoyoğlu

Committee Member

Gultekin Özsoyoğlu

Committee Member

Mehmet Koyutürk

Committee Member

Marc Buchner

Committee Member

Soumya Ray

Committee Member

Andy Podgurski

Committee Member

Xiang Zhang

Date of Defense

September 22, 2015

*We also certify that written approval has been obtained for any proprietary material contained therein.
# Table of Contents

Acknowledgement ............................................................................................................................ 13

Abstract ............................................................................................................................................. 17

Chapter 1 Introduction ..................................................................................................................... 19

Chapter 2 Related Works .................................................................................................................. 24
  2.1 Querying Techniques for RDF ................................................................................................ 24
    2.1.1 Keyword based Queries ............................................................................................... 24
    2.1.2 Graph Template Matching .......................................................................................... 26
  2.2 Indexes for RDF Graphs .......................................................................................................... 28
    2.2.1 Feature Graph Pattern Index ....................................................................................... 28
    2.2.2 Neighborhood Signature Index .................................................................................. 30
    2.2.3 Graph Partitioning Index ........................................................................................... 31

Chapter 3 RDF-h ............................................................................................................................ 32
  3.1 Template Matching for RDF graphs .......................................................................................... 33
  3.2 Indexes ..................................................................................................................................... 36
  3.3 RDF Query Framework ............................................................................................................. 38
    3.3.1 Query Preprocess ......................................................................................................... 38
    3.3.2 Candidate Generation .................................................................................................. 39
    3.3.3 Neighborhood Check .................................................................................................. 39
    3.3.4 Component Matching ................................................................................................... 42
    3.3.5 Connectivity Check ....................................................................................................... 44
    3.3.6 Query Result Generation ............................................................................................. 45
  3.4 Execution Plans ......................................................................................................................... 46
4.1.2.1 Real Datasets .................................................................................. 75
4.1.2.2 Benchmark Datasets ..................................................................... 77
4.1.2.3 Results .......................................................................................... 78
4.1.3 Limitation of Coverage and Coherence Metrics ............................. 79

4.2 New Evaluation Metrics .......................................................................... 81
4.2.1 Relationship Specialty [QO-2] .............................................................. 82
4.2.2 Literal Diversity [QO-2] ..................................................................... 88
4.2.3 Insights from new Metrics ................................................................. 90

4.3 Dataset Characteristics and RDF-h ....................................................... 90

4.4 Conclusions and Comments ..................................................................... 91

Chapter 5 RBench: Application-Specific RDF Benchmarking ..................... 92

5.1 Problem Definition ............................................................................... 92
5.2 RBench [QO-3] ................................................................................... 94
5.2.1 Dataset Preprocessing .......................................................... 95
5.2.1.1 Restoring Unknown Types ..................................................... 97
5.2.2 Dataset Analysis .......................................................................... 100
5.2.2.1 Resource Identity .............................................................. 101
5.2.2.2 Relationship Patterns .......................................................... 102
5.2.2.3 Predicate Dictionary .............................................................. 104
5.2.2.4 Attribute Patterns/Stars .......................................................... 104
5.2.3 Benchmark Graph Generation ....................................................... 105
5.2.3.1 Resources and Literals .......................................................... 105
5.2.3.2 Feature Graphs and Node Assignment .................................... 107
6.4.1 Single Protein Patterns ................................................................. 147
6.4.2 Multiple Protein Patterns ............................................................ 153
  6.4.2.1 Protein-Protein Interaction .................................................... 153
  6.4.2.2 Shared Functions ................................................................. 155
  6.4.2.3 Shared Pathways ................................................................. 156
6.4.3 Query Evaluation Efficiency ....................................................... 157

6.5 Conclusions and Comments ........................................................... 158

Chapter 7 Optimizations for RDF-$h$ ...................................................... 160

7.1 Motivation ......................................................................................... 160
  7.1.1 BitWeaving .................................................................................. 162

7.2 ByteMap & Neighborhood Byte Indexes ......................................... 165
  7.2.1 Index Space Efficiency ............................................................... 167

7.3 Optimized Connectivity Check ....................................................... 168
  7.3.1 Merge Check Algorithm ............................................................. 170
  7.3.2 Cached Check Algorithm ......................................................... 171

7.4 Query Framework based on Neighborhood Byte Index .................. 175

7.5 Handling Dataset Updates ............................................................... 176

7.6 Experimental Results ...................................................................... 178
  7.6.1 Index Space Comparison ........................................................... 179
  7.6.2 Query Evaluation Analysis ....................................................... 181

7.7 Conclusions and Comments ............................................................ 185

Chapter 8 Future Works ....................................................................... 187

8.1 SPARQL Hybrid ............................................................................. 187
8.2 Distributed Framework ................................................................................................. 190

Chapter 9 Conclusions ................................................................................................... 194

Bibliography ................................................................................................................... 196
List of Tables

Table 1. Query Performance of Query Templates with Connectivity Edge ..................... 71
Table 2. Relationship specialty statistics for all datasets.................................................. 83
Table 3. Statistics of dictionary sizes with different sampling sizes for all datasets ....... 86
Table 4. Query Running Times for all Queries............................................................... 158
List of Figures

Figure 1. RDF Graph and Query Template Example with Matching Results .................. 34
Figure 2. Indexes Examples .............................................................................................. 36
Figure 3. Query Framework Structure .............................................................................. 38
Figure 4. Threshold Tuning Decision Tree ....................................................................... 58
Figure 5. Indexes Space Comparison ................................................................................ 62
Figure 6. Query Comparison (Datasets) ........................................................................... 64
Figure 7. Dataset Scalability Test ..................................................................................... 66
Figure 8. (a) Template Scalability Test ............................................................................. 68
Figure 9. Coherence metric for all datasets ...................................................................... 78
Figure 10. Indexes Examples ............................................................................................ 81
Figure 11. Relationship specialty for all datasets ............................................................. 84
Figure 12. Dictionary sizes for all datasets with $\mathcal{M} = 100k$ ......................................... 87
Figure 13. RDF graph example ($Yago$) ........................................................................... 94
Figure 14. Dataset Preprocessing ...................................................................................... 95
Figure 15. Dataset Analysis ............................................................................................ 101
Figure 16. Benchmark Graph Generation ....................................................................... 106
Figure 17. Query Generation Process ............................................................................. 108
Figure 18. Time and Memory Complexity of $RBench$ ....................................................... 115
Figure 19. Benchmark Evaluation Results for $Literal Diversity$ (a), $Dataset Coherence$ (b) and $Relationship Specialty$ (c, d) ................................................................................... 119
Figure 20. Benchmark Evaluation Results for $Literal Diversity$ (a), $Dataset Coherence$ (b) and $Relationship Specialty$ (c) ................................................................................................. 121
Figure 21. Query Evaluation for AO Benchmark and RBench ............................................. 125
Figure 22. Query Performance Evaluation for R-DBLP and DBLP ............................. 127
Figure 23. Query Performance Evaluation for R-LUBM, LUBM and LUBMD (Spath+, Stwig+, and RDF3X) ....................................................................................................... 128
Figure 24. Query Examples and Motivation .................................................................. 138
Figure 25. UniProtKB RDF Graph Example ................................................................. 142
Figure 26. Query Specifications of Q3 ........................................................................... 148
Figure 27. Query Specifications of Q5 ........................................................................... 149
Figure 28. Query Result Visualization of Q3 ................................................................. 151
Figure 29. Disease-Disease Relationships for Q5 ......................................................... 152
Figure 30. Full Visualization of Query Results for Q5 ................................................. 152
Figure 31. Disease-Disease Relationships for Q6 ........................................................ 154
Figure 32. Protein-Protein Relationships for Q7 .......................................................... 154
Figure 33. Protein-Protein Relationships for Q8 ........................................................... 157
Figure 34. Protein-Protein Relationships for Q10 .......................................................... 157
Figure 35. BitWeaving Storage Layout ........................................................................ 163
Figure 36. BitWeaving Predicate Scan & Early Pruning ............................................... 163
Figure 37. ByteMap ....................................................................................................... 165
Figure 38. NB Index ....................................................................................................... 165
Figure 39. Example Uniprot Query ............................................................................... 174
Figure 40. Neighborhood Check .................................................................................. 174
Figure 41. Index Space Comparison (Uniprot, Size 0.4 GB) .......................................... 179
Figure 42. Index Space Comparison (LUBM, Size 0.4 GB) ........................................... 179
Figure 43. Query Evaluation Comparison for Single Protein Patterns......................... 180
Figure 44. Query Evaluation Comparison for Multiple Protein Patterns ................... 180
Figure 45. Memory Cost for Cached Check................................................................ 182
Figure 46. Query Comparison (in Percentage)........................................................... 182
Figure 47. SparQL Hybrid Query Process................................................................. 188
Figure 48. RDF Schema Graph................................................................................. 188
Acknowledgement

They say it takes a village to raise a child. Little did I know it would also take a village to travel the beautiful journey culminating in this dissertation. There are many, near and far, who played a vital role in helping me pull this off with much grace.

I want to begin by thanking Dr. Meral Z. Özsoyoğlu, my dissertation committee chair and more importantly my advisor and mentor who deeply influenced how I see research and, my identity as a scholar along my PhD journey. I cannot recall any conversation where I did not feel safe to make mistakes, as Prof. Meral patiently let me ‘work through’ the mess of nascent ideas. We began with what I brought to the table and ended with what I want to take away. She, in her characteristic subtlety, taught me that work of value stems from your deep connection with it—what is it in you that makes you pursue this question and spirals out to make an impact and contribution in the world because of your deep connection with it. Thank you, Prof. Meral, for sharing the most important lesson that will guide my endeavors as a scholar and educator.

I also want to thank Dr. Gultekin Özsoyoğlu, member of my committee and my mentor on many interesting research projects over the years. Prof. Tekin provided me with many opportunities to hone my research skills. He epitomized how quality research is executed, and encouraged me to get there. There never was a time when he was not available to promptly respond to any and all of my queries. Because he truly believes in doing work of value he has inspired me to surpass the previous with every subsequent effort.

Dr. Mehmet Koyutürk, member of my committee, has contributed to my work through
many conversations and collaborations over the years. Prof. Mehmet always saw and highlighted the strengths of my work, in turn not only providing me with the energy to pursue my question but also enabling me to stay connected to my strengths. Thank you, Prof. Mehmet, for always finding what’s right in my inquiry.

Dr. Soumya Ray, member of my committee, has provided the keen pragmatic eye to amplify the relevance of my research efforts. I am thankful to Prof. Soumya for not only providing insightful feedback and technical refinements on my research but also teaching me the skills of translating my work for education and practice. Thank you, Soumya, for exemplifying the integration of research and practice, and for always being available to read, discuss, and share.

I am also deeply indebted to Dr. Andy Podgurski, Dr. Marc Buchner and Dr. Xiang Zhang, who are also in my committee, introduced me to a new paradigm of learning along my PhD study journey. As my guides and mentors since the beginning of the program, they showed a confidence in me that was critical in helping me settle into a new identity. In my dissertation period, they provided many spaces and moments of learning that are the cornerstone to how I see myself as a researcher and educator.

There are many others in the program who have touched me in significant ways. I want to thank my colleagues and friends in this program Xinjian Qi, En Chen, and Yubao Wu for being my support, sounding board, and space for learning. Every interaction with them has been a gift, leaving me energized. I look forward to many years of our beautiful friendships.

Xinjian Qi has been my friend and confidant without whom many colors would be missing in my experience of the program. By exemplifying a personal transformation to a
place of greater joy in work and life, he has taught me that nothing is out of reach if you try. Thank you, Xinjian, for your friendship.

I want to thank En Chen for being a patient friend and colleague. En’s sincere interest in co-creating ideas has taught me not only the joy of quality work, but also the process of valuing and building on other’s inputs.

I also want to thank Yu Bao, my best man in my wedding as well as my good friend in this program. Yubao has always patiently provided his inputs, insights, and sometimes simply a space for me to reflect as I verbalized my ideas. He has not only generously shared his learning such as tricks to navigate software or the latest in our overlapping research interests, but also patiently listened to my ups and downs of life in and outside of school. Thank you, Yubao, for your contribution to my journey.

In my personal space many have facilitated my path forward. My wife, Lili, has been pivotal in making this happen. For all the times I thought otherwise, she kept me focused and motivated. In this journey, she provided me with everything else outside of content – time to experiment till I feel I get it, security and means to pursue my inquiry in the way I envision, and a patient ear to dissipate the negativities of small failures along the way. As my soul mate and lifetime partner, Lili has kept me goal oriented without allowing dysfunctional emotions to slow me down. Thank you, Lili, for walking every step of this path with me.

Finally, I want to acknowledge my parents in their unwavering support. My mother has always willingly read, shared, and heard my work and never missed an opportunity to share her pride in my every accomplishment, from as small as presenting at a conference to as big as publishing a paper. My father, as in all my learning endeavors, reminded me to
trust the process. By describing the ends as secondary to the means, he alleviated the anxieties associated with failure, and enabled me to enjoy every step. He expressed his confidence in my capabilities and has played an invaluable role in my marathon run to the finish line.

It is with much gratitude to all, many of whom I have not mentioned, that I put this bow on years of learning and joy.
Querying Graph Structured RDF Data

by

SHI QIAO

Abstract

Providing an efficient and expressive querying technique for graph structured RDF data is an emergent problem as large amounts of RDF data are available from applications in many areas. Current techniques do not fully satisfy this goal due to the nature of the RDF model which requires highly flexible use of keywords, and a structure expression in query language. Viewing RDF as graphs requires additional graph-based functionalities, such as querying a path or a tree connection. We propose a querying framework, called RDF-h, which uses the query template as a basic query unit, and supports both partially entered keywords and query conditions based on graph-structure. In order to provide efficient query evaluation, signature-based index is utilized.

Though most existing techniques which utilize signature-based index claim its benefits on all datasets and queries. The effectiveness of signature-based pruning varies greatly among different RDF datasets and highly related with their dataset characteristics. The performance benefits from signature-based pruning depend not only on the size of the RDF graphs, but also the underlying graph structure and the complexity of queries. We
propose several dataset evaluation metrics, namely, coverage and coherence, relationship specialty and literal diversity to understand the query performance differences among real and synthetic RDF datasets. Based on these results, we further propose an application-specific framework, called RBench, to generate RDF benchmarks.

By evaluating the characteristics of RDF datasets and the complexity of query templates, RDF-h selectively utilizes signature-based pruning when it is considered to be beneficial. Two aspects of RDF-h framework are evaluated in experiments: 1. extensive query performance evaluation based on randomly generated queries for different datasets; 2. utilization of RDF-h for biomedical applications. For random query evaluation, the RDF-h algorithm can automatically capture frequent user query patterns and be adjusted to maximize the benefits of signature-based pruning which provide an overall 30% query performance improvement by tuning the parameters. For biomedical applications, RDF-h provides integrated analyses of the interaction and association among different biological entities and is proven to be useful in extracting knowledge, and generating novel hypotheses for biomedical research.

Furthermore, ByteMap and neighborhood byte indexes are proposed as optimizations of RDF-h framework. The new indexes can achieve higher index space efficiency, incremental updates and better query performance. We propose the hybrid check algorithm which selectively utilizes cached check algorithm with merge check algorithm by evaluating the complexity of processing a connection edge between components. Hybrid check algorithm outperforms the original integer check algorithm for all queries on Uniprot dataset and saves more than 50% of query run time for most of the queries.
Chapter 1 Introduction

Resource Description Framework (RDF) is a metadata model and language to serve as the basis for building an infrastructure of machine-readable semantics for the data on World Wide Web [1]. It is designed as a simple data model represented by triple statements (subject, predicate and object). It can be viewed as a graph where each triple represents an edge pointing from subject to object. A wide range of data and information from structured, semi-structured or unstructured sources can be represented as RDF. Thus, it enables seamless interoperability and integration of the data on the web. Currently, large amounts of semantic data are available in RDF format in many fields of science, engineer, and business, including bioinformatics, life sciences, business intelligence and social networks. A growing number of organizations or community driven projects, such as White House, New York Times, Wikipedia and Science Commons, have begun exporting RDF data [22]. Linked Open Data Project announced 52 billion triples were published by March 2012 [22].

With the increasing popularity of RDF data, there has been great progress on querying graph structured RDF data recently. However, providing an efficient and expressive query language for such data is still an emergent problem. The main challenges of querying graph structured RDF data is due to the following: (i) RDF data is designed to have a very simple data structure by using subject-predicate-object triples without any schema, and usually
leads to extremely large data graphs with millions of nodes and billions of edges; (ii) there is a mismatch between the expressiveness of RDF which includes all kinds of information and the expressive power of query languages for RDF; (iii) the new trend of viewing RDF as graphs requires functionalities such as querying for a path or a tree connection; (iv) most of the users who are trying to query RDF data do not have the knowledge about the underlying database structure, and may lack the knowledge about RDF processing languages. Even with professional users who are familiar with RDF data, it is difficult to remember entire subject (or object) identifiers since RDF data often uses URIs to identify different resources and these URIs are typically long strings. Here, we focus on three query evaluation criteria for RDF graphs, namely, (1) flexibility and expressiveness of query templates: query templates that are flexible to specify both keywords and the graph structure; (2) utilization of the characteristics of dataset and query templates for query optimization; and (3) scalable query evaluation in order to scale to RDF datasets with millions or more triples.

To address all these challenges, we propose a query framework called RDF-h for querying graph structured RDF data. We propose a new query graph template formulation to support flexible connection expressions for graph structured RDF data and utilize different variations of neighborhood signature indexes and different query execution strategies to accelerate query performance. Based on our extensive experiments, we summarize criteria to determine the best fit index and query strategy for RDF-h framework. All criteria are based on the RDF dataset we are dealing with and the types of query templates users propose.
As observed in experiments, the efficiency of signature-based pruning differs from one RDF dataset to another. To better understand the differences among the characteristics of RDF datasets, we re-evaluated the coverage and coherence metrics proposed in [25] and confirm that current benchmarks are limited in dataset structuredness. We also identify the limitations of the coverage and coherence metrics and find that using these two metrics along is not sufficient to conclude that “current RDF benchmarks have little in common with real data”. We then propose two new RDF evaluation metrics: relationship specialty and literal diversity [12]. These two metrics provide a more effective way to differentiate the existing RDF benchmarks from real datasets.

Based on the dataset evaluation results, we identify that the existing RDF benchmarks are limited in their coverage of diverse structures and do not quite mimic the characteristics of real datasets. To reconcile these limitations, we propose an application-specific framework, called RBench [13], to generate RDF benchmarks. RBench takes an RDF dataset from any application as a template, and generates a set of synthetic datasets with similar characteristics including its graph structure and literal labels, for the required “size scaling factor” and “degree scaling factor”. Based on the design of RBench, a query generation process is also proposed to generate different types of queries systematically for any generated benchmark. We empirically show that benchmark datasets generated by RBench can achieve different scaling factors to fulfil different benchmark generation tasks, consistent with real scaling datasets, and address the limitations of the previous application-specific benchmark generator [8].

To incorporate the dataset evaluation results into RDF-h, we define parameters describing dataset characteristics and complexity of query templates to evaluate the
effectiveness of signature-based pruning on querying RDF graphs. RDF-h uses on query optimization which selectively uses signature-based pruning based on its effectiveness. A threshold tuning process is utilized to properly set parameters for RDF-h framework. By using the dataset evaluation results, we analyze the levels of expected performance gains resulting from signature-based pruning in query evaluation. By using biomedical RDF data, we demonstrate that RDF-h enables efficient processing and flexible specification of graph template matching queries and integrative querying of biomedical databases to discover complex patterns of associations among a diverse range of biological entities.

The experience with biomedical applications highlights potential improvements for RDF-h framework: 1) handling routine updates; 2) improving connectivity check performance; 3) reducing space requirements for neighborhood signature index. Thus, we propose the optimized ByteMap and neighborhood byte (NB) indexes for RDF-h. We also propose the merge check and cached check algorithms for evaluating connection edges based on neighborhood byte index. By using the Vertical Bit-Parallel storage, the new connectivity check algorithms achieve early termination and skip of unmatched index entries. By analyzing the query evaluation results, an improved connectivity check algorithm, hybrid check algorithm is proposed which selectively utilizes cached check algorithm by evaluating the complexity of processing a connection edge between components.

The rest of the thesis is organized as follows. In chapter 2, we introduce the related work in the literature. RDF-h query framework is discussed in chapter 3. Chapter 4 introduces the dataset evaluation metrics for characterizing different RDF datasets. The application specific RDF benchmark generator, RBench, is proposed in chapter 5.
Utilization of RDF-h for Biomedical Applications is demonstrated in chapter 6. Chapter 7 shows the optimized indexes and algorithms for RDF-h. We discuss some potential extensions for RDF-h as future work in chapter 8 and chapter 9 concludes.
Chapter 2 Related Work

Querying RDF data is a hot research area and lots of work have been proposed during the last decade. In a broader view, we present related work in two research directions most relevant to this thesis:

1. Querying techniques or languages;
2. Indexes for RDF graphs.

2.1 Querying Techniques for RDF

Querying techniques for RDF datasets focus on providing a query language or query interface which allows users to specify and evaluate certain types of queries on RDF datasets. Here, we mainly focus on two types of queries: 1. keyword based queries; 2. graph template queries. For keyword queries, the users propose a set of interesting keywords to the query engine and the query engine returns a small subgraph of the RDF graph which contains all the keywords with structure information. For graph template queries, the users propose a graph template to the query engine and the query engine returns all subgraphs of the RDF graph which are isomorphism to the graph template.

2.1.1 Keyword based Queries

Keyword based queries are the most straightforward and popular way to query RDF datasets as they allow users to access the graph databases using simple keywords rather
than ask them to learn a structured query language and understand complex and possibly fast evolving data schemas. Although the idea of keyword based queries is simple, there are many sub-problems involved in keyword based querying techniques: query result definition, ranking functions, snippet generation, result clustering, query cleaning, performance optimization and search quality evaluation.

A common assumption to rank the results of a keyword based query is that users are more interested in the subgraphs with close connections (for a weighted RDF graph), i.e. the minimal weight subgraph with all keywords connected. This leads to a NPC problem: find the optimal Steiner tree for a set of nodes. Rather than returning the optimal Steiner tree solution, BANKS [BH+02] and STAR [KM+09] focus on providing good approximate solutions to this problem. BANKS algorithm identifies a central node from which the user can browse the rest of the database and cover all keywords. This central node represents the closest relationship to all other nodes and thus it represents the best answer for the keyword query. The algorithm runs Dijkstra’s shortest path with each keyword node and outputs a solution if a common node is reached by all Dijkstra’s algorithms which cover all keywords. The BANK algorithm works better if the optimal Steiner tree is balanced and the keyword nodes tend to have a common center. STAR is the following work of BANK and the approximation algorithm proposed in STAR outperforms the BANK algorithm for both result subgraph weight and query running time. STAR algorithm begins with quickly building a first tree that interconnects all keywords using breadth first search instead of Dijkstra’s algorithm. In the second step, it aims to iteratively improve the current tree by scanning and pruning its neighborhood.

Other techniques propose different types of subgraphs to answer keyword based
queries rather than Steiner trees. Community [QY+09] uses multi-centered graphs as results for keyword based queries. It finds an induced subgraph which contains all the keywords and has some common centers connecting to these keyword nodes with bounded hops, called radius. The authors propose two enumeration algorithms, COMM-all and COMM-k, for supporting enumerating all and top-k communities with polynomial delay. R-clique [KA11] extends the idea of Community to find a loose clique based on a specified radius connecting all keywords. The result subgraph of R-clique is not a real clique but a generalized form of clique. The keyword nodes act as nodes in the ‘clique’ and edges are the path among keyword nodes limited by distance.

2.1.2 Graph Template Matching

The standard way to query RDF datasets is to use the default RDF query language, called SPARQL [PA+09]. SPARQL queries are defined by basic graph patterns, which are conjunctions of simple access patterns. A simple access pattern is a triple whose elements are any combination of atoms or variables, where variables are prefixed by a “?” . Atoms in simple access patterns are called bound patterns, and variables are called unbound patterns. A simple access pattern selects triples that match all of its bound and unbound patterns. The SPARQL query is then evaluated by extensive triple joins. In graph view, evaluating SPARQL query is graph template matching based on graph isomorphism implemented as edge joins. Though SPARQL is described as a graph-matching query language, most of the database management systems which utilize SPARQL build triple centered indexes (indexing different permutations of the three dimensions of triples) rather than graph based indexes.
Jena [WC+03] is the RDF database management system for Java. It manipulates RDF graphs stored in main memory. The database management system is implemented using a SQL database through a JDBC connection. RDF triples are stored in the database-backend of Jena as three tables. The URIs of resources are indexed in one table and large literals are stored in another table. Triples are stored in the statement tables using indexes for URIs and large literals. Jena supports SPARQL queries through its own query language RDQL. It converts the SPARQL queries into a set of find patterns of RDQL and executes them as joins.

3store [HG03] follows the ideas of Jena which is implemented on top of the MySQL. It supports inferential queries that are mainly evaluated by MySQL queries. To provide storage efficiency, 3store uses hashing to translate URIs into an internal form of representation. The query language of 3store is RDQL defined in Jena. RDQL expressions are translated into relational calculus with constraints. The relational calculus expressions are then translated into SQL executed by MySQL.

Virtuoso [EI07] stores all triple statements in a four-column table with one optional field. The main idea of the Quad storage (subject, predicate, object and graph) for RDF data management is to exploit existing relational techniques to support features specific to RDF data. Virtuoso directly translates SPARQL queries into SQL queries during parsing. It uses sampling during query translations to estimate the cost of alternative plans to optimize the query performance.

RDF-3X [NW08] indexes all 6 permutations of the three dimensions of RDF triples (Subjects, Predicates and Objects). The indexes are compressed using a byte-wise method (only indexing two dimensions of RDF triple) to improve query performance. The query
engine focuses on optimizing the triple joining orders. The optimization uses selectivity
statistics calculated for the given queries using selectivity histograms and statistics of
frequently accessed paths.

Other graph template matching techniques focus on providing sophisticated graph-
based indexes and will be introduced in next section.

2.2 Indexes for RDF Graphs

As graph template matching based on subgraph isomorphism is a NPC problem, many
techniques for querying RDF using graph templates focus on building different types of
indexes to improve the query performance. Though most studies is not RDF specific, they
all apply to graph structured RDF data. Based on indexing strategies, these studies can be
categorized into three groups: feature graph pattern index, neighborhood signature index
and graph partitioning index.

2.2.1 Feature Graph Pattern Index

The motivation of feature graph pattern indexes is to utilize small subgraphs which are
frequently queried as basic index units and to accelerate subgraph isomorphism matching
by joining matching candidates of the indexed subgraphs. The performance of feature
graph pattern indexes highly depends on the choices of the indexed subgraphs.

In GraphGrep [GD02], it selects paths as feature graph and proposed a path-based
indexing table which enumerates all paths up to a threshold length from the data graphs.
The “path-representation” of a graph is the set of label-paths in the graph, where each label-
path has a set of id-paths. These label paths are stored in a hash table whose keys are the
hash values of the label paths. Since all paths up to a threshold length are enumerated in
GraphGrep, the index is not applicable for single large graphs with only thousands of nodes.
In order to resolve the inefficiency of indexing all paths as in GraphGrep [GD02], other work tries to mining all frequent graph patterns as feature graphs and build the index based on them. Indexing frequent graph patterns as basic units reduce the index space as well as improving the filtering rate but selecting optimal frequent subgraphs is an NPC problem.

gIndex [YY+04] utilize discriminative power of the subgraphs to select frequent structures as index patterns. The evaluation begins with identifying the graphs in the database which contain the frequent subgraphs of query template. Subgraph isomorphism is then utilize to verify the matching candidates. The problem of gIndex is that it checks all candidates matching any subgraph of the query template which generates a large search space.

TreePi [ZH07] follows the idea of gIndex which utilizes frequent subtrees as feature graph rather than subgraphs to build the index. To provide more compact index, the feature subtrees can be selected based on a support threshold function and discriminative power. The canonical form of trees is utilized to translate labeled trees into unique string representations. The matching candidates of the query graph template is significantly reduced by filtering the graphs according to the feature-tree-partitions.

GADDI [ZL09] uses a hybrid indexing idea. The core of GADDI is based on neighboring discriminating substructure (NDS) distance which finds frequent subgraphs of intersecting subgraphs for each pair of nodes. Though the query performance of GADDI is better than GraphGrep, the scalability of GADDI is limited by the index construction time and the size of the index.
2.2.2 Neighborhood Signature Index

Another index strategy is to index the neighborhood information of all nodes in the graph to prune unnecessary candidates. The idea of neighborhood signature pruning is to check whether the labels of neighbors of the query node is contained in the labels of neighbors of its matching candidates. The neighborhood information is obtained by breadth first search and stored in varied formats.

In TALE [TP08], authors proposed an approximate matching algorithm for large query graphs based on NH-Index which records the degree, neighbor connections (number of edges between neighbors) and the labels of neighbors for each node in the data graph. To ensure linear growth of size of NH-Index with data graph size, a compact bit array where each bit represents a unique label in data graph is utilized.

SAPPER [ZY+10] addresses the same problem as TALE, which matches spanning trees of the query graph based on a hybrid neighborhood unit index. The index unit of SAPPER extends NH-Index as: 1). index 2 hops of neighbors rather than 1 hop; 2). utilize bloom filter to index labels of neighbors.

GraphQL [HS08] uses the neighborhood within distance r as a pruning criterion. The labels of neighbors are indexed directly as a sequence of node labels in lexicographic order rather utilizing a bit array or a bloom filter. GraphQL uses a greedy strategy based on the cardinality of intermediate joined results to optimize the joining order.

SPATH [ZH10] improves GraphQL [HS08] by introducing a more deliberate neighborhood signature and utilizing shortest paths as the basic join units to answer subgraph queries. The neighborhood signature in SPATH group the same labels of neighbors for each node into one entry and counts the occurrences of each label as a pruning
gStore [ZO+14, ZO+11] (i) extends the utilization of signature-based pruning to RDF databases which stores the RDF graph in the form of adjacent lists, (ii) uses the neighborhood signature as a bitstring according to the adjacent edge labels and node labels, and (iii) indexes all vertex neighborhood signatures using a special index schema, VS tree, to provide efficient query evaluation.

### 2.2.3 Graph Partitioning Index

The motivation of graph partitioning index is divide and conquer. By partitioning the large graph into smaller subgraphs, the query template is evaluated in these subgraphs more efficiently.

GRIN [UP+07] uses graph partitioning and distances in the graphs as basis for indexing for graphical queries. The idea of GRIN is to draw circles around selected “center” vertices in the graph where these circles encompass those vertices in the graph that are within a given distance of the “center” vertex. It builds a balanced binary tree structure to store all these circles efficiently. The number of leaf nodes of GRIN index is $O(|N|)$ and the height of the tree is $O(log_2 |N|)$ where $|N|$ is the number of nodes in RDF graph. This leads to a worst case complexity for building the index of $O(|N|^4 log_2 |N|)$. Query time complexity of GRIN index ranges from $O(|N|)$ in the best case to $O(|N|!)$ in the worst case.

DOGMA [BP+09] follows the idea of graph-partitioning indexing structure of GRIN but the index is designed as disk-resident. Compared with GRIN, DOGMA uses k-merge graphs to divide the RDF graph into small regions rather than circles which is more suitable for disk-oriented indexing.
Chapter 3 RDF-h

In this chapter, we present the RDF-h [QO-1] query framework. The main challenges of querying graph structured RDF data is due to the following: (i) RDF data is designed to have a very simple data structure by using subject-predicate-object triples without any schema, and usually leads to extremely large data graphs with millions of nodes and billions of edges; (ii) there is a mismatch between the expressiveness of RDF which includes all kinds of information and the expressive power of query languages for RDF; (iii) the new trend of viewing RDF as graphs requires functionalities such as querying for a path or a tree connection; (iv) most of the users who are trying to query RDF data do not have the knowledge about the underlying database structure, and may lack the knowledge about RDF processing languages. Even with professional users who are familiar with RDF data, it is difficult to remember entire subject (or object) identifiers since RDF data often uses URIs to identify different resources and these URIs are typically long strings. Thus, we focus on three important query evaluation criteria for RDF graphs, namely, (1) flexibility and expressiveness of query templates: query templates that are flexible in terms of both specifying keywords and the graph structure; (2) utilization of the characteristics of dataset and query templates: query optimization which takes the features of RDF graphs and the complexity of query templates into consideration is necessary; and
(3) scalable query evaluation: query evaluation needs to scale to RDF datasets with millions or more triples.

One focus of RDF-h is to study the effectiveness of signature-based pruning for querying graph-structured RDF data using graph templates. Signature-based pruning has been used extensively to improve performance of graph template matching (based on subgraph isomorphism) problem. Many different variations of neighborhood signature indexes have been developed [HS08, SW+12, ZY+10, ZH10, ZO+11]. Whether signature-based pruning is beneficial for queries on RDF datasets, there are several factors need to be considered. First, RDF graphs have unique node labels and use URIs to identify millions of different resources which are typically long strings. To be applicable, partial(ly entered) keywords should be supported in specifying RDF query templates. Thus, the design of neighborhood indexes and the procedure of neighborhood containment check in query processing should handle queries with partial keywords. More importantly, the effectiveness of signature-based pruning highly depends on the characteristics of datasets and the queries [LH+12]. The underlying graph structure of RDF datasets can range from strict relational like structure to arbitrary graphs for different applications. Thus, selectively using signature-based pruning for different RDF graphs and queries needs to be explored.

3.1 Template Matching for RDF graphs

In our context, we define graph template matching for RDF graphs with a flexible query template supporting paths, distance constraints, and partial matching of keywords as follows:

DEFINITION 1.1: An RDF Graph is a directed graph \( G = \{V, E, l, f\} \) where \( V \) is a
Figure 1. RDF Graph and Query Template Example with Matching Results
set of vertices representing either subjects, objects or both. \( E \subseteq V \times V \) is a set of directed edges representing predicates pointing from subjects to objects. \( l \) is a label set for subjects, objects and predicates. \( f : V / E \rightarrow l \) denotes the mapping function between vertices/edges to labels.

**Definition 1.2**: *Connection edge* \((\mathcal{E})\) represents a path \( \omega_{i,j} \) between two nodes \( n_i \) and \( n_j \), where \( \omega_{i,j} \) can be one directional or bi-directional. Expression \( E \) describes the distance constraints of \( \omega_{i,j} \) (Distance is the length of the shortest path).

**Definition 1.3**: A *Query Template* in our framework is a directed graph \( G_q = \{V,E\} \) over an RDF graph \( G \), where vertices (objects or subjects) are labeled by partial keywords (that are substrings of labels in the label set \( l \) of RDF graph \( G \)), and edges represent predicates or connection edges (edge label is not required).

**Definition 1.4**: Given RDF graph \( G = \{V,E,l,f\} \) and a query template \( G_q = \{V,E\} \), *Template Matching* finds all subgraphs of \( G \) that satisfy both structural and label constraints in \( G_q \).

An example query template is shown in Figure 1(b) on RDF graph of Figure 1(a). The query template finds the title of a paper authored by “Philip S.Yu” which is published in VLDB conference, and has a connection within 4 hops of another paper authored by “Jiawei Han” with additional information. Using the RDF graph in Figure 1(a), two matches are shown in Figures 1(c-d). The first result is a paper directly referenced by the second paper from “Jiawei Han”, and the second result is a paper which is connected to it with a 2 hop path. Connection edges are also used between Paper ID and its author name in the query template to handle Ontology nodes (People ID) between them. This example demonstrates the uses of (i) connection edges in query templates and (ii) partial keywords.
for querying data instead of detailed labels. Similar query templates with connection edges are also used elsewhere [FL+10, MC+11], with the differences: (i) multi-attribute labels are used, but partial keyword is not supported; (ii) graph simulation is used, instead of graph isomorphism.

<table>
<thead>
<tr>
<th>HashID</th>
<th>NodeLabel</th>
<th>ID</th>
<th>Distance</th>
<th>ID Interval</th>
<th>Count</th>
<th>IDlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Albert_Austin</td>
<td>1</td>
<td>1</td>
<td>2-5</td>
<td>3</td>
<td>2, 4, 5</td>
</tr>
<tr>
<td>2</td>
<td>Albert_Band</td>
<td>1</td>
<td>1</td>
<td>7-7</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Bruce_Abbott</td>
<td>1</td>
<td>2</td>
<td>3-10</td>
<td>3</td>
<td>3, 9, 10</td>
</tr>
<tr>
<td>5</td>
<td>Bruce_Baron</td>
<td>1</td>
<td>-1</td>
<td>12-14</td>
<td>2</td>
<td>12, 14</td>
</tr>
<tr>
<td>6</td>
<td>Claude_Akins</td>
<td>1</td>
<td>-2</td>
<td>13-13</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Don_Lake</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 2. Indexes Examples**

### 3.2 Indexes

Two indexes namely, IDMap and NI (neighborhood interval) Indexes (shown in Figure 2) are utilized to support efficient evaluation of partial keywords and connection edges.

**DEFINITION 2.1:** Given an RDF graph \( G \), an **IDMap** Index is a mapping function \( H: l_i \in l(G) \rightarrow \{j | j \in Int\} \) with two properties: 1. The set of IDs for all RDF labels forms an interval of consecutive integers; 2. IDs of labels are assigned in lexicographic order.

IDMap Index basically maps RDF labels into integer IDs in lexicographic order. For partial keywords specified as prefixes of RDF labels, the look-up time is \( O(logN) \), where \( N \) is the total number of RDF labels, since all matching IDs form one interval of consecutive integers. Other indexes can also be built to accelerate look up time of partial keywords.

**DEFINITION 2.2:** Given two vertices \( n_i \) and \( n_j \) in graph \( G \), if there is a directed path from \( n_i \) to \( n_j \), \( n_j \) is a **forward neighbor** of \( n_i \) and \( n_i \) is a **backward neighbor**
of $n_j$.

The NI index materializes the results of BFS with one or more hops for the nodes in the graph. It extends the neighborhood signature index in SPATH [ZH10] by group labels of neighbors into ID intervals to provide similar pruning criteria. The NI index in our context is a table with five columns: for any node $n_i \in G$, it contains ID of $n_i$, Distance, Label ID interval, Number of indexed neighbor nodes in this entry, and neighbor node IDs. The Distance is the length of the shortest path from $n_i$ to the indexed neighbor node. The positive (negative) distance indicates that the indexed node is a forward (backward) neighbor. There are two pre-defined parameters for the neighborhood index: maximum indexed distance $d_{\text{max}}$ and binning factor $m$. The neighbor nodes sharing the same distance are grouped together, ordered by their IDs, and partitioned into rows by the binning factor $m$, which limits the maximum number of indexed neighbor nodes in each index entry. For each index entry, a Label ID interval is directly generated from neighbor nodes’ IDs.

The NI index accommodates unique node labels of RDF graphs and compact its size by binning factor $m$. Increasing the value of $m$ decreases the indexing space as less indexing entries, but increases the risk of reading more neighbor nodes from each entry. The maximum indexed distance $d_{\text{max}}$ dramatically influences the space of neighborhood index by including more indexed neighbor nodes for each node. However, large $d_{\text{max}}$ means more local information indexed which improves both the pruning power and the process time for connection edges with long distance. Tradeoff between the costs of NI index (in terms of the space needed) and its benefits on improving the query performance are explored in later.
3.3 RDF Query Framework

There are eight major modules in graph by example query engine framework (shown in Figure 3), six of which are required for all query templates and two are optional. In the following subsections, we will go through some of these modules.

3.3.1 Query Preprocess

Query preprocessing separates connection edges from query templates, which may decompose the query templates into several connected components. The main process first identifies all matches of the connected components (template matching), and then evaluates the connection edges among them. Evaluating connection edges is more complex than evaluating an edge even if BFS results in neighborhood index are utilized since they allow different direction specifications and support structural constraints. The common strategy, pushing the time consuming processes later and doing further query candidate pruning first also applies here.
3.3.2 Candidate Generation

The “Candidate Generation” module tries to find all possible matching candidates for each query node. The simplest process is directly based on query nodes’ keyword matching. We use IDMap Index to find all possible matching candidates for partial keywords in the template. If the partial keywords are prefix of some RDF graph labels, a binary search on Index table can be utilized to find the lower bound ID and upper bound ID to generate all possible candidates IDs. Otherwise, a table scan is needed. For large graphs with billions of nodes, extra indexes may be needed to accelerate partial keyword matching process. In this paper, we only consider partial keywords which are prefixes of RDF graph labels for simplicity.

Using only keyword matching to generate candidate sets for all query nodes may lead to poor query performance, as also observed in [ZH10]. In order to further prune candidate sets for all query nodes and improve the query performance, we build the “Neighborhood Check” module interacting with “Candidate Generation” module.

3.3.3 Neighborhood Check

The neighborhood containment check for NI index is defined as ID interval check (here, we assume partial keywords are specified as prefixes of node labels).

DEFINITION 3.1: Given a partial keyword \( p_j \) of any query node \( q_j \in V(G_q) \), the ID Interval, \( Z_j \), is all IDs of node set \( N_j \) where \( \forall n_t \in N_j \), the label \( l_t \) of \( n_t \) is a valid match of \( p_j \).

Algorithm 1: Neighborhood Check \((n_i, q_j)\)

Input: \( n_i \in V(G), q_j \in V(G_q), \) ID Intervals \( Z^*, \) \{Distance, Count\} \( \psi_j^* \), NI Index \( NI_i \)
for node $n_i$

Output: If $n_i$ pass neighborhood check of $q_j$, return true; Otherwise, return false

1. FOR all $k$ where $|k| \leq d_{max}$

2. FOR any partial keyword $p_k$

3. FOREACH value pair $\{d, c\}$ in $\psi^k_j$

4. Exact all entries from $N_{Ii}$ where ID interval intersect with $\mathbb{Z}_k$ and Distance $\leq d$

5. Count all IDs in $\mathbb{Z}_k$ as $c'$

6. IF $c' \leq c$, RETURN false

7. RETURN true

DEFINITION 3.2: Given any node $n_i \in V(G)$, the $K$-Neighbor of $n_i$, denoted as $Neighbor_k(n_i)$, is a set of nodes where, if $k$ is positive, $\forall n_j \in Neighbor_k(n_i)$, there is a directed path from $n_i$ to $n_j$ with no more than $|k|$ hops; if $k$ is negative, $\forall n_j \in Neighbor_k(n_i)$, there is a directed path from $n_j$ to $n_i$ with no more than $|k|$ hops.

DEFINITION 3.3: Given a node $n_i \in V(G)$, and a query node $q_j \in V(G_q)$, $n_i$ passes the Neighborhood Check of $q_j$ if $\forall q_k \in Neighbor_k(q_j)$, ID Interval $\mathbb{Z}_k$ uniquely contains ID of any $n_g \in Neighbor_k(n_i)$, for all $|k| \leq d_{max}$.

Each partial keyword in query template is checked by IDMap index to form one interval of consecutive integers. Since query templates can contain query nodes with the same partial keyword, value pairs as $\{\text{Distance, Count (total appearance within Distance)}\}$ for each partial keyword are maintained for each query node. The neighborhood check is performed based on partial keywords one by one, and the count of occurrences of this
A partial keyword is taken into consideration. The term “uniquely contains” in the definition 3.3 means that the node $n_g$ cannot be used to match more than one ID interval. If one partial keyword contains another partial keyword, Count in value pairs is updated. Algorithm 1 shows the neighborhood check process, \{Distance, Count\} pairs associated with query node $q_j$ and partial keyword $p_l$ are denoted as $\psi_j^l$.

Neighborhood check based on NI index is optimized for partial keywords: 1) only index entries with Label ID interval intersecting with the ID interval of the partial keyword needs to be retrieved; 2) all IDs in the index entries are valid matches for the partial keyword if the ID interval of the partial keyword contains the Label ID interval.

**Algorithm 2: Component Match**

**Input:** Query Component $G_c$, All Query Node Candidate Sets $C_{q_i}$, RDF Graph $G$

**Output:** Decomposed D-trees $T_{olt}$, All Tree Candidates $C_t$, Tree Join Queue $Q_{olt}$, Component Candidates $C_{G_c}$

//Decompose Query Component into 1 Level D-trees

1. While ($G_c$ != $\emptyset$)
2.     If $T_{olt}$ is Empty
3.         Pick $E(q_i, q_j) \in G_c$ with largest $S(q_i) + S(q_j)$
4.         Add D-trees $t_i$ and $t_j$ into $T_{olt}$
5.         Remove All Edges in $t_i$ and $t_j$ From $G_c$
6.     Else
7.         $nextTopRoot(G_c, T_{olt})$ as $q_k$
8.         Add $t_k$ to $T_{olt}$
9.         Remove All Edges in $t_k$ From $G_c$

//Find Candidates For each D-tree

10. Foreach $t_i \in T_{olt}$
11.     Foreach node $n_i \in C_{q_i}$
12.         Add $n_i$ to $C_{n_i}$
Foreach $E(q_i, q_j) \in t_i$ (or $E(q_j, q_i) \in t_i$) 

Intersect $\text{Neighbor}_1(n_i)$ (or $\text{Neighbor}_{-1}(n_i)$) with $C_{q_j}$ as $C_{ij}$ 

$C_{n_i} = \text{Join}(C_{n_i'}, C_{ij})$

$C_{t_i} = \text{Union}(C_{t_i'}, C_{n_i})$

//Join All D-tree Candidates Together

Foreach $t_i \in T_{olt}$

If $C_{G_c} == \emptyset$

$C_{G_c} = C_{t_i}$

Else $C_{G_c} = \text{Join}(C_{G_c}, C_{t_i})$

---

### 3.3.4 Component Matching

We use 1 level D-tree as basic join unit to process component matching shown in Algorithm 2. There are 3 main steps in it: query decomposition (line 1-9), D-tree candidate generation (line 10-16), and D-tree candidates join (line 17-20). The decomposition process has great influence on query performance and determines the complexity of next two steps’ work since our algorithm generates all matching candidates for each decomposed D-tree and joins them together to get matching results. The time complexity of our algorithm is proportional to $\prod_{i=1}^{K} |C_{t_i}|$ where $K$ is the number of all decomposed 1 level D-trees and $|C_{t_i}|$ is the number of matching candidates for each D-tree. Clearly, in order to improve query performance, we prefer fewer decomposed D-trees ($K$ is small) and less matches for each tree ($|C_{t_i}|$ is small). 1 level D-tree decomposition can be proven to be equivalent with the vertex cover problem and we find the optimal solution can be NP hard. Similar as 2-approximation vertex cover algorithm [CL+09], we use the 2-approximation algorithm to generate 1 level D-tree decomposition of query component by recursively picking an edge
\((q_i, q_j)\) from it and adding 1 level D-trees rooted at \(q_i\) and \(q_j\) into result. Same as vertex cover problem, choosing large degree nodes first can possibly yield better result since D-trees rooted at these nodes can cover more edges in query component. Also, the number of matching candidates for a D-tree greatly depends on the number of candidates for its root node. Thus, we define selectivity value function \(S(q_i) = \frac{\text{deg}(q_i)}{|C_{q_j}|}\) which takes both query node’s degree and its corresponding candidate set size into consideration as a good measurement of the priority to be selected as root nodes. In order to find good join order for all decomposed D-trees, it is better to choose a decomposition order that the root of next decomposed tree is bounded by existing decomposed tree nodes. This is implemented by function \(\text{nextTopRoot}(G, T)\): it scans all nodes of existing decomposed trees and picks the node with highest \(S(q_i)\).

After we get all decomposed 1 level D-trees, we try to find all candidate matches for them by using neighborhood index. Each candidate match is a Dictionary with query node ID as key and original node ID as value. We define two important functions as \(\text{Union}(C_i, C_j)\) and \(\text{Join}(C_i, C_j)\). \(\text{Union}(C_i, C_j)\) is same as set union which combines candidates together. \(\text{Join}(C_i, C_j)\) is a function which combines all keys from two candidate sets by evaluating the predicate: all shared keys of two candidate matches should have equal values to join. Here, \(\text{Union}\) function is used to combine matches for D-tree \(t_l\) rooted at different nodes together in line 16 while \(\text{Join}\) function is used to combine matches for D-tree \(t_l\) into existing joint pieces.

A similar approach is used in STWIG [SW+12] with the differences: 1) D-trees are used as basic join units; 2) new selectivity function is defined based on the size of candidate sets; 3) the NI index is used to generate all D-tree candidates; 4) tree join order is
Algorithm 3: Connectivity check \((n_i, n_j, d_c)\)

Input: \(n_i \in V(G), n_j \in V(G)\), Maximum Path Distance \(d_c\), NI Index \(NI_i\) and \(NI_j\) for node \(n_i\) and \(n_j\)

Output: If there is a path from \(n_i\) to \(n_j\) with distance no greater than \(d_c\), return \text{true}; Otherwise, return \text{false}

1. Extract entries from \(NI_i\) where \(0 < \text{Distance} \leq \text{Ceil}(\frac{d_c}{2})\) and combine all IDlists as \(\psi_i\)

2. Extract entries from \(NI_j\) where \(\text{Ceil}(\frac{d_c}{2}) - d_c \leq \text{Distance} < 0\) and combine all IDlists as \(\psi_j\)

3. Intersect \(\psi_i\) and \(\psi_j\) as \(\psi_{ij}\)

4. If \((\psi_{ij} \neq \phi)\)

5. Return \text{true}

6. Else Return \text{false}

3.3.5 Connectivity Check

Connection edges can be either inside of one query component or joining two different components. If the edge is inside of one component then the connectivity check is used to prune the component candidates. Otherwise, connectivity check acts as the predicate to determine whether the two component candidates can join or not. For inside component connection edges, the number of connectivity checks is exactly the size of the component candidate set. For a connection edge between components, the number of connectivity checks depends on the product of the sizes of components’ candidate sets. In the worst case, if we have a sequence of \(N\) components to be joined by connection edges, the number of connectivity checks that need to be performed can be as large as \(\prod_{i=1}^{N} |C_{G_c}|\). In order to improve query performance, two rules are utilized to determine the order to process
connection edges: 1) connection edges inside components are processed before connection edges between components; 2) connection edges between components are processed in the order of the smallest product of candidate sets first. As shown in Algorithm 3, processing connection edges is based on the NI index. Here, we assume the maximum indexed distance $d_{max}$ of neighborhood index is greater than $Ceil(\frac{d_c}{2})$, where $d_c$ is the distance specified with connection edge. Otherwise, we need to combine more index entries of $n_i$’s neighbor nodes together with $I_i$ in order to get more hops of neighborhood information for $n_i$.

3.3.6 Query Result Generation

Once we get final matches for our query template, we use “Display Final Results” module to show our query results to users. We don’t instantiate any paths for the connection edges in the original graph until now. For each final matching result, if we generate possible paths for each connection edge in it, the number of results we need to return to users become very huge. It adds extra difficulty for users to understand and capture interesting matches. In addition, path generation needs a lot of time to process since we need to do it for every final match result and all connection edge. It will slow down our query process greatly. Thus, we decide to delay the path generation process and give users options to choose which final results they are interested in and which connection edges they want to instantiate. Another design decision is needed for which paths will be returned for a connection edge in a specified final match. It might be all paths with length less than or equal to $d_c$. However, it has some drawbacks: 1. if users set large $d_c$ values for connection edges, there will be too many different paths even for a single path generation for one connectivity operator in one match; 2. if users set small $d_c$ values, there might be
no final matches for query template; 3. generating all paths between 2 nodes up to length $d_c$ is too time consuming. In this paper, we assume users are more interested in finding all shortest paths when they instantiate a connectivity operator. Finding all shortest paths between two nodes $n_i$ and $n_j$ can be implemented by a recursive function. Each step, an additional hop of neighbor nodes of $n_i$ or $n_j$ are combined by looking at their neighborhood index and then find the intersection of two sets of neighbors of $n_i$ and $n_j$. Once two neighbor sets intersect with each other, for each intersection node $n_k$, the function to find all shortest paths between $n_i$ to $n_k$ and $n_k$ to $n_j$ is called recursively.

Obviously, the time complexity of this algorithm depends on how many index entries are extracted from the neighborhood index. Since the number of index entries is linear with the number of indexed neighbor nodes, extracting as few neighbor nodes as possible is preferred. We use a balanced growth of neighborhood to optimize path generation process similar to [KM+09] which always extracts neighbors for the node with fewer already combined neighbors.

### 3.4 Execution Plans

Based on the query engine framework we presented, many different query strategies can be considered. As seen from Figure 3, the most straightforward execution of template matching is directly going through all required modules without using neighborhood check process. As another extreme, the “neighborhood check” module can be enforced to be executed for all queries before “query graph decomposition” module to further prune candidates for each query node. Additionally, since neighborhood index can index up to $d_{max}$ hops of neighbors for each node, neighborhood check process based on different $d_{max}$ value can be performed. Obviously, there is a tradeoff problem between the space
efficiency and query time performance for different $d_{max}$ value. Another observation is
the neighborhood check will not always improve the template matching process. Since
neighborhood check process brings time overhead to total query execution time, it might
hurt the query performance if the pruning effect of neighborhood check to reduce the size
of candidate sets is negligible. In the following subsections, we go through all these
different strategies and justify their value based on different assumptions.

### 3.4.1 Baseline: 1 Level Tree

The most straightforward strategy, “1 level tree”, which acts as our baseline execution
plan is directly going through the middle 5 modules with “connectivity check” module in
Figure 3. As only IDMap index and 1 hop NI index is required by “candidate generation”
and “component matching” modules, which are all indexes needed for “1 level tree”
strategy. The size of ordered hash index depends on the number of unique RDF labels
which is same as the number of nodes in RDF graph. As we state before, the space of
neighborhood index is $O(N\left(\frac{\mu}{2}\right)^{d_{max}})$ where $N$ is the number of vertices in $G$, $\mu$ is the
average node degree and $d_{max}$ is the maximum hops of neighborhood index. Additionally,
since we use binning strategy to compress our neighborhood index, if we set binning
factor $m = \frac{\mu}{2}$, the size of 1 hop neighborhood index will be same as $O(N)$. The total index
space for “1 level tree” is $O(N)$ which has good scalability. However, the query
performance is a concern if our query templates have very general keywords and complex
graph structure. Then, without taking the structure information in query template into
consideration, the number of intermediate tree joins can be very huge to process in
reasonable response time (at most seconds). In that situation, using neighborhood check to
prune unnecessary candidates for each query node is needed. Another major drawback of
“1 level tree” is it can’t solve connectivity operators efficiently since only 1 hop NI index is built.

3.4.2 Plans with Neighborhood Check

The second strategy is based on “neighborhood check” process and enforces all queries to go through “neighborhood check” module. Utilizing only label matching to generate candidate set for each query node might generate too many unnecessary candidate nodes and these unnecessary candidates might lead to expensive costly intermediate tree generation and join process. The neighborhood information can be used to check the validity of these candidates and prune the candidate sets. This process is explained in 3.3.3. However, one major concern for this strategy is the space efficiency due to poor scalability of NI index especially if we want to use more hops of neighborhood information to improve its pruning power. Even with binning technique to compress index entries and IDMap index, NI index still needs a lot of space. However, observe that using large $d_{max}$ value is not practical for most real RDF graphs and in this paper we use $d_{max} \leq 3$.

3.4.2.1 2 Hops Check and 3 Hops Check

“2 hops check” and “3 hops check” utilize 2 and 3 hops of NI indexes to perform the neighborhood check process. The more hops of neighborhood indexed and checked, the more possibility to prune unnecessary candidate nodes for each query node at the expense of more space to store the neighborhood index. If we compare these two strategies with “1 level tree”, the benefits of using neighborhood check process are: 1. reduced size of candidate set for each query node; 2. fewer number of candidates generated for all decomposed 1 level trees; and 3. reduction in the number of intermediate joins of all 1 level
tree candidates. All these benefits are built on the extra expense of index space and time overhead from neighborhood check process. The decision whether to choose “2 hops check” or “3 hops check” mainly depends on two factors: (i) RDF data sets (ii) The complexity of frequent query templates that are used. More specifically, as also indicated in [DK+11], different RDF datasets may have very different features, including indegree, outdegree, number of subjects, and these features have big influence on space size of neighborhood indexes. The complexity of frequent query templates users propose to our system also has an impact on the choice of index. The complexity of graph structure and generality of partial keywords are two aspects we consider as the complexity of query templates. More complex graph structure leads to more 1 level D-trees after the decomposition, and availability of more hops in the neighborhood index improves the performance. While more general partial keywords directly implies more candidates matches for the query nodes, which leads to more intermediate tree candidates and more intermediate joins. All these have impact on the utility of neighborhood check process.

3.4.2.2 Vertex Cover Check

In order to reduce the space required for neighborhood indexes, multi-hops neighborhood indexes can be combined with the vertex cover idea. The strategy is, to index more hops in the NI index for the nodes in the vertex cover than those which are not in the vertex cover. The motivation behind this heuristic is that the total space required is less than having the same number of hops indexed for all nodes, and for any edge in an intermediate matching result of a template, at least one node in the edge is checked by more hops of neighborhood check process. Obviously, if we have fewer nodes in vertex cover
set, we need index fewer nodes with more hops of neighborhood which might lead to smaller index space. Since finding optimal vertex cover is NP hard and optimal vertex cover solution is not equivalent with optimal index space, an approximation generating a relatively small vertex cover set will be sufficient. We consider the 2-approximation vertex cover algorithm [CL+09] which recursively picks an edge, adds both nodes in this edge into vertex cover set and removes all edges adjacent to them until no edges is left in the original graph. As opposed to picking edges randomly, we tried several other greedy edge selection alternatives but the improvements are negligible. Experimental results show that the 2-approximation vertex cover algorithm based on selecting random edges is best among the other alternatives considering both space and time costs. The “vertex cover check” strategy tries to provide a compromise between the reduced space requirements and better neighborhood check performance with the additional cost of handling two types of vertices.

3.5 Signature-based Pruning Evaluation

The main advantage of signature-based pruning is the reduction in the number of intermediate candidates and joins. This advantage comes with extra space cost, and the time overhead of performing the neighborhood check process. The more hops of neighbors it checks, the more time the neighborhood check takes. Since the effectiveness of signature-based pruning varies by the datasets and queries used, evaluating its benefit and cost to use it selectively is important.

3.5.1 Dataset Features

Two features describing properties of a RDF dataset are utilized in query evaluation: *Predicate Selectivity* and *Literal Selectivity*. Both *Predicate Selectivity* and *Literal
**Selectivity** are pre-computed statistics over the dataset, utilized to estimate the potential pruning power of the neighborhood structure for a given query template. We use some basic definitions here: For an RDF graph $G(V,E)$, any node $n_i \in V$ associated with URI reference label is denoted as **Resource Node** (blank nodes are treated as resource nodes). Other nodes associated with literal labels are denoted as **Literal Nodes**. Any edge $e_i \in E$ between two resource nodes is treated as **Relationship Edge**. Other edges (from resource node to literal node) are treated as **Attribute Edges**. (Literal nodes has no outgoing edges in the RDF graph).

**Predicate Selectivity**

**Predicate Selectivity** for a predicate $p$ in a RDF graph $G$ is defined as $s(p) = \frac{|p|}{|E|}$, where $|p|$ is the number of occurrences of predicate $p$ in $G$, and $|E|$ is the total number of edges in $G$. **Predicate Selectivity** is an important factor to determine the pruning power of a given predicate. Lower $s(p)$ indicates rare occurrence of a predicate which also indicates high pruning ability to reduce the matching candidates by checking predicate $p$.

**Literal Selectivity**

Different from the URI labels which are often long and hard to memorize, literal labels are more descriptive and typically used as important keywords to query RDF databases. The similarity of these literal labels of a given RDF dataset is another important factor influencing query performance. Given a length $n$ partial keyword associated with predicate $p_a$, the average number of literal labels matching this partial keyword in a given RDF dataset is a good measure to estimate literal similarity. However, enumerating all possible $n$-grams for literal labels of a large RDF graph is infeasible, and most of these $n$-grams are not meaningful and won’t be utilized as query keywords. Thus, we only use
all prefix $n$-grams ($n$-gram begins with the first character in a literal label) as candidates for partial keywords. Let $T_{n,p_a}$ be the set of all prefix $n$-grams of literals associated with attribute predicate $p_a$, and $m_{n,p_a}$ be the average number of literals associated with $p_a$ matching a prefix $n$-gram in $T_{n,p_a}$. Then, the **Literal Selectivity** of an attribute predicate $p_a$ is defined as $f_{n,p_a} = \frac{m_{n,p_a}}{|l(p_a)|}$, where $|l(p_a)|$ is the number of unique labels for predicate $p_a$. **Literal selectivity** can be pre-computed for all possible $n, p_a$ combinations and stored as statistic information for a RDF dataset. If the number of different predicates is very large, there is a tradeoff between the cost of computing and storing **Literal Selectivity** values for all $n, p_a$ combinations and the benefit of using more detailed statistics for query evaluation. As an alternative, a fixed $n$ value (e.g. average length of literals) can be used to estimate **literal selectivity** of an attribute predicate.

For large RDF datasets or RDF datasets with frequent minor updates, **Predicate Selectivity** and **Literal Selectivity** can be pre-computed by sampling multiple sets of triples from the RDF datasets and use the average Predicate Selectivity and Literal Selectivity from the sampled triples as a good estimation.

### 3.5.2 Query Template Complexity

The complexity of query templates mostly depends on two parts: 1. complexity of graph structure of the template; 2. generality of partial keywords. Two parameters, **Iteration Threshold** and **Join Threshold** are defined to identify query templates that are relatively complex with respect to the number of intermediate matching candidates, and joins in query execution. In order to estimate the potential pruning power of the neighborhood structure for the query template, **Neighborhood Selectivity** is defined.
Signature-based pruning tends to be effective if a query template is relatively complex and neighborhood structures associated with some query nodes have high selectivity.

**Iteration Threshold and Join Threshold**

These two parameters are defined as thresholds to determine whether a query is complex with respect to D-tree candidate generation and the number of joins needed for query evaluation. The number of candidates of the root node of a decomposed D-tree determines the number of iterations to generate its candidates (see Algorithm 2). Since higher number of iterations often indicates more D-tree candidates will be generated, it can be used as an estimation of the number of candidates for a decomposed D-tree. We also use the product of the sizes of candidate sets of all root nodes to estimate the number of intermediate tree joins. If the number of candidate generating iterations for any decomposed D-tree exceeds *Iteration Threshold* $\tau_1$ or the estimated number of intermediate tree joins exceeds *Join Threshold* $\tau_2$, the query template is considered as complex and has potential room for the neighborhood check to improve query performance.

**Neighborhood Selectivity**

**DEFINITION 4.1:** Given a query template $G_q$ for a RDF graph $G$, the *Neighbourhood Selectivity* $N_{q_i}$ of any query node $q_i \in G_q$ is defined as:

$$|\sum_{p_r \in \text{Neighbor}_k(q_i)} \ln(s(p_r)) + \sum_{p_a \in \text{Neighbor}_k(q_i)} \ln(s(p_a) \times f_{n,p_a})|$$

where $p_r$ is any relationship predicate, $p_a$ is any attribute predicate, $k$ is the number of hops of neighborhood check process, $n$ is the length of partial keywords associated with edge $p_a$, $s(p_r)$ and $s(p_a)$ are predicate selectivity for $p_r$ and $p_a$, and $f_{n,p_a}$ is literal selectivity.
The **Neighborhood Selectivity** of a given query node $q_i$ estimates the probability that all predicates appear in its K-Neighbors co-incidentally associated with a node in the original RDF graph $G$. It also takes partial keywords associated with attribute predicates into consideration. As this estimated probability often turns to be very small ($0 < N_{q_i} \ll 1$), we take the absolute value of its natural logarithm as the measurement to simplify its computation and comparison. Larger **Neighborhood Selectivity** indicates rareness of the neighborhood structure associated with $q_i$ appears in $G$; in other words, the neighborhood check is potentially beneficial for pruning the number of candidate matches if the neighborhood selectivity is high.

### 3.5.3 RDF-$\mathcal{h}$

RDF-$\mathcal{h}$ algorithm optimizes the query execution of a query template by selectively using signature-based pruning based on two conditions: (i) whether there is potential room for a query template to improve regarding the number of intermediate candidates and joins; (ii) whether the neighborhood structure of a query template has high selectivity to provide effective pruning. If both conditions are true, the signature-based pruning is utilized in processing the query template. The query plan begins with query preprocess and candidate generation as in Figure 3. After query decomposition, the number of candidate generating iterations for each D-tree and the estimated number of intermediate tree joins are checked with **Iteration Threshold** $\tau_1$ and **Join Threshold** $\tau_2$. If any number exceeds the threshold, then neighborhood check is considered. The next step is to compute the **Neighborhood Selectivity** for all query nodes to check if there is any query node with **Neighborhood Selectivity** satisfying the minimal **Selectivity Threshold** $\tau_3$. If so, neighborhood check is utilized.
Algorithm 4: Threshold Tuning

Input: Sampled Queries $L_Q$, IDMap Index $M_{id}$, 1 Hop NI Index $NI_1$, 3 Hop NI Index $NI_3$

Output: Thresholds $\tau_1$, $\tau_2$ and $\tau_3$

1. FOR each query $q_i$ in $L_Q$
   2. RunTime $T_1 = \text{MatchWithoutNICheck}(q_i, M_{id}, NI_1)$
   3. RunTime $T_2 = \text{MatchWithNICheck}(q_i, M_{id}, NI_3)$

   number of Iterations, number of Joins and NS of $q_i$ are recorded as $I_{q_i}$, $J_{q_i}$ and $S_{q_i}$

5. IF $T_1 > T_2$ then
   6. $R_{q_i} = \text{True}$
   ELSE
   8. $R_{q_i} = \text{False}$
   9. ADD $I_{q_i}$ to $I$, $J_{q_i}$ to $J$, $S_{q_i}$ to $S$ and $R_{q_i}$ to $R$

10. $L_T, L_F, \tau_1 \leftarrow \text{maxGain}(L_Q, I, R)$
11. $L'_T, L'_F, \tau_2 \leftarrow \text{maxGain}(L_F, J, R)$
12. $L_T = L_T.\text{addRange}(L'_T)$
13. $L''_T, L''_F, \tau_2 \leftarrow \text{maxGain}(L_T, S, R)$

FUNCTION $\text{maxGain}(L_Q, V, R)$

14. $mGain = -1$, $\tau = 0$, $L_T = \text{null}$, $L_F = \text{null}$
15. FOR each query $q_i$ in $L_Q$
\[ C_{TT} = 0, C_{TF} = 0, C_{FT} = 0, C_{FF} = 0, g = 0, L'_T = \text{null}, L'_F = \text{null} \]

FOR each query \( q_j \) in \( L_Q \)

IF \( V_{q_j} > V_{q_i} \) then

ADD \( q_j \) to \( L'_T \)

IF \( R_{q_j} \) then

\( C_{TT}++ \)

ELSE

\( C_{TF}++ \)

ELSE

ADD \( q_j \) to \( L'_F \)

IF \( R_{q_j} \) then

\( C_{FT}++ \)

ELSE

\( C_{FF}++ \)

\[ C_T = C_{TT} + C_{TF}, \quad p_T(L'_T) = \frac{C_{TT}}{C_T}, \quad p_F(L'_T) = \frac{C_{TF}}{C_T} \]

\[ C_F = C_{FT} + C_{FF}, \quad p_T(L'_F) = \frac{C_{FT}}{C_F}, \quad p_F(L'_F) = \frac{C_{FF}}{C_F} \]

\[ C = C_T + C_F, \quad w(L'_T) = \frac{C_T}{C}, \quad w(L'_F) = \frac{C_F}{C} \]

\[ g = w(L'_T) \times (p_T(L'_T) \times \log p_T(L'_T) + p_F(L'_T) \times \log p_F(L'_T)) + w(L'_F) \times (p_T(L'_F) \times \log p_T(L'_F) + p_F(L'_F) \times \log p_F(L'_F)) \]

IF \( g > m\text{Gain} \)

\[ m\text{Gain} = g, \quad L_T = L'_T, \quad L_F = L'_F, \quad \tau = V_{q_i} \]

Return \( L_T, L_F, \tau \)
3.5.3.1 Threshold Parameter Tuning

The choices of thresholds $\tau_1$, $\tau_2$ and $\tau_3$ significantly influence the query performance of the RDF-$\mathcal{H}$. The values of thresholds $\tau_1$, $\tau_2$ and $\tau_3$ can be set and tuned by using a sample set of queries on a given dataset. The tuning process is shown in Algorithm 4. It begins with executing a set of randomly sampled queries from user inputs, with and without the neighborhood check (line 2-3). During the query execution, the maximum number of candidate generating iterations $I_{q_i}$, the estimated number of intermediate joins $J_{q_i}$ and the maximum Neighborhood Selectivity (NS) $S_{q_i}$ for each query $q_i$ are recorded (line 4). Based on the query results, each sample query is labeled as true or false based on whether the neighborhood check improves the query performance or not (line 5-9). The tuning process is then formulated as a decision tree problem to set thresholds $\tau_1$, $\tau_2$ and $\tau_3$. According to the RDF-$\mathcal{H}$ algorithm, a sample query is expected to benefit from neighborhood check (i.e. labeled as true) if $(I_i > \tau_1 \lor J_i > \tau_2) \land (S_i > \tau_3)$. The candidate values of thresholds $\tau_1$, $\tau_2$ and $\tau_3$ are selected from $I$, $J$ and $S$ where $I$ is the list containing the maximum number of candidate generating iterations, $J$ is the list containing the estimated number of intermediate joins and $S$ is the list containing the maximum Neighborhood Selectivity for all sampled queries. The decision tree utilized in the threshold tuning process (Algorithm 4) is shown in Figure 4. It is designed to first check if $I_i > \tau_1$ (queries are separated into two groups as $L_T: I_i > \tau_1$ and $L_F: I_i \leq \tau_1$), then to check if $J_i > \tau_2$ for queries in $L_F$ (queries in $L_F$ are separated into two groups as $L'_T: J_i > \tau_2$ and $L'_F: J_i \leq \tau_2$). It finally checks if $S_i > \tau_3$. 

57
for queries in both $L_T$ and $L'_T$. Thus, thresholds $\tau_1$, $\tau_2$ and $\tau_3$ are set to achieve the maximum information gain (defined with FUNCTION maxGain based on entropy) for each check in the decision tree (line 10-13). FUNCTION maxGain enumerates all possible choices of threshold $\tau$ according to the parameter list $V$ where $V$ can be $I$, $J$ and $S$. For each candidate $\tau$ value, the list of queries are split into two groups $L'_T$ and $L'_F$, by evaluating whether the parameter $V_{q_j}$ associated with query $q_j$ exceeds the candidate threshold (line 18-19 and 24-25). For each group of queries ($L'_T$ or $L'_F$), the number of queries with label true are counted as $C_{TT}$ for $L'_T$, and $C_{FT}$ for $L'_F$. Similarly, the number of queries with label false are also counted as $C_{TF}$ for $L'_T$, and $C_{FF}$ for $L'_F$ (line 20-23 and 26-29). Since a query is labeled either true or false (based on whether the signature-based pruning improves query performance or not), the computation of a two class entropy is utilized as:

$$Entropy(L) = -(p_T \times \log_2 p_T + p_F \times \log_2 p_F)$$

where $p_T$ represents the probability of queries with label true and $p_F$ represents the probability of queries with label false. Then, the entropy of the two groups $L'_T$ and $L'_F$ are computed in a weighted

\[
Figure 4. Threshold Tuning Decision Tree
\]
average formula where the weight of each group \( w(L'_\tau) \) and \( w(L'_F) \) is proportional to the number of queries in it. Comparing the information gain by using different candidate \( \tau \) values is the same as comparing the negative values of the entropy of the two resulting groups since the entropy of \( L_Q \) (the given list of sample queries) is the same for all candidate \( \tau \) values (line 30-33). Finally, the value of \( \tau \) with the maximum information gain is returned (line 34-36).

The overall time complexity of Algorithm 4 is \( O(|L_Q|^2) \) where \( |L_Q| \) is the number of sampled queries. Based on the RDF-\( \mathcal{H} \) algorithm, value pairs \( I_i \) and \( J_i \) are compared with the thresholds \( \tau_1 \) and \( \tau_2 \) to determine whether a query is expected to be complex to evaluate. However, utilizing value pairs in the decision tree increase the time complexity of Algorithm 4 due to considering the candidate pairs of thresholds \( \tau_1 \) and \( \tau_2 \). Thus, the decision tree is designed to check one threshold at a time. Note that, the check if \( I_i > \tau_1 \) and the check if \( J_i > \tau_2 \) in the decision tree (Figure 4) can be swapped without violating the heuristic in the RDF-\( \mathcal{H} \) algorithm which provides a flexible way to reduce the possible over-fitting for setting the thresholds \( \tau_1 \) and \( \tau_2 \). Since the decision tree is solved by using information gain based on entropy, it tries to minimize the impurity of the labels of queries in each resulting group regardless the expectation whether a query should benefit from signature-based pruning indicated by the heuristic in the RDF-\( \mathcal{H} \) algorithm. The experimental results confirm that using the thresholds \( \tau_1 \), \( \tau_2 \) and \( \tau_3 \) to determine whether a query can benefit from signature-based pruning in RDF-\( \mathcal{H} \) algorithm provides a good heuristic.
3.6 Experimental Analysis

The project is implemented with Visual C# 2010 and SQL Server 2008. All experiments were performed on a 2.93GHZ Intel(R) Xeon machine with 48GB ram running Windows Server 2008 R2. STWIG [SW+12] is revised as STWIG+ to support partial keyword and directed graph by using IDMap index and 1 hop NI index to check partial keywords, and generating STWIG matching candidates. An improved version of SPath [ZH10], named as SPath(NI2), which utilizes 2 hop NI index instead of its original neighborhood index to optimize the neighborhood check for partial keywords and supports directed graph is implemented. An improved version of GraphQL which utilize NI indexes can also be utilized here. However, it performs similarly as SPath(NI2) since the same NI index is utilized and we don’t show it in experiments. Different neighborhood checks are employed by RDF-$\mathcal{A}$, namely, $\mathcal{A}$-2Hops (2 hop neighborhood check), h-3Hops (3 hop neighborhood check) and h-VC (neighborhood check based on vertex cover NI index).

3.6.1 Experimental Set Up

We use two RDF datasets from synthetic benchmarks LUBM [GP+05] (Lehigh University Benchmark) and SP2B [SH+09] (SPARQL Performance Benchmark). We also use two RDF datasets from real applications IMDB (Internet Movie Database, same dataset utilized in [KM+09]) and DBLP [Ley09] (Computer Science Bibliography Database). Five different versions of the DBLP dataset with sizes varying from approximately 1 million triples to 5 million triples are used to perform scalability test. To scale the DBLP dataset in a uniform way, newer years’ publications are added until the dataset contains approximately 1 million more triples each time. We denote these five DBLP datasets as
Queries are generated by first randomly selecting a subgraph from the original RDF graph, and then generalizing labels of the nodes to get partial keywords. Subgraph selection begins with a small subgraph (a node, an edge or a path) and recursively selects random edges adjacent to the already selected subgraph until query template size requirement is reached. Two additional rules are employed for this process: (i) at most 3 adjacent edges of each node that is already chosen are added to the candidate edge set for a random selection; (ii) candidate edges which can increase the longest path length in the subgraph have higher probability to be chosen. Rule (i) aims to avoid the situation that, once a node with a large degree is chosen, all the following selections fall into its neighbors so that query templates can stretch in all directions. Rule (ii) increases the probability of having query nodes which contain 2 or more hop neighborhoods. Once a subgraph is selected, the partial keywords are generalized from the RDF labels and assigned to each query node. For URI labels of resource nodes, partial keywords are generated by removing the long IDs. For literal labels, partial keywords are generated by recursively removing the last character until it can match 10 to 500 labels in the RDF graph (if more than 1 partial keyword can be generated, we randomly choose one of them). During the experiments, few queries take significantly more time to evaluate for the STWIG+ algorithm as compared with other algorithms for SP2B, DBLP and IMDB datasets. These queries often result in unacceptable response time (more than tens of seconds) and degrade the average query time of the STWIG+ algorithm dramatically (2 to 3 times slower). As our purpose is to investigate the performance improvement of the RDF-$\mathcal{H}$ algorithm in general, we treated these queries as
outliers (2 queries in SP2B, 1 query in DBLP and 1 query in IMDB from a set of 40 random queries) and excluded them from final results.

As introduced in Section 4.3, the iteration threshold $\tau_1$, the join threshold $\tau_2$ and the neighborhood selectivity threshold $\tau_3$ can be tuned by using a set of query samples from user inputs. As we use random query generation to perform the experiments, we have used a set of 20 random query templates with size 6 (6 edges) as our sample set of queries to set up the parameters for each dataset.

### 3.6.2 Space Comparison for Different Indexes

The average space required for NI index is $O\left(\frac{N(\mu^{d_{\text{max}}})}{m}\right)$ where $d_{\text{max}}$ is the maximum indexed distance, $m$ is the binning factor, $N$ is the number of nodes in the RDF graph and $\mu$ is the average node degree. Thus, once parameters $d_{\text{max}}$ and $m$ for the NI index are fixed, the RDF dataset directly determines the index space. Figure 5 shows...
information on different indexes for different RDF datasets in percentage of the original dataset (IMDB: 63.31 MB, DBLP-1: 207.961 MB, LUBM: 249.695 MB and SP2B: 182.891 MB). For all different NI indexes, the binning factor $m$ is set as 5. The IDMap table is relatively small compared with NI indexes. Four different NI indexes are compared here, and are utilized by different querying techniques (1 hop index for “STWIG+”, 2 hop index for “SPath(NI2)” and “$h$-2Hops”, 3 hop index for “$h$-3Hops” and vertex cover index for “$h$-VC”). Clearly, the space needed for NI indexes increases dramatically as $d_{max}$ increases. The vertex cover index uses 2 hop neighbors for nodes in the vertex cover set, and 1 hop neighbors for all other nodes. Thus, the space requirement for the vertex cover index is between 1 hop index and 2 hop index. The average node degree of the RDF graph also has great impact on index sizes. SP2B and DBLP each has an average node degree of approximately 3, and it is approximately 8 for LUBM and IMDB. Thus, the space increase of NI index by indexing more hops of neighbors is much sharper for LUBM and IMDB compared with SP2B and DBLP. As one can also observe from SP2B and DBLP, 2 hop and 3 hop neighborhood indexes are almost equivalent in terms of space percentage, which indicates that 3 hop neighbors are very rare for these datasets.

### 3.6.3 Query Performance for Different Datasets

We randomly generate 40 queries with size 6 for each dataset separately as each dataset has its own graph structure and RDF labels. Query performance results of all different query techniques for different datasets are shown in Figure 6. Clearly, there is a big difference between LUBM and the other three datasets. For LUBM, the neighborhood check is not beneficial for most queries due to its little pruning power resulting from the coherent graph structure and the uniform literal labels. Predicates of a given type of
resources are uniformly duplicated in LUBM which leads to an extremely high coherence as 0.95. In addition, the literal uniformity is high with a dictionary size under 20 for 6-gram prefixes (e.g., names of classes are “class1”, “class2” …). Thus, most random subgraph queries do not benefit from the neighborhood check for the LUBM dataset. As the pruning power of the neighborhood check here is negligible, the more hops of neighbors are checked, the more performance degrades due to its time overhead. The other three RDF datasets, SP2B, DBLP and IMDB have performance patterns similar to each other. As expected, the neighborhood check helps to prune unnecessary intermediate candidates and joins effectively, and, as a result, more hops of neighbors are checked, resulting in better performance.

Figure 6. Query Comparison (Datasets)
Since SPath(NI2) always uses the additional neighborhood check, query performance suffers because of it for LUBM, but benefits from it as effective pruning for other datasets. On the other hand, as STWIG+ never uses the additional neighborhood check, it performs well for LUBM, but suffers from the unnecessary intermediate candidates generated for datasets SP2B, DBLP and IMDB. RDF- $\mathcal{H}$ algorithm outperforms SPath(NI2) and STWIG+ since it combines their advantages. Simple queries are processed directly by D-tree candidate generation and joins while complex queries are accelerated by utilizing the additional neighborhood checks. Checking for 2 hop and 3 hop neighbors are more effective in IMDB compared with SP2B and DBLP since 2 hop paths and 3 hop paths are rare in them which leads to smaller number of query nodes generated with 2 hop or 3 hop neighbors. Compared with 2 hop and 3 hop neighborhood checks, the vertex cover neighborhood check works well since the space required is smaller. Performance of the vertex cover neighborhood check is slightly worse than the complete 2 hop neighborhood check, as expected.

3.6.4 Scalability Test

3.6.4.1 Dataset Scalability Test

The purpose of dataset scalability test is to simulate the scalability challenge for real applications. Each year, more and more new entities are added to the dataset which leads to larger and larger dataset sizes. For a similar set of user queries, the query processing time becomes longer. This is a major concern for real applications and greatly influences the choices of query algorithms. Here, we use DBLP datasets with increasing sizes (section 6.1) to perform the dataset scalability test with the same set of 40 random queries with size
6. As the datasets contain more triples, partial keywords for literal labels should be re-generated in order to control the number of RDF labels matched. Results of scalability experiments are shown in Figure 7. Clearly, STWIG+ suffers the most when the size of the dataset increases since the number of unnecessary candidates and intermediate joins increase dramatically. With neighborhood check processes, query time increases slightly with the size of the dataset. For randomly generated queries with size 6, 3 hop neighborhood check is sufficient to capture most of the graph structure, and scales best in terms of query performance. \( \mathcal{H} \)-2Hops algorithm has a similar performance pattern with SPath(NI2) since they both rely on the 2 hop neighborhood check. Due to enforcing all queries to utilize the neighborhood check, and not all queries benefit from it, SPath(NI2) has overhead, and ends up performing worse than the \( \mathcal{H} \)-2Hops algorithm. The \( \mathcal{H} \)-VC
algorithm which uses combined 1 hop and 2 hop neighborhood indexes works well when
the RDF dataset is small. But query time increases sharper than SPath(NI2) and \( h \)-2Hops
algorithms. Thus, the neighborhood check is powerful in pruning unnecessary candidates
and reducing intermediate joins which scales well with dataset size increases.

Here, our dataset scalability test produces different conclusions for the STWIG+
algorithm of [SW+12]. Authors of [SW+12] conclude that the graph size has no significant
impact on the response time of the STWIG algorithm when the average node degree is
fixed. This difference is due to two factors: (i) the synthetic data generated in [SW+12] has
a completely different structure than the DBLP dataset; (ii) STWIG algorithm in [SW+12]
is a distributed algorithm implemented in a computing cluster rather than a single machine
environment.

3.6.4.2 Template Scalability Test

With the two rules defined in the random query generation, the more edges in a query
template, the higher probability that it contains a complex graph structure (e.g., longer
paths). In this section, we conduct a template scalability test on all four RDF datasets using
random queries with size 4, 6 and 8 (larger query template size is rare in real application).
Figure 8 shows query performance results for different query techniques for all four RDF
datasets. For SP2B, IMDB and DBLP, the neighborhood check benefits more when the
template size of random queries increases due to the more complex graph structure and
having more query nodes with more hop neighbors, as expected. \( h \)-3Hops algorithm
performs best and \( h \)-2Hops algorithm works slightly better compared with the \( h \)-VC
algorithm when the size of the query template increases. For the two existing query
Figure 8. (a) Template Scalability Test

Figure 8. (b) Template Scalability Test
Figure 8. (c) Template Scalability Test

Figure 8. (d) Template Scalability Test
techniques, STWIG+ outperforms Spath+ when the template size is relatively small for IMDB, DBLP, but degrades sharply when template size increases.

For LUBM, the neighborhood check is not effective for most of the queries with different sizes. Thus, as a result, neighborhood check processes have extra time overhead and lead to worse performance. Spath+ suffers most as it enforces all queries to perform the neighborhood check. Hybrid algorithms are comparable to STWIG+ when the template size is small, but degrade as the template size increases. This is due to threshold parameters set up based on a set of sampled queries with size 6. When the sizes of queries become larger, more queries are qualified as complex queries and hybrid algorithms choose neighborhood checks to improve query performance which is not effective for LUBM.

3.6.5 Connection Edge Queries

The connectivity check performance is tested using DBLP-1, and query templates with a connection edge between two components. We generate queries with connection edges as follows: 1. generate all length 5 paths in DBLP-1; 2. Randomly pick one 5 hop path and use its source node and destination node as two seed nodes to generate two random queries with size 3 each; 3. Take these two queries as two components, mark the connection edge between the two seed nodes, and set the Maximum Path Distance $d_c$ for connection edge as 5. The results in shown in Table 1 (The three corresponding algorithms are $\mathcal{H}$-3Hops, $\mathcal{H}$-2Hops and STWIG+. The average number of connectivity checks is 277.88). For the 3 hop NI index, the overhead of the connectivity check is quite trivial compared to the time to match components, since the connectivity test between $n_i$ and $n_j$ can be directly performed by retrieving 3-hop neighbors from the index (Algorithm 3). However, for 2 hop and 1 hop indexes, the overhead of the connectivity check increases sharply due
to recursive retrieval of all border nodes’ neighborhood indexes, and the time needed to combine them together. The vertex cover neighborhood index can answer connection edges with distance 5 in the same way as 2 hops; results for it are not shown here. Clearly, if only 1 hop index is used to perform connectivity checks, it will become a bottleneck for the query performance, resulting in a poor performance.

**Table 1. Query Performance of Query Templates with Connectivity Edge**

<table>
<thead>
<tr>
<th>NI Indexes</th>
<th>Query Time (Component Matching)</th>
<th>Query Time (Connectivity Check)</th>
<th>Time Percentage (Connectivity Check)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Hops</td>
<td>70.89 ms</td>
<td>2.65 ms</td>
<td>3.6%</td>
</tr>
<tr>
<td>2 Hops</td>
<td>70.47 ms</td>
<td>49.31 ms</td>
<td>41.17%</td>
</tr>
<tr>
<td>1 Hop</td>
<td>70.76 ms</td>
<td>866.92 ms</td>
<td>92.45%</td>
</tr>
</tbody>
</table>

**3.6.6 Choice of Neighborhood Checks for Hybrid Algorithm**

Choice of neighborhood checks for RDF-\(\mathcal{H}\) algorithm is important for different datasets. For most RDF datasets which can benefit from the neighborhood check, there is a tradeoff between query performance and space efficiency as shown in experiments. The frequency of complex user queries and the frequency of connection edges with large distance constraints have a big impact on the query performance of different NI indexes. The RDF-\(\mathcal{H}\) algorithm is more effective when user input queries have different levels of complexity. The query evaluation takes the characteristics of the dataset and query templates into consideration and chooses an optimized query plan. Compared with SPath(NI2) and STWIG+, the RDF -\(\mathcal{H}\) algorithm can provide an overall 20% to 30% higher performance improvements in most cases. The dataset evaluation metrics also give us hints whether a dataset can benefit from the neighborhood check. For a dataset with a high coherence, low specialty and high label uniformity, such as LUBM, one should
consider \( \mathcal{A} \)-VC first due to little expected benefits from the neighborhood check and best space efficiency.

### 3.7 Conclusions and Comments

In this project, we evaluate the usability of signature-based pruning on querying graph-structured RDF data using graph templates. Due to the observation that signature-based pruning is not always beneficial, we propose a hybrid algorithm RDF-\( \mathcal{A} \), which selectively uses neighborhood check based on the characteristics of RDF datasets and query templates. By tuning the parameters, the RDF-\( \mathcal{A} \) algorithm can automatically capture frequent user query patterns and be adjusted to maximize the benefits of signature-based pruning which provide an overall 30% query performance improvement for random generated subgraph queries. Based on the RDF dataset characteristics analysis, we can also identify datasets with respect to the expected level of performance gains from signature-based pruning.

One important problem remained is to understand the characteristics of RDF datasets. Performance gains of signature-based pruning can differ from dataset to dataset significantly. This is also observed in Figure 6, that signature-based pruning is not effective for all datasets. In the next chapter, we will introduce some dataset evaluation metrics to provide insights on identifying the expected level of performance gains from signature-based pruning for different RDF datasets.
Chapter 4 Characterizing RDF Datasets

In this chapter, we focus on evaluating different real and benchmark RDF datasets to understand their difference. Performance evaluations of RDF management systems are typically done using synthetic RDF benchmark data. However, achieving the same levels of performance on real datasets depends on how closely the synthetic RDF benchmark data mimics real datasets. As we observe from chapter 3, some RDF benchmark datasets yield significantly different results compared with real datasets.

An important previous paper, titled as “Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets” by S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea [DK+11] has addressed this problem. Duan et al [DK+11] begin with evaluating RDF datasets, both real and synthetic, by primitive metrics (e.g. in-degree, out-degree, the number of Subjects, Objects, Predicates, etc.), and conclude that these primitive metrics are not adequate to identify the differences between benchmarks and real datasets. The coverage and coherence metrics are then introduced as an intuitive way to combine the primitive metrics into one single measure of RDF datasets named as “structuredness”. Experiments by Duan et al show that existing benchmarks are limited and biased in terms of dataset structuredness, as compared with real datasets which cover a wide spectrum of structuredness. Hence, they conclude that “existing benchmark data have little in common with real data” [DK+11] and an improved benchmark generator is necessary.

Although the coverage and coherence metrics do provide evidence to differentiate
benchmark data from real data, the conclusion is questionable since there is an overlap in coherence values for benchmark data and real data. This situation becomes more salient when UOBM [ZC+13] is included in our experiments. Thus, some RDF benchmarks should be considered as good testing bases by checking the coverage and coherence metrics alone. As an example, the SP2B dataset is developed by simulating the structure of the VLDB dataset, and it ends up having similar coherence values. Based on the reevaluation of the coverage and coherence metrics, we identify several limitations of them which motivates us to design new benchmark metrics to understand the characteristics of RDF datasets.

4.1 Coverage and Coherence Metrics

4.1.1 Metrics Reexamination

The coverage metric is utilized [DK+11] to measure how uniformly predicates are distributed among the same type/class (the terms, type and class, are used interchangeably in this paper) of resources in a RDF dataset. The coherence metric is computed from the coverage metric which combines coverage values of all different types in an RDF dataset via a weighted sum formula. The RDF dataset type system utilized by the coverage and coherence metrics directly comes from the predicate “rdf:type” which defines the instances (which are resources) of different RDF classes. The resources associated with multi-classes are analyzed for each class separately in the coverage and coherence computation. To avoid potential bias, the weighted formula takes into consideration both the number of resources in each class and the number of distinct predicates associated with the resources of each class.

DEFINITION 2.1: Given an RDF dataset and its type system $T$, the coverage $\text{Cov}(t)$ for any type $t \in T$ is defined as
where \( r(t) \) is the set of resources of type \( t \), \( P(t) \) is the set of predicates associated with any resource in \( r(t) \), \( c(r(t),p) \) is the number of resources in \( r(t) \) associated with predicate \( p \), and \( |P(t)|, |r(t)| \) are the sizes of these two sets.

**DEFINITION 2.2:** Given an RDF dataset \( \mathcal{D} \) and its type system \( T \), the coherence \( \mathcal{C}_\mathcal{D} \) of the dataset \( \mathcal{D} \) is then defined as

\[
\mathcal{C}_\mathcal{D} = \sum_{\forall t \in T} \omega(t) \times \mathcal{C}_t
\]

where \( \omega(t) = \frac{|r(t)| + |P(t)|}{\sum_{t \in T}(|r(t)| + |P(t)|)} \).

### 4.1.2 Evaluation Results

To extend the experiments in [DK+11], we reexamine the coverage and coherence metrics for a large set of real RDF datasets and RDF benchmarks.

#### 4.1.2.1 Real Datasets

Below we briefly describe eight real RDF datasets which are frequently utilized in RDF research papers. The first four datasets are also used in [DK+11]; but we use the newest versions.

**DBpedia [BL+09]:** DBpedia extracts structured information from Wikipedia in the form of RDF. The dataset we use contains more than 250 million RDF triples (150 million triples in [DK+11]), and comes with a broader type system, named as DBpedia-B in [DK+11].

**YAGO [SK+08, Yago]:** The YAGO dataset is a large knowledge base which brings knowledge from both Wikipedia and Wordnet. The newest version of YAGO has more
than 240 million RDF triples (19 million triples in [DK+11]).

**Barton** [Barton]: The Barton library dataset recorded the Machine Readable Catalog (MARC) data of the MIT Libraries Barton catalog and converted it into RDF. The version we use contains about 77 million RDF triples (45 million triples in [DK+11]).

**WordNet** [Wordnet]: WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. This dataset contains 1.9 million of RDF triples (same in [DK+11]).

**LinkedMDB** [LMDB]: LinkedMDB contains hundreds of thousands of high-quality interlinks to several movie-related data sources in the Linking Open Data project. The newest version of LinkedMDB has more than 60 million of RDF triples.

**DBLP** [Ley09]: DBLP provides bibliographic information on major computer science journals and proceedings. It contains the metadata of over 2 million publications, over 1 million authors in several thousands of publication venues. The version we use has over 100 million RDF triples.

**NCBI Taxonomy** [NA+08, NCBI]: This RDF dataset is obtained from the Bio2RDF project. The NCBI taxonomy database contains the names of all organisms represented in genetic databases, and with at least one nucleotide or protein sequence. Our version contains approximately 180 million RDF triples.

**SGD (Saccharomyces Genome Database)** [NA+08, SGD]: This dataset is also from the Bio2RDF project. It has information about the molecular biology and genetics of the yeast Saccharomyces cerevisiae. The dataset contains about 6 million RDF triples.
4.1.2.2 Benchmark Datasets

We use four benchmark datasets which provide their benchmark data generators. The first three are also utilized in [DK+11] and the last one (UOBM) is an extension of the third benchmark dataset (LUBM). We do not utilize the TPC benchmark which is utilized in [DK+11] due to the fact that it originally generates a relational database which cannot be judged as an RDF benchmark. To be consistent with [DK+11], the first three benchmark datasets are generated same as the datasets utilized in [DK+11].

_BSBM_ [BA09]: Berlin SPARQL Benchmark (BSBM) considers an e-commerce domain with types _Product, Offer, Vendor, Person_ and _Review_. The relationships between products and vendors offering them and between users and product reviews these users write are simulated (25 million triples are generated as in [DK+11]).

_LUBM_ [GP+05]: Lehigh University Benchmark (LUBM) considers a University domain. The types (e.g., _Student, Course_, and _Professor_) and common relationship among them (e.g., _takeCourse, teachOf, worksFor_) are simulated (100 million triples are generated as in [DK+11]).

_SP2Bench_ [SH+09]: SP2Bench benchmark [SH+09] uses the same domain as DBLP. It contains types as _Person, Article, Proceeding, Journal, and Book_, etc. (10 million triples are generated as in [DK+11]).

_UOBM_ [ZC+13]: University Ontology Benchmark (UOBM) extends LUBM with a more complex ontology, having disjunctive axioms and negation, more relationships among persons, and more properties for different types of resources. We generate a dataset with 20 million triples.
4.1.2.3 Results

Coherence values of these twelve datasets are shown in Figure 9, where the first eight are real datasets and the remaining four are benchmark datasets. One can observe that the coherence values of benchmark datasets are limited to a smaller range as compared to those of real datasets. Compared with other benchmark datasets, UOBM (not utilized in [DK+11]) ends up having the lowest dataset coherence value. Including UOBM into the benchmark datasets increases the overlap between the range of coherence values of real datasets and that of benchmark datasets. As claimed in [DK+11], the dataset coherence is a good indicator of the dataset structuredness which differentiates real data from benchmark data. However, the conclusion “existing benchmark data have little in common with real data”
based on checking only the dataset coherence is questionable since several real datasets have similar coherence values as benchmark datasets, and the UOBM dataset tends to be a good counter-example. This motivates us to explore additional dataset evaluation metrics to provide better insight for the task of differentiating benchmark datasets from real datasets.

4.1.3 Limitation of Coverage and Coherence Metrics

The coverage and coherence metrics is indeed an indicator of the limitations of existing RDF benchmarks, but they have several shortcomings.

First, the coverage and coherence metrics are defined based on the type system of an RDF dataset, and the type system itself is not required to be complete for different RDF datasets. In other words, the types of some resources are unknown, e.g., DBLP contains a large number of publications with the unknown type (not classified as Proceedings, Journals, Book, etc.). This limits the usability of the coverage and coherence metrics when a poor type system is defined in an RDF dataset with a large number of resources without type definitions. Thus, the coherence metric may be a biased measure for some cases. Based on our observations, the twelve datasets which are commonly utilized in database research all have relatively complete type systems.

Second, the coverage and coherence metrics are vulnerable to minor updates of a small number of RDF triples. This is observed from our experiments where for some real datasets, several predicates are only defined for resources in an early time period and never utilized later which leads to a sharp drop of dataset coherence. It can be easily understood from Definitions 2.1 and 2.2: assuming that $T$ contains only one type, $|t(\mathcal{E})|$ is large and
\[ c(\tau(\tau), p_a) = |\tau(\tau)| \] for all \( p_a \in P_a(\tau) \), adding \( k \) triples with different predicate labels changes the dataset coherence from 1 to approximate \( \frac{|P_a(\tau)|}{k+|P_a(\tau)|} \). If \( |P_a(\tau)| \) is small, the coherence value of the dataset will change significantly by a small \( k \) value. Also, there are always type errors in different datasets. If some predicates are typed erroneously, it will cause the dataset coherence to drop in the same manner. As the coverage and coherence metrics are proposed to measure the structuredness of a RDF dataset, a small number of triples should not be the dominating factor to decide their values. Thus, some thresholds should be utilized in coherence computation to prune predicates with extremely small number of occurrences.

Third, the coverage and coherence metrics only take into account whether a predicate is associated with a resource or not, but the number of occurrences of a specific predicate associated with a resource is not considered. The datasets with a uniform neighbor structure (e.g., each person has only one friend in a social network) and the datasets with diverse neighbor structures (e.g., the number of friends associated with each person has a large range) cannot be distinguished from each other by checking the coverage and coherence metrics. As the purpose of the coherence value is to measure the structuredness of an RDF dataset, this is a big limitation since a RDF dataset with diverse relation patterns can’t be distinguished from a RDF dataset with uniform structure.

The observation that “a large number of real datasets overlap with benchmark data” by checking the coverage and coherence metrics, and limitations of the coverage and coherence metrics necessitate additional metrics to compare RDF benchmarks from real RDF datasets.
4.2 New Evaluation Metrics

DEFINITION 3.1: An **RDF Graph** is a directed graph \( G = \{V, E, l, f\} \) where \( V \) is a set of vertices representing either Subjects, Objects or both, in RDF triples. \( E \subseteq V \times V \) is a set of directed edges representing predicates pointing from Subjects to Objects. \( l \) is a label set for subjects, objects and predicates. \( f: V/E \rightarrow l \) denotes the mapping function between vertices/edges to labels. Example RDF graph from DBLP is shown in Figure 10.

DEFINITION 3.2: For an RDF graph \( G \), any node \( n_t \in V \) associated with URI reference label is denoted as **Resource Node**\(^1\). All other nodes associated with literal labels are denoted as **Literal Nodes**.

\(^1\) Blank nodes associated with local identifiers are treated same as resource nodes in this paper.
DEFINITION 3.3: For an RDF graph $G$, any edge $e_i \in E$ between two resource nodes is treated as **Relationship Edge**. All other edges (from resource node to literal node) are treated as **Attribute Edges** (Literal nodes have no outgoing edges in the RDF graph).

In Figure 10, resource nodes and relationship edges are in blue while literal nodes and attribute edges are in red. There are two reasons we want to distinguish the analysis of relationship edges and attribute edges in dataset evaluation: (1) Subjects in triples can only be resource nodes and the graph structure of attribute edges is simple (no cycles, no paths); (2) As the dataset size increases over time, the number of attribute edges associated with resources of the same class/type tend to remain the same while the number of relationship edges associated with resources increases if the average degree of RDF graph increases. Thus, attribute edges form a graph structure which is relatively static and simple over time while the graph structure of relationship edges is highly dynamic, and relatively more complex, over time.

4.2.1 Relationship Specialty [QO-2]

As explained in Section 3, the number of occurrences of the same predicate associated with each resource provides useful insights on the graph structure of an RDF dataset, especially for relationship predicates. In a real dataset, it identifies the diverse relationship patterns for different resources, and makes some resources distinguishable from others. Taking the "#has-author" relationship in VLDB as an example, the number of papers published is an important factor to identify “special” persons who publish significantly more papers than others. This kind of relationship specialty is common for real datasets in all different areas, (e.g., a movie star is liked by millions of fans in social networks, a
common molecule is involved in thousands of reactions in biological pathways, a concept is linked by millions of concepts in Wiki, and a book is referenced by thousands of other publications in library information systems). At the same time, synthetic data is limited in showing the relationship specialty because either uniform relationship patterns are simulated for all resources, or a random relationship generation process is utilized. The motivation behind the *relationship specialty metric* is to capture this fundamental difference between real data and benchmark data.

**Table 2. Relationship specialty statistics for all datasets**

(\(|R|\): the number of relationship predicates in the dataset; Median, Mean, Max: the median, mean, max of *relationship specialty* values of all relationship predicates in a dataset; 0 indicates a uniform distribution)

| Datasets  | \(|R|\) | Median | Mean     | Max          |
|-----------|--------|--------|----------|--------------|
| DBpedia   | 39     | 123.1  | 12,257.9 | 200,430.6    |
| Yago      | 37     | 139.6  | 961.7    | 13,778.6     |
| Barton    | 36     | 3955.8 | 150737.2 | 1,061,130    |
| Wordnet   | 12     | 71.8   | 8,572.6  | 101,570.4    |
| Taxonomy  | 5      | 0      | 20,067.4 | 99,746.4     |
| SGD       | 30     | 9.5    | 10,376.1 | 101,127.3    |
| Linkedmdb | 23     | 42.6   | 105.9    | 1,275.1      |
| DBLP      | 7      | 782.5  | 15,476.9 | 101,006.3    |
| SP2B      | 5      | 0      | 1.17     | 3.17         |
| BSBM      | 10     | 0      | 0.31     | 3.07         |
| LUBM      | 12     | 0      | 0.70     | 4.35         |
| UOBM      | 23     | 0      | 1.72     | 9.10         |

We clarify this problem as identifying the existence of entities which indicate distinguishable relationship patterns for a relationship \(\tau_i\). For this, the distribution \(d_i\),
which indicates the number of occurrences of $\tau_i$ associated with each resource, is analyzed. The next question is which measure can be utilized to measure the relationship specialty of distribution $d_i$.

There are two factors we consider: 1) the population of the distribution $d_i$ is typically very large (billions of persons involved in analysing a relationship in social networks); 2) most resources share similar relationship patterns for a relationship $\tau_i$ (e.g., most authors publish less than 10 papers in VLDB). These two factors make variance in $d_i$ not sufficient in capturing the “special” resources as the variance changes too little due to the influence of small numbers of “special” resources on an extremely large population. Here, the Pearson's kurtosis value [Jack13] of $d_i$ is utilized to understand relationship specialty.

Figure 11. Relationship specialty for all datasets
Compared with variance, the kurtosis value of a distribution measures the fourth moment about the mean, magnifying the effect of small number of “special” resources. A higher Pearson's kurtosis value indicates that the variance of $d_i$ is mostly related to infrequent extreme deviations, as opposed to frequent modestly sized deviations.

The *relationship specialty* of a relationship predicate $r_i$ is defined as the kurtosis value of the distribution $d_i$ which records the number of occurrences of $r_i$ associated with each resource. There are many different relationship predicates in a dataset, and they should be all analysed. We show the *relationship specialty* values of “major” predicates (which appear in more than 1,000 triples) for all different datasets in Table 1. Differences between real datasets and benchmark datasets for relationship specialty values are significant. Most relationship predicates in benchmark datasets are either uniformly distributed or normally distributed among all resources. In contrast, all real datasets contain relationship predicates with kurtosis values over thousands which indicate the existence of special relationship patterns. In order to compare different RDF datasets, we define *relationship specialty* of a dataset in the form of a weighted sum of *relationship specialty* values of all the relationship predicates in the dataset where each relationship predicate contributes in proportional to the number of triples associated with this predicate:

$$\text{RS}(D) = \sum_{r_i \in R} \left( \frac{|r_i|}{\sum_{r_j \in R} |r_j|} \times \mathcal{K}_{r_i} \right),$$

where $|r_i|$ is the number of triples containing relation predicate $r_i$, $\mathcal{K}_{r_i}$ is the Pearson's kurtosis of relationship $r_i$ and $R$ is the set of all relationship predicates in $D$.

The results of the *relationship specialty* for all datasets are shown in Figure 11. As observed, the dataset *relationship specialty* of all benchmarks are below ten while the dataset *relationship specialty* of all real datasets are more than hundred. Compared with
the coherence metric, the relationship specialty metric provides a more effective way to differentiate real data from benchmark data. This is due to the fact that resources of the same type are treated similarly in benchmark generation, while roles of “importance” associated with resources are generally different in real datasets.

| Table 3. Statistics of dictionary sizes with different sampling sizes for all datasets |
|-----------------------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Datasets          |                  |                  |                  |                  |                  |                  |                  |                  |                  |                  |
|                  | Mean             | Standard Deviation | Coefficient of Variation |
|                  | Sample Size     | Sample Size      | Sample Size      |
|                  | 1K               | 10K              | 100K             | 1K               | 10K              | 100K             | 1K               | 10K              | 100K             |
| DBpedia         | 2191.4           | 1354.1           | 66039.4          | 560.6            | 687              | 704.6            | 221              | 1061.972         | 1.61%           |
| Yago            | 684.9            | 4732.1           | 31131.5          | 204.2            | 7                | 256.8            | 5                | 827.9            | 5.43%           |
| Barton          | 953.6            | 6084.3           | 30503.6          | 167.9            | 638              | 284.8            | 692              | 663.1            | 4.68%           |
| Wordnet         | 964.2            | 6582.9           | 34148.7          | 172.9            | 912              | 290.4            | 565              | 267.7            | 0.78%           |
| Taxonomy        | 136.2            | 1182.1           | 9721.4           | 35.37            | 67               | 135.8            | 066              | 413.8            | 11.49%          |
| SGD             | 723.4            | 3966.4           | 28488.6          | 35.75            | 907              | 161.5            | 886              | 537.6            | 4.07%           |
| Linkedmdb       | 427.7            | 2938.2           | 16458.9          | 58.74            | 247              | 178.7            | 561              | 272.0            | 1.89%           |
| DBLP            | 1832.1           | 1089.3           | 49922.4          | 398.0            | 174              | 377.9            | 857              | 558.9            | 6.08%           |
| SP2B            | 1580.9           | 1601.2           | 97796.4          | 321.8            | 504              | 985.8            | 322              | 850.7            | 0.87%           |
| BSBM            | 6038.5           | 4674.2           | 93070.2          | 682.9            | 642              | 1577.9           | 995              | 64.54            | 3.38%           |
| LUBM            | 13.5             | 15               | 15               | 0.849            | 837              | 0                | 0                | 0                | 0.00%           |
| UOBM            | 14.1             | 20.3             | 21               | 1.523            | 884              | 1.251            | 666              | 0                | 0.00%           |
For example, considering the number of friends associated with a student as a measure of popularity, popularity of students in UOBM are similar while, in the real world, some students are much more popular than others. Unless a benchmark generator assigns different roles to resources which makes some resources distinguishable from others, dataset relationship specialty can always identify synthetically generated data. Similar to relationship specialty, some datasets also show attribute specialty, e.g. a movie with tens of aliases. However, the relationships among resources shows much more diverse patterns compared with attributes associated with resources.
4.2.2 Literal Diversity [QO-2]

The diversity of literals is an important factor that influences the quality of an RDF benchmark. Indeed, generating keywords similar to real datasets is always an important task for synthetic data generation. However, existing RDF benchmarks are limited in this aspect. There are two general methods for literal generation: 1) use a small set of predefined words plus a number (for LUBM, UOBM); 2) use random selection from a predefined dictionary containing a large number of possible words (for BSBM, SP2B). It is clear that both methods are limited, and may introduce bias to query evaluation as literals are queried most frequently as keywords. The first method results in low keyword selectivity due to utilizing smaller number of words as literals while the second method results in high keyword selectivity for relatively small benchmark dataset sizes due to the random selection on large number of possible words. Thus, measuring the diversity of literals provides another useful information for differentiating real data from benchmark data.

One straightforward way to measure the diversity of literals in an RDF database is to measure the dictionary size of literals in all attribute triples. This, however, may not work well as some datasets contain billions of attribute triples, creating problems for efficient evaluation. Thus, we use literals from a random sample of attribute edges with size $\mathcal{M}$, and use the number of unique “words” $w_{\mathcal{M}}$ contained by these literals as a measurement of literal diversity of a RDF dataset. The next question is how to determine the value of $\mathcal{M}$. If $\mathcal{M}$ is too small, the sampled attribute edges may not cover all different attribute predicates, and introduce a bias as the literals associated with different attribute predicates can have different ranges, e.g., the number of unique words associated with “#hasGender” and “#hasName” predicates in a social network can be significantly different. Thus, the
sample size $M$ should be large enough to ensure that it covers all predicates of an RDF dataset. However, the value of $M$ should be smaller than the number of attribute edges in all datasets (Wordnet has the smallest number of attribute edges at 0.8 million). In order to identify a good $M$ value, we use different $M$ values to do random sampling of all RDF datasets. We randomly select 20 samples of attribute edges with size $M$ and record the mean, standard deviation, and coefficient of variation of the dictionary sizes of all 20 samples in Table 2 for all RDF datasets. As one can observe, the sample size of 100,000 ($100k$) seems to work well since the coefficient of variation for the dictionary sizes of 20 samples are below 5% for all datasets and most are below 3%. In other words, the dictionary size of sampled attribute edges are stabilized for all datasets for the sample size of 100,000. Dictionary sizes, computed using a random sample size of $M = 100k$ for all datasets are shown in Figure 12. Using dictionary sizes as an estimation for literal diversity, clearly, LUBM and UOBM have limited literal diversity which contain only a small number of unique words. In contrast, SP2B and BSBM have the largest literal diversity due to random selection from a large set of words. Compared with real datasets, SP2B and BSBM have much less number of frequent words which appear multiple times in sampled attribute edges. This leads to an even larger literal diversity for them. Based on extended experiments, for even larger sample sizes, 100,000 attribute edges are enough to capture almost all words in BSBM’s generation files while SP2B has a larger dictionary file to generate literals.

The literal diversity metric highlights a potential limitation of the existing RDF benchmarks. There are lots of work [DM05, WD+08, RC+11] directly related with generating more realistic synthetic literals. However, whether it has a big impact on query
evaluation is another important aspect to explore.

### 4.2.3 Insights from new Metrics

The two new metrics, *relationship specialty* and *literal diversity*, provide an effective way to differentiate the existing RDF benchmarks from real data. They also provide hints on improving the current RDF benchmark generators. *Relationship specialty* indicates that a hidden “role” system exists in real datasets and some resources are distinguishable from others. It provides a finer granularity in understanding the degree distribution by considering different relationship predicates separately. *Literal diversity* shows that more deliberate literal generation process can be employed by RDF benchmark generators to make the benchmark data more realistic.

### 4.3 Dataset Characteristics and RDF-$h$

Dataset *Coherence*, *Relationship Specialty* and *Label Uniformity* provide hints of the effectiveness of neighborhood check for different RDF datasets. High *Coherence* and low *Relationship Specialty* often indicates similar neighborhood structures are uniformly distributed among resources. High *Label Uniformity* (indicating small number of unique terms) reduces the potential pruning rate of checking partial keywords. Thus, datasets with high *Coherence*, low *Relationship Specialty* and high *Label Uniformity* are not likely to benefit much from the signature-based pruning due to the uniform graph structure and literals associated with resources.
4.4 Conclusions and Comments

We demonstrate that using only the coverage and coherence metrics as proposed in [DK+11] is not sufficient to conclude that “current RDF benchmarks have little in common with real data”. We also show that the coverage and coherence metrics are vulnerable to minor updates of a small number of RDF triples, and may provide a biased view of the underlying graph structure of RDF datasets.

Using these benchmark metrics as the starting point, an improved benchmark generator can be provided. A finer granularity should be utilized in RDF benchmark generator by (i) considering different types of resources and different predicates independently and (ii) utilizing a more deliberate literal generation process. The results of these benchmark evaluation metrics can then be used to generate more realistic benchmark data. In the next chapter, we will describe our new application-specific RDF benchmark generator, called RBench.
Chapter 5 RBench: Application-Specific RDF Benchmarking

Benchmark evaluation based on the metrics coherence, relationship specialty and literal diversity in Chapter 4 indicates that the existing RDF benchmarks are limited in their coverage of diverse structures and do not quite mimic the characteristics of real datasets. This motivates us to design an application-specific RDF benchmark generator which can generate benchmarks similar to the real ones. Dataset coherence is a direct evidence that the traditional top-down approach to generate domain-specific benchmarks is not a good option for RDF benchmarking. Using predefined schema to describe all resources of the same type is limited since most real applications use resources from multiple domains and multiple sources. This leads to diverse predicate patterns for resources of the same type (relatively smaller dataset coherence). Relationship specialty shows that uniform relationship patterns are utilized in current RDF benchmarks while a hidden “role” system exists in real datasets. This “role” system makes some resources distinguishable from others. Relationship specialty can also be viewed as a finer granularity in analyzing the degree distribution by considering different relationship predicates separately. The literal diversity highlights potential improvement for the existing RDF benchmarks.

5.1 Problem Definition

Similar to the definition of dataset scaling problem [Tay11] (proposed as the central
problem of application-specific benchmarking for relational databases), the application-
specific benchmarking problem for RDF data is defined as follows.

*Given an RDF data graph $G$, a size scaling factor $s$ and a degree scaling factor $d$, generate a synthetic dataset $\tilde{G}$ which is similar to $G$, and has its number of edges (triples) and the average node degree changed by factors $s$ and $d$, respectively.*

The size scaling factor $s$ here is utilized to describe different benchmarking goals. Scaling up ($s > 1$) an empirical dataset can provide the scalability test environment for applications’ software and hardware design which is essential for applications with sharp dataset size increases. Miniaturizing ($s < 1$) an empirical dataset allows efficient data communication, testing, and analysis. Synthetic copying ($s \approx 1$) of an empirical dataset can be used for anonymization to protect sensitive data. As compared with real datasets, controllable dataset size is an important advantage in performance evaluation using benchmark datasets.

The degree scaling factor $d$ is introduced to simulate different types of data scaling applications. The fact that RDF datasets can change significantly over time even for primitive metrics, e.g., the node degree, is explained [LK+05] as the Densification Power Law which states that the average degree of data graphs increases over time for real applications. Moreover, different applications may require completely different degree scaling factors. Another important reason to use the degree scaling factor in RDF benchmarking is that average degrees of graphs have a significant impact on query evaluation. For graphs with the same number of edges, queries can be much more complex to evaluate on dense graphs than sparse graphs.

A generated benchmark dataset is considered similar to the given dataset if their values
for the dataset evaluation metrics and query evaluation times for different techniques are similar. For the dataset evaluation metrics, we use the *dataset coherence* [DK+11], *relationship specialty* [QO-2] and *literal diversity* [QO-2]. Query evaluation time similarity is motivated by the expectation that the generated benchmarks are useful if they provide a controllable environment with comparable results so that performance of different query techniques on the given dataset can be estimated.

### 5.2 **RBench** [QO-3]

Application-specific design is the key to overcome the limitations of existing domain-specific RDF benchmarks where predefined schemas are utilized. The challenge is to understand characteristics of a given RDF dataset, and define the criteria to generate a set of “similar” synthetic datasets. Next we propose a new framework, called *RBench*, which analyzes features of the graph structure and literal labels of a given RDF dataset, and uses these features to generate benchmark data with the desired scaling factors. *RBench* generates benchmarks in three steps: (1) preprocessing the given RDF dataset $G$; (2)
generating a set of features characterizing the given dataset $G$ in dataset analysis; (3) generating benchmark graph with the required scaling factors based on these features.

**Type System $T$ (1)**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource/Stephen_Cook</td>
<td>rdf:type</td>
<td>#Person</td>
</tr>
<tr>
<td>resource/Manuel_Billum</td>
<td>rdf:type</td>
<td>#Person</td>
</tr>
<tr>
<td>resource/Hao_Wang</td>
<td>rdf:type</td>
<td>#Person</td>
</tr>
<tr>
<td>resource/University_of_Toronto</td>
<td>rdf:type</td>
<td>#University</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>resource/Turing_Award</td>
<td>rdf:type</td>
<td>#Award</td>
</tr>
</tbody>
</table>

**Unknown Type Restore (4)**

**Extract Relationship Triples and Attribute Triples Using Type System (2)**

**Algorithm 1**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Type Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>wikt/University_of_Toronto</td>
<td>Unknown1</td>
</tr>
<tr>
<td>wikt/Harvard_University</td>
<td>Unknown1</td>
</tr>
<tr>
<td>resource/1921-05-20</td>
<td>Unknown2</td>
</tr>
<tr>
<td>resource/1939-12-14</td>
<td>Unknown2</td>
</tr>
</tbody>
</table>

**Figure 14. Dataset Preprocessing**

**5.2.1 Dataset Preprocessing**

Preprocessing a given RDF dataset $G$ has four steps (Figure 14): (1) identifying the type system $T$ associated with $G$ by checking predicate “rdf:type”; (2) identifying relationship triples and attribute triples using type system $T$ in $G$; (3) evaluating the type system $T$ associated with $G$; and (4) restoring the type system for resources without type definition (if the evaluation of $T$ fails).

The motivation for identifying different types of resources and edges/triples is to provide a finer granularity to better understand a given RDF dataset. As explained in
Section 2, attribute triples associated with each resource forms simple graph structure since literal nodes have no outgoing edges while the graph structure of relationship triples is much more complex. The labels of literals in attribute triples convey important semantics, and are frequently queried as keywords while the labels of resources are typically long strings with randomly generated IDs. Thus, the method to analyze relationship triples and attribute triples should take these differences into consideration. The type system associated with a given RDF dataset provides a more effective way to understand its data structure. Resources of the same type often share the same set of attribute predicates, and show similar relationship patterns. This can be observed as most real RDF datasets (6 out of 8 in experiments) have dataset coherence values above 0.5. In other words, the same set of predicates are associated with over half of resources of the same type. This is mainly due to the utilization of popular ontology systems. Many ontologies define standard sets of possible predicates for different types of resources which are adopted in the design of real RDF datasets. Note that some resources are associated with multiple type definitions. Similar to the strategy in [DK+11], different possible type combinations are defined to distinguish resources with multiple types.

Although the type system associated with an RDF dataset provides useful information for dataset analysis, it is not required to be complete, and some resources are not associated with type definition. As RBench performs dataset analysis using the type system, it may become a potential limitation if a large number of resources are not fully utilized in the analysis process due to unknown types. For most of the datasets (both real and synthetic) utilized in section 2, the type systems associated with them are relatively complete, and over 90% of resources are well-defined with types. Resource analysis with defined types
is sufficient for these datasets. But, type systems associated with some datasets are limited, e.g., Yago dataset (more than half of the resources do not have types). Thus, an evaluation of the type system is performed in order to validate the analysis results. For RBench, type system evaluation checks two aspects: 1) percentage of resources with type definition; 2) percentage of triples containing only resources with type definitions. If both percentages exceed the expected threshold (in this paper, 80%), the type system is claimed to be high quality, and passes the evaluation. Otherwise, the original type system fails the evaluation, and the process of restoring unknown types is utilized.

5.2.1.1 Restoring Unknown Types

The goal of restoring unknown types is to provide a complementary type system for datasets which have a large number of resources without type definition for better dataset analysis results. Simply using one new type “unknown” to label all resources without types provides too little information for dataset analysis. As we observed, most of these resources with unknown types result from incomplete type system. For example, missing types for resources integrate from other domains as in the case of the Yago dataset. This means some unknown types may be restored to provide a better type system for dataset analysis. Since URI labels of resources are not required to include semantic information, using these labels to identify the potential types of resources can’t be achieved. However, resources of the same type tend to be described by a similar set of predicates, and some predicates can be utilized as features to distinguish different types of resources. This motivates us to recover unknown types by using predicates. As shown in Figure 14, step (4), the resource predicate table $D_{R,P}$ is constructed by using all triples associated with resources of the unknown
type, which records all possible resource-predicate combinations in these triples. Then, the process to restore types is formulated as a decision tree problem. Internal nodes of the decision tree is partitioned into two branches by checking whether a predicate \( p \) is associated with the resource. Each leaf node represents a quality unknown type. Information gain is calculated by using the coherence metric. In other words, the partition based on predicate \( p \) is high quality if the coherence value of \( D_{R,P} \) improves with the new types. As high coherence value often indicates more structured graph patterns associated with resources of the same type, using the coherence metric to compute information gain can facilitate better feature graph extraction for resources of unknown types. The process to generate the complementary type system is shown in Algorithm 1.

Algorithm 1 begins with labeling all resources without types as one new type \( t_1 \), and finds the predicate to partition \( t_1 \) to achieve the largest information gain by using function \( \text{predPrtn}(t, T, D_{R,P}) \) (steps 1-2). Function \( \text{predPrtn}(t, T, D_{R,P}) \) enumerates all possible partition plans for resources labeled as type \( t \), and calculates the coherence increase for each partition plan. The partition plan with the highest coherence increase is returned. Then, Algorithm 1 finds the type \( t_j \) with the largest coherence increase \( c_{t_j} \) of all partition plans in \( P \), and checks whether \( c_{t_j} \) meets the minimal required gain \( \tau \) (steps 4-5). If \( c_{t_j} \) meets the minimal requirement, \( t_j \) is partitioned into two types \( t_j \) and \( t_{max+1} \) using the partition plan \( (p_{t_j}, c_{t_j}) \) and the partition plans for the new resulting types \( t_j \) and \( t_{max+1} \) are calculated (steps 6-10). If \( c_{t_j} \) is smaller than the minimal requirement, the algorithm terminates, and returns the resulting type system. The run time complexity of Algorithm 1 is \( O(|L_R| \times |T| \times |p|) \) where \( |L_R| \) represents the number of resources without types, \( |T| \).
Algorithm 1: RUT \((L_R, D_{R,P}, \tau)\)

Input: \(L_R\): list of Resources; \(D_{R,P}\): resource, predicate table; \(\tau\): minimal required information gain

Output: \(T\): type system for \(L_R\)

1. add all resource \(r_i \in L_R\) to \(t_1\) and add \(t_1\) to \(T\)

\[\text{predPrtn}(t_1, T, D_{R,P}) \rightarrow p_{t_1}, c_{t_1}\] and add \((p_{t_1}, c_{t_1})\) to \(P\) //\(P\) represents the partition plans

2. WHILE (1)

3. find \((p_{t_j}, c_{t_j}) \in P\) with max \(c_t\) //max information gain

4. IF \(c_{t_j} \geq \tau\)

5. partition \(t_j\) into \(t_j\) and \(t_{max+1}\) by checking whether resource \(r_k \in t_j\) is associated with \(p_{t_j}\) //if true, \(r_k \in t_j\); else \(r_k \in t_{max+1}\)

6. \[\text{predPrtn}(t_j, T, D_{R,P}) \rightarrow p_{t_j}, c_{t_j}\]

7. \[\text{predPrtn}( t_{max+1}, T, D_{R,P}) \rightarrow p_{t_{max+1}}, c_{t_{max+1}}\]

8. update \(t_j\), \(t_{max+1}\) in \(T\)

9. update \((p_{t_j}, c_{t_j})\), \((p_{t_{max+1}}, c_{t_{max+1}})\) in \(P\)

10. ELSE BREAK

11. ENDIF

12. ENDWHILE

13. RETURN \(T\)

FUNCTION \(\text{predPrtn}(t, T, D_{R,P})\)

14. \(p_t = \text{null}, c_t = 0\)
is the number of types in the restored type system, and \(|p|\) is the number of all distinct predicates associated with any resource in \(L_R\). \(|T|\) is bounded by \(\frac{1}{\tau}\) since the coherence value is increased at least by \(\tau\) for each loop (each new type), and the coherence value is at most 1. Thus, the run-time complexity of Algorithm 1 is \(O(|L_R| \times |p|)\) if \(\tau\) is set. The process of restoring unknown types runs offline at most once for each given dataset.

### 5.2.2 Dataset Analysis

To provide a benchmark dataset “similar” to a given dataset \(G\), we consider two aspects of similarity: similar characteristics for (i) literal labels, and (ii) the graph structure. As shown in Figure 15, four dataset features are defined to characterize the given dataset, namely, resource identity and relationship patterns for relationship triples; predicate dictionary and attribute patterns/stars for attribute triples.
5.2.2.1 Resource Identity

We define resource identity as the three-column table shown in Figure 15, containing the resource ID, resource type and the number of relationship edges associated with this resource (denoted as $\mathcal{R}$Degree in Figure 15). Though each resource can be uniquely identified by its URI label, most URI labels convey little semantics, and are not important in synthetic data generation. On the other hand, type of the resource is highly related with its predicates and the graph structure. $\mathcal{R}$Degree describes the importance associated with each resource in the relationship graph. It is also used as a threshold to assign the generated resources to relationship patterns in the benchmark graph generation.
5.2.2.2 Relationship Patterns

We introduce *relationship patterns* to characterize the structure of the given relationship graph (the subgraph of the given RDF graph which contains only relationship edges, e.g., the blue subgraph in Figure 13). By using these patterns as building blocks, a set of similar synthetic graphs with different required sizes can be generated. The relationship graph is analyzed via the resource types rather than the resource IDs to generate *relationship patterns*. For *RBench*, the *relationship patterns* are generated by decomposing the relationship graph into subgraphs. As shown in Figure 15, one *path pattern*, one *star pattern*, and two *edge patterns* are generated, and the occurrences of each pattern are recorded for the given relationship graph.

Choosing among different graph patterns generated by decomposing the relationship graph is a tradeoff between the flexibility to reconstruct the synthetic graphs and the ability to preserve structure similarity. Using only edge *patterns* is a flexible way to reconstruct benchmark graphs while preserving the least structure similarity (only the types of resources implicitly identified by predicates and the occurrences of different predicates are preserved). In comparison, using the entire relationship graph as one pattern, or recursively generating the largest spanning trees as patterns is good in preserving the highest structure similarity. However, with such approaches, only some specific size-scaling factors can be achieved when generating benchmark graphs (degree scaling factor $d \neq 1$ is infeasible).

As a response, *RBench* utilizes three types of *relationship patterns* (in preference order): *path patterns*, *star patterns* and *edge patterns*. The motivation is to achieve a flexible benchmark graph generation while frequently queried structures are preserved. To understand the frequently queried structures, the query workloads from different RDF
benchmarks [BA09, GP+05, ML+11, SH+09, ZC+13] are analyzed. Queries proposed in [BA09, GP+05, SH+09, ZC+13] are targeted at a specific domain. Most of these queries are path- or star-shaped, which contain less than 5 relationship triples. Queries proposed in [ML+11] are clustered from over 30 million real user queries. These queries are more complex, but the number of relationship edges contained in most of them is still small (more than 80% of queries contain less than 5 relationship triples). Based on this observation, small path patterns and star patterns (with less than 5 relationship edges) are used for generating benchmark graphs to achieve a query performance similar to that of the given dataset. Using small patterns also provides flexible building blocks to achieve different scaling factors for benchmark graphs. Path patterns are generated before star patterns for two reasons: (1) path patterns represent transitive relationships among a set of resources, and play an important role in query evaluation; (2) no path patterns will be generated if star patterns are generated first (path patterns can be viewed as multiple two edge star patterns).

The decomposition process of the relationship graph for RBench can be achieved by scanning the relationship triples and maintaining lists of paths with the same-source nodes. For each relationship edge, the process tries to attach the edge to a decomposed path, and generates a new path by checking whether the object node in the edge is a source node for any decomposed path. After generating path patterns and edge patterns, the process tries to combine edge patterns with the same-source nodes to generate star patterns. Generating star patterns with large cardinality is avoided by using a preset threshold (set as 5 in experiments) to constrain the number of edges contained by each star pattern.
5.2.2.3 Predicate Dictionary

Predicate dictionaries characterize attribute predicate literals by recording words and occurrences in them as shown in Figure 15. Predicate dictionaries can be used to generate “similar” synthetic literals for attribute predicates where the word appearing more frequently has a higher probability to be selected to form new literals. This way, the percentage of synthetic literals containing a specific word is similar to the percentage of literals containing the word in the given dataset. This is motivated by the observation that the number of attribute triples containing a specific word (that frequently appears in literals) is almost proportional to the dataset size for real applications. Some attribute predicates are associated with literals which are not textual strings. For these attribute predicates, predicate dictionaries are not used.

5.2.2.4 Attribute Patterns/Stars

The graph structure of attribute triples can be characterized by star patterns as there are no outgoing edges for literal nodes (no paths, no circles). Similar to relationship patterns, these star patterns are abstracted into attribute stars by considering the node types, rather than node labels, as shown in Figure 15. Different resources of the same type may share the same attribute star, and occurrences of each attribute star in the given RDF dataset $G$ are recorded. It is also observed that the number of different attribute stars is significantly smaller than the number of resources, and most resources of the same type share the same set of attribute predicates.
5.2.3 Benchmark Graph Generation

Benchmark graph is generated in three steps: (1) generate resources and literals based on resource identity and predicate dictionaries; (2) generate feature graphs based on attribute stars and relationship patterns; (3) assign the generated resources and literals to the generated feature graphs. An example demonstrating the process of generating benchmark graphs is shown in Figure 16.

5.2.3.1 Resources and Literals

The generated resources are recorded in the same form as resource identity. As the labels of resources are URIs and have little semantics, labels of generated resources are in the form of types with random GUIDs (globally unique identifiers). Types are assigned to generated resources according to the type distribution (the percentage of resources belonging to each type) in resource identity. Also, the distribution of \( \mathcal{R} \)Degrees for resources of the same type in resource identity is utilized to assign \( \mathcal{R} \)Degrees to the generated resources by taking the degree scaling factor into consideration. The number of generated resources is \( \frac{s}{d} \) times the number of resources in the given dataset where \( s \) the size scaling factor, and \( d \) is the degree scaling factor. To ensure the connectivity of the benchmark graph, triples with predicate “\( \text{rdf:type} \)” which describe the types of all generated resources are produced first in \( RBench \).

Generating literals from predicate dictionaries is performed in a similar way as in SP2B [SH+09] and BSBM [BA09]. In order to generate literals more efficiently, large numbers of candidate words for each attribute predicate are pre-generated as plain text files according to the occurrence distribution of these words in predicate dictionaries. Literals
are generated by simply reading a sequence of words from these files. The length (the
number of words) of the generated literals for each attribute predicate is set as the average length of literals in the given dataset. Using *predicate dictionaries* to generate literals is a good option for attribute predicates satisfying two properties: (1) the literals are textual strings; (2) the number of literal nodes is proportional to the dataset size. For other attribute predicates, the set of literals in the given dataset can be directly utilized in the benchmark.

### 5.2.3.2 Feature Graphs and Node Assignment

As shown in Figure 16, feature graphs are *attribute stars* and *relationship patterns*. The generated dataset changes the number of occurrences of *relationship patterns* in the given dataset by the size scaling factor \( \mathcal{S} \). For attribute triples, there are two alternatives: the number of occurrences of *attribute stars* in the generated dataset can be (1) the size scaling factor \( \mathcal{S} \) times that in the given dataset; (2) or \( \frac{\mathcal{S}}{d} \) times that in the given dataset, where \( d \) is the degree scaling factor. With the first option, the number of triples in the resulting benchmark graph is exactly \( \mathcal{S} \) times that in the given dataset. However, for real datasets that grow over time, the number of attribute triples associated with each resource may remain almost constant. Since the number of triples and the average degree in the generated benchmark are estimates, we consider both options (1) and (2).

Assignment of resources to generated *attribute stars* is based on the resource type of their root nodes. As shown in Figure 16, the generated resource with type “Person” can be assigned to any *attribute star* rooted at “Person”. The literals assigned to generated *attribute stars* are extracted from pre-generated files based on attribute predicates. Assignment of resources to generated *relationship patterns* should not only match the type of the resource, but also consider the generated \( \mathcal{R} \)Degree. The rule of resource assignment
to relationship patterns is that the sum of the degree of a resource in all assigned relationship patterns is not to exceed its generated $\mathcal{R}$Degree. Triples are generated simultaneously with node assignment for each edge in feature graphs. Node assignment process may generate duplicate triples when the same pair of resources is assigned to a specific predicate multiple times (either in one feature graph or multiple feature graphs). As the analysis of attribute stars and relationship patterns contains no duplicate triples, resources in these duplicate triples are swapped to generate distinct triples to ensure that the number of generated triples match the size requirement (empirically, the probability to generate duplicate triples is low as the number of distinct resource pairs is significantly larger than the number of generated triples for most RDF datasets).

5.3 Query Workload Generation

Query workloads defined for existing benchmarks [BA09, GP+05, SH+09, ZC+13] are strictly domain-specific, and limited in their coverage of diverse graph structures. Most
queries are defined with SPARQL, a graph matching query language that uses graph patterns as basic query units. Attribute edges with specified labels provide the most effective way to locate the matching resources. The more attributes specified in the query, the less matching resources are found. In comparison, relationship edges in query patterns often lead to nested joins among resources, resulting in an increased number of intermediate matching results. Thus, the types of queries utilized in RBench are defined based on the structures of relationship edges. Five types of queries are defined (based on the shapes of graph structures involving relationship edges similar to relationship patterns): node queries, edge queries, star queries, path queries, and subgraph queries (including tree queries and cyclic queries). Queries defined with similar structure types are also utilized in [AO14-1].

Query workload generation utilizes the results in dataset analysis to propose a systematic way to generate different types of queries. The first four types of queries are generated based on attribute stars and relationship patterns, and are generated in parallel with the benchmark generation. The top $k$, where $k$ is an adjustable parameter, resource assignment combinations for each unique relationship pattern are recorded, and attribute triples associated with these recorded resources are retrieved. Similarly, for attribute stars, the top $k$ resource and literal combinations assigned to each unique attribute star are recorded. Recorded attribute stars and relationship patterns with assigned resources and literals are utilized as the candidate queries. The process to generate RBench queries has the following four steps, which is illustrated in Figure 17 for a star query: (1) choose a relationship pattern according to the desired type, and formulate as triple patterns; (2) choose a recorded resource combination for this relationship pattern, and retrieve the
attribute triples associated with the assigned resources; (3) formulate attribute triples as triple patterns by relabeling the resource node accordingly (e.g., “?Person #familyName ‘Cook’”); (4) formulate \( RBench \) query to identify resource IDs by using a subset of these triple patterns in query generation. The main advantages of utilizing attribute stars and relationship patterns in generating query workload are: (a) all predicates of benchmark data are covered by candidate queries; (b) frequent relationship patterns among resources are covered by candidate queries; (c) it is easier to generate a set of identical queries for benchmarks with different scaling parameters by utilizing the candidate queries for the same set of relationship patterns. However, these queries may provide a biased coverage of the graph structure for the resulting benchmark graphs since the analysis of relationship patterns is based on the given graph. This bias can be reduced by using subgraph queries which are discussed next.

Subgraph queries are generated by first randomly selecting a subgraph from the benchmark graph, and then processing the subgraph to generate SPARQL queries by using the types of resources in formulating triple patterns. Subgraph selection begins by selecting a small component of the benchmark graph, and recursively adds one random edge adjacent to the already selected subgraph until the query template size requirement is reached (the selection takes the types of edges into consideration). For cyclic queries, Tarjan's strongly connected components algorithm [Tar72] is utilized to identify cycles in the benchmark graph. The process to generate SPARQL queries from a subgraph is similar to generating other types of queries. Triple patterns are formulated by using resource types, and SPARQL queries are generated to identify resource IDs by using a subset of these triple patterns. Subgraph queries provide additional graph structure coverage for \( RBench \).
*RBench* emphasizes benchmark data generation and query workload generation in the context of viewing and querying RDF data as graphs. This fits the trend in the database community, and the requirement of graph-based functionalities to query RDF data, such as querying for paths. Many new RDF management systems [AO14-2, EI07, ZO+14] rely on more sophisticated, structure-based indexes. Note that query workload generation utilized in *RBench* does not consider some complex functionalities of SPARQL, such as aggregation, and property paths. Most of these complex functionalities require pre-knowledge for the given datasets and users can generate SPARQL queries with complex functionalities by using subgraph queries. The design of *RBench* also takes RDF reasoning into consideration. *RBench* generation is based on the type system and predicates associated with the given dataset which preserve characteristics of the ontologies utilized in it. For relation patterns, *path patterns* are considered as the first priority before *star patterns* and *edge patterns* to preserve the transitive connections in the given dataset. However, reasoning queries are more related with a specific domain (a set of ontologies), and specifying reasoning queries often needs pre-knowledge of the dataset. Thus, the query workload generation for *RBench* focuses on generating graph templates directly from the given dataset. Comparing with benchmarks which focus on reasoning queries (LUBM [GP+05] and UOBM [ZC+13]), the same set of queries can be executed on *RBench* benchmarks generated from them without any modification.

### 5.4 RBench Evaluation

We evaluate *RBench* for three aspects: (i) time and memory complexity of *RBench*, (ii) benchmark evaluation, and (iii) query evaluation analysis. *RBench* framework is
implemented with Visual C# 2010 and SQL Server 2008. All experiments were performed on a 2.93GHZ Intel(R) Xeon X5570 machine with 48GB ram running Windows Server 2008 R2.

5.4.1 Time and Memory Complexity Analysis

*RBench* generator is designed to preprocess and analyze each given RDF dataset once, and use the dataset analysis results to generate benchmarks with different parameters. The results of dataset preprocessing and dataset analysis are stored in SQL tables. Benchmarks are generated in six steps:

1. Parser Step: The given RDF dataset is scanned to generate three tables in SQL: relationship edge table, attribute edge table and resource type table.

2. Triple Extraction: The resource type table is loaded in memory to build dictionary look up for the types of each resource, and triples containing only resources defined with types are extracted by scanning the relationship edge table and attribute edge table. During this scan, resource identity table and predicate dictionary table are generated.

3. Type Restore: The current type system is evaluated. If the type system passes the evaluation, we continue to step 4. Otherwise, restoring unknown types is needed (by running Algorithm 1), and we go to step 2.

4. Pattern Decomposition: *Relationship patterns* and *attribute stars* are generated from the extracted relationship and attribute triples.

5. Resource Generation: The required number of resources are generated based on parameters *s* and *d*.

6. Triple Generation: Relationship triples are generated by assigning resources to
relationship patterns and attribute triples are generated by assigning resources to attribute stars.

The most time-consuming part of RBench generator is writing the results into the database. By loading the resource type table into memory, the time complexity for steps 1, 2 and 4 are linear in the size of the given dataset. The time complexity of Algorithm 1 is $O(|L_R| \times |p|)$ as explained in Section 3.1.1. Compared with resource generation (step 5), resource assignment to relationship patterns (step 6) is more time consuming since the $\mathcal{R}$Degrees for all generated resources need to be maintained and updated. To provide efficient triple generation, the generated resources are loaded into memory to build the dictionary look-up. After step 4, only four tables are necessary to generate benchmarks: resource identity table, predicate dictionary table, relationship pattern table and attribute star table. Compared with the given RDF dataset, these four tables are much smaller (10-30% of disk space). The highest memory overhead of RBench is loading resource type table in step 2, and maintaining the generated resources with $\mathcal{R}$Degrees in step 6. Utilization of efficient resource type look-ups for triple extraction and triple generation processes is a must for generating benchmarks with millions of triples.

Time and space efficiency tests for RBench are performed by generating benchmarks from both real and synthetic datasets. For real datasets, we focus on evaluating the time complexity of generating benchmark graphs (step 5 and step 6). Figure 18 (a) shows benchmark graph generation time for RBench from DBLP dataset (~20 million triples) and Yago dataset (~20 million triples) for different sizes in number of million triples (here, $d$ is 1 to ensure that the number of resources generated increases linearly with benchmark size). For synthetic datasets, we first generate LUBM datasets with different sizes (20 M
Benchmark Generation Time for Real Datasets

Run Time Evaluation of RBench for LUBM Datasets
to 100 M triples). \textit{RBench} generator is then utilized to generate a benchmark dataset for each LUBM dataset by setting $s = 5$ and $d = 1$. The time for each step (except step 3 as LUBM has a complete type system) is shown in Figure 18 (b), and the peak of memory overhead for each step is also identified in Figure 18 (c). As observed, the dataset preprocessing and dataset analysis times are linear with the size of the given dataset, while the benchmark graph generation time is linear with the size of the generated benchmark.

Note that, the type restoring process (step 3) is only required for Yago. \textbf{Algorithm 1} runs in less than 5 minutes on the Yago dataset with over 2 million resources without type definition. By observing the results, four unknown types are generated for Yago dataset as wiki resources, wiki category resources, wordnet resources and date resources. As Yago

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure18}
\caption{Time and Memory Complexity of \textit{RBench} for Real and Benchmark Datasets}
\end{figure}
dataset brings knowledge from both Wikipedia and Wordnet, this result provides a better
dataset analysis for Yago dataset. Triple generation (step 6) has the highest memory
overhead since all generated resources with $\mathcal{R}$Degrees are maintained in main memory.

5.4.2 Dataset Metrics & RBench

Benchmark evaluation aims to validate the claim that benchmark datasets generated
by RBench have similar structure as the given RDF dataset. For this experiment, RBench
is utilized to generate benchmarks from six datasets (both real and synthetic) utilized in
Section 2. Three dataset metrics (dataset coherence, relationship specialty and literal
diversity) are utilized to evaluate the generated RBench benchmarks. Results are shown in
Figure 19 (RBench benchmarks are labeled as R-given dataset). For Figure 19 (a), (b) and
(c), the degree scaling factor is set as 1 to generate RBench benchmarks (different size
scaling factors are tested which give similar evaluation results for all datasets). As observed,
the benchmark datasets generated by RBench yield similar evaluation results as the given
dataset for all three metrics. Detailed analyses of these results are in following subsections.

5.4.2.1 Literal Diversity

The literal diversity of RBench datasets and the given datasets are identical by design,
as shown in Figure 19 (a) and 20 (a), due to utilizing attribute stars and predicate
dictionaries features. Attribute stars provide information about the distribution of different
attribute predicates and the attribute patterns associated with resources of each type.
Predicate dictionaries capture the dictionary information for each attribute predicate, and
are utilized to simulate the new literals. By utilizing attribute stars and predicate
dictionaries in data generation, RBench benchmarks inherit the characteristics of attribute triples (both predicates and literals) from the given dataset. Different scaling factors have no impact on literal diversity evaluation.

5.4.2.2 Dataset Coherence

The dataset coherence measures how uniformly predicates are distributed among resources of each type. RBench provides an effective way to preserve the graph structures among types and predicates by utilizing relationship patterns and attribute stars. Relationship patterns and attribute stars are generated by considering the types of resources and the associated predicates in dataset analysis. Thus, predicates associated with resources of each type remain the same for RBench benchmarks and the given dataset. Feature graphs generated as star shapes provide identical structures between the center resource and its associated predicates. Since resources are all assigned as the centers for attribute stars, and attribute predicates are distributed identically among resources of each type between benchmarks and the given dataset. The situation is different for relationship predicates: 1) relationship patterns are in the shapes of paths, stars and edges; 2) resources are not always assigned to the centers of patterns with star shape. Since the generated resources are assigned to relationship patterns by considering types and ṚDegrees only, the distribution of relationship predicates among resources of each type may be slightly different from the given RDF dataset. As shown in Figure 19 (b) and 20 (b), dataset coherence values of RBench benchmarks are slightly smaller than the given RDF datasets. As observed in the coverage metric computation, resource assignment to relationship patterns yields minor variance on the distribution of relationship predicates. For
(a) Literal Diversity Evaluation for RBench

(b) Dataset Coherence Evaluation for RBench
Figure 19. Benchmark Evaluation Results for Literal Diversity (a), Dataset Coherence (b) and Relationship Specialty (c, d)
(a) Literal Diversity Evaluation for RBench

(b) Dataset Coherence Evaluation for RBench
relationship predicates which are originally associated with most resources of a type in the given dataset, this variance leads to a small reduction in coverage values. As different degree scaling factors can be utilized to generate benchmark data, RBench benchmarks can achieve different dataset coherence values from the same given dataset.

A smaller than 1 degree scaling factor value results in (i) a much sparser graph, and (ii) the same predicates being associated with smaller numbers of resources, which lead to smaller dataset coherence values. Dataset coherence values increase if the degree scaling factor is increased to a value larger than 1.
5.4.2.3 Relationship Specialty

*Star patterns* (one type of *relationship patterns*) and the utilization of the generated \( R \)Degrees in resource assignment are important to preserve *relationship specialty* of the given dataset. Most of the resources associated with the same relationship predicate significantly more frequently than others are decomposed as *star patterns*. These *star patterns* are assigned with new resources with large simulated \( R \)Degrees. The utilization of \( R \)Degrees provides a way to imply a hidden “role” system similar to real datasets. As observed in Figure 19 (c), *relationship specialty* values for *RBench* benchmarks and the given RDF datasets are not exactly the same; but the trend of *relationship specialty* for different datasets are illustrated in a consistent way between the given datasets and *RBench* benchmarks. Note that there is a significant difference between the *relationship specialty* values of synthetic datasets and real datasets, which is also preserved in the benchmarks generated by *RBench*.

Different degree scaling factors has little impact on relationship specialty since relationship specialty is measured by kurtosis, and degree scaling factors scale the distribution of relationship predicates for each resource uniformly. This is shown in Figure 19 (d) using *RBench* benchmarks generated from Yago and DBLP with different degree scaling factors.

5.4.3 Query Evaluation Analysis

Query evaluation analysis of *RBench* has two parts: 1) comparing *RBench* with the existing application-specific RDF benchmark generator [8]; 2) comparing *RBench* benchmarks with scaling datasets both from real applications and benchmarks.
5.4.3.1 Experimental Set Up

**Query Techniques:** Three query algorithms are utilized in the query evaluation analysis, namely, Spath [ZH10], Stwig [SW+12] and RDF3X [NW08]. Spath [ZH10] is a graph template matching algorithm which takes paths as basic join units, and utilizes neighborhood signature-based pruning to improve query performance. Stwig [SW+12] is a graph join algorithm which takes Stwig as basic join units, and utilizes the trinity memory (similar functionalities are implemented in single machine environment). RDF3X [NW08] is a SPARQL query algorithm which takes triples/edges as basic join units, and utilizes indexes over 6 permutations of the three dimensions (Subjects, Predicates and Objects) of RDF triples. Spath and Stwig are revised to support directed graphs with Visual C# 2010 (named as Spath+ and Stwig+ in experiments). The motivation to utilize these three query algorithms are: 1) to cover different basic join units (path pattern for Spath, star pattern for Stwig and edge pattern for RDF3X); and 2) to utilize diverse techniques to accelerate query processing (neighborhood index for Spath, trinity memory for Stwig, and dimension indexes for RDF3X).

**Query Generation:** Since the three query algorithms utilize different types of graph patterns as basic join units, subgraph queries are used to provide results without bias in query evaluation. Since Spath and Stwig do not utilize edge labels in query evaluation, the subgraph queries are formulated without specifying the predicates. Subgraph queries are generated as size 6 (6 triples/edges) with at most two attribute edges.

5.4.3.2 Application-Specific Benchmark Comparison

The application-specific benchmark generator [DK+11] (referred as *AO benchmark*)
is the only comparable benchmark generator for RBench. By using AO benchmark generator, a series of benchmarks with smaller sizes and smaller coherence values can be generated by selectively removing triples from the given dataset. AO benchmark focuses on providing benchmarks with varied “structuredness” from the given dataset and show that changing the characteristic (coherence) of the dataset changes query execution times.

In comparison, RBench is designed to generate benchmarks preserving the characteristic of the given dataset. To show this difference, Stwig and Spath are chosen to evaluate subgraph queries for benchmarks generated by AO Benchmark and RBench. Five datasets are utilized: LUBM, AO-LUBM, R-LUBM, SP2B and R-SP2B. All five datasets utilized have similar sizes (about 1 million triples). LUBM is generated directly by using the LUBM generator. For AO-LUBM and R-LUBM, they are both generated from a larger LUBM dataset (with about 1.5 million triples). The coherence value of AO-LUBM is smaller than that of LUBM and is the same as the coherence of SP2B. Two sets of subgraph queries are generated for LUBM and SP2B (40 queries with 6 edges). The average query time of these 40 subgraph queries on different datasets are shown in Figure 21.

As observed, the query evaluation results of Spath and Stwig on SP2B are completely different compared with LUBM. By reducing the coherence of LUBM from 0.938 to 0.796, the queries are executed much faster on AO-LUBM than on LUBM. This reaffirms that changing the characteristics of the dataset (coherence) affects query evaluation times. In comparison, query evaluation shows similar results for RBench-generated datasets and the given datasets (both LUBM and SP2B), confirming that RBench generates benchmark datasets preserve the characteristics of the given datasets. Compared with RBench, the biggest drawback of the AO benchmark generator is that it can only provide benchmarks
with sizes smaller than that of the given datasets.

5.4.3.3 Scaling Dataset Comparison

A major goal of the application-specific benchmarking is to provide a controllable testing environment for different applications. Different size scaling factor values are utilized to perform different tasks. For datasets approaching billion triples (e.g., DBpedia and YAGO), providing a smaller copy for efficient data communication and analysis is important. For application deployment, scalability testing which requires scaling up the existing dataset is critical to analyze potential workloads in the future. The purpose of scaling dataset comparison is to analyze whether the benchmarks with varied scaling factors generated by RBench provide a testing environment similar to the given datasets,
Query Evaluation for realBench and Real Datasets (Spath+)

- DBLP
- R-DBLP d=1

Query Evaluation for realBench and Real Datasets (Stwig+)

- DBLP
- R-DBLP d=1

126
Figure 22. Query Performance Evaluation for $R$-DBLP and DBLP (Spath+, Stwig+ and RDF3X)

(a) Query Evaluation for RBench and LUBM Datasets (Spath+)

(b) Query Evaluation for realBench and Real Datasets (RDF3X)
Figure 23. Query Performance Evaluation for R-LUBM, LUBM and LUBMD (Spath+, Stwig+ and RDF3X)
i.e., whether the query performance evaluation of the same set of queries is consistent between the *RBench* benchmarks and the given datasets with varied sizes. For this purpose, two datasets are chosen: publication dataset (DBLP, real dataset) and university dataset (LUBM, benchmark dataset).

For DBLP, we use the yearly updated versions to generate five DBLP datasets with different sizes which contain publications of different time periods (M represents million): 0.25M triples (1980-84), 0.5M triples (1980-86), 1M triples (1980-90), 2M triples (1980-94) and 4M triples (1980-99). To generate the corresponding *RBench* benchmarks, the DBLP dataset containing publications from year 1980-90 (1 M triples) is utilized as the given dataset to analyze dataset features. Since the average node degree for different DBLP datasets remains almost the same over the years (slight increase), R-DBLP is generated by using the parameter $d$ as 1 and $s$ as 0.25, 0.5, 1, 2 and 4. Since *RBench* utilizes the DBLP dataset with 1M triples as the given dataset, different tasks of real application testing (*scaling up and miniaturization* of the given dataset) are addressed. The evaluated query workload is a set of 40 subgraph queries generated from the first version of the DBLP dataset (0.25M triples). Matching results of these queries on the first version of the DBLP dataset are duplicated on all R-LUBM datasets to ensure that the same set of queries can be executed. Query evaluation results (average run time for 40 queries) of algorithms Spath, Stwig and RDF3X are shown in Figure 22. As observed, all three algorithms show similar patterns between the DBLP dataset and the *RBench* benchmark for varied sizes. This confirms that *RBench* benchmarks provide a similar testing environment as those of the real scaling datasets.

LUBM dataset is used for both size scaling and degree scaling experiments. For size
scaling, LUBM datasets are generated by changing the number of universities in LUBM generator. In this way, the size of LUBM dataset is increased by including more universities while its average node degree remains almost the same. To generate LUBM datasets for degree scaling (named as LUBMD), some preset parameters in the LUBM generator are multiplied by a scaling factor to allow more triples generated for each university, such as min/max number of publications for each faculty, min/max number of courses taken by a student. In this way, the LUBM dataset size is increased by including more triples in a university while the number of resources in a university changes very slightly. By using both scaling methods, 7 datasets are generated: LUBM2 (14 universities, 2M triples), LUBM4 (28 universities, 4M triples), LUBM8 (56 universities, 8M triples), LUBM16 (110 universities, 16M triples), LUBMD4 (14 universities, 4M triples), LUBMD8 (14 universities, 8M triples) and LUBMD16 (14 universities, 16M triples). R-LUBM is generated from LUBM2 by using two different settings of parameters, $d = 1$ and $d = s$ ($s$ is set as 1, 2, 4 and 8). Setting $d$ as 1 means that the dataset size increase is merely a result of newly added resources as the average degree for each resource remains the same. In the opposite direction, setting $d$ as $s$ indicates that the number of resources in a dataset remains the same; but the connections among resources are drastically increased. The results (average query run time for 40 subgraph queries) are shown in Figure 23. Compared with the LUBM dataset with similar average node degree, the running time of the same set of queries degrades much faster as the dataset size increases for all algorithms on LUBMD. As observed, R-LUBM with different parameters simulates both scaling situations successfully, and the query evaluation results on R-LUBM are consistent with those from LUBM and LUBMD datasets. Thus, R-Bench can provide a different testing
environment to simulate different scaling datasets (both size and degree scaling) which are important for generating RDF benchmarks. Though the purpose here is not to compare the query performance of different algorithms, RDF3X tends to yield the best results for datasets with varied sizes. Indexing all permutations of the three dimensions of triples and the utilization of hash join and merge join are beneficial.

5.5 Related Work

Many RDF benchmarks have been developed recently. Considering the design, we categorize them as domain-specific [AO14-1, AB+14, BA09, GP+05, ML+11, SH+09, ZC+13] and application-specific [DK+11] benchmarks. LUBM [GP+05], BSBM [BA09] and SP2B [SH+09] are among the first RDF benchmarks which use strict pre-defined domain schema to generate synthetic data. Due to the small numbers of types and predicates utilized in these three benchmarks, the generated synthetic data has a limited variability; and inherent graph structures are repetitively generated. BSBM and SP2B both use SPARQL queries to test performances of different RDF management systems; and a variety of SPARQL features (such as FILTER, and OPTIONAL) are evaluated. The queries proposed by LUBM benchmark are plain SPARQL queries which focus on evaluating systems with different reasoning capabilities. UOBM [ZC+13] extends the ontology utilized in LUBM to include disjunctive axioms and negation, more relationships, and more properties for different types of resources. WatDiv [AO14-1] (Waterloo SPARQL Diversity Test Suite) is a great effort to allow customizable domain schema by using a dataset description language. WatDiv enables users to create their own test databases and/or to customize existing ones. The query template generator is also provided.
by WatDiv to generate queries with different graph structures. DBPSB benchmark [ML+11] is the first domain-specific benchmark which uses non-artificial data and queries. The data are generated by duplicating or sampling triples directly from DBpedia. Queries utilized in DBPSB are analysed and clustered from 31.5M real user queries. The Linked Data Benchmark Council (LDBC) [AB+14] works to bridge the best practices of the TPC (Transaction Processing Council) to the benchmark of RDF database industry. Two benchmarks, the Social Network Benchmark (SNB) and the Semantic Publishing Benchmark (SPB) are currently under development.

The AO benchmark proposed in [DK+11] is the first effort of application-specific benchmarking for RDF data. Compared with RBench, the AO benchmark is designed to generate benchmarks with varied structuredness from the given dataset. By computing the impact of removing predicates for resources, the AO benchmark formulates an integer programming problem for selectively removing triples in the given dataset to achieve the desired smaller dataset coherence and smaller dataset sizes. Coverage and coherence metrics are introduced [DK+11], as an intuitive way to combine primitive metrics into one single measure of structuredness of RDF datasets. A comprehensive study of the structuredness of RDF graphs is also presented in [AS+14]. A framework is proposed [AS+14] to discover a partitioning of the entities of an RDF graph into subsets which have high structuredness with respect to a specific function chosen by the user.

5.6 Conclusions and Comments

RBench provides application-specific benchmarking for RDF data. By utilizing the three dataset evaluation metrics (dataset coherence, relationship specialty and literal
diversity), we show that existing domain-specific RDF benchmarks are limited in coverage for the diverse graph structures of real RDF datasets, and can not provide a comparable testing environment. As RDF data model is defined without a strict schema, the application-specific benchmarking is a better fit to generate RDF benchmarks. Motivated by this, we propose RBench which can be utilized to generate benchmarks from any given RDF dataset having similar characteristics of its graph structure and literal labels. The process to generate RBench benchmarks contains three steps: dataset preprocessing, dataset analysis and benchmark graph generation. Four features are utilized to analyze the characteristics of the given RDF dataset as resource identity, relationship patterns, predicate dictionary and attribute stars. Benchmark data are generated by simulating feature graphs based on relationship patterns and attribute stars, and then new resources and literals are assigned to feature graphs to reconstruct the benchmark graph. Consistent with the application-specific design, the query workload generation of RBench can generate different types of queries in parallel with the benchmark generation. Three aspects of RBench are evaluated through experiments: time and memory complexity of benchmark generation, benchmark dataset evaluation, and query evaluation analysis. By using real and synthetic datasets, we empirically show that RBench benchmarks can achieve different benchmark generation tasks with size and degree scaling, and provide the expected testing environment similar as the given scaling datasets.

There are several directions to extend the RBench framework. Providing benchmarks generated from multiple given datasets to simulate the results of the data integration of different applications; and addressing the attribute correlation and evaluating different choices of relationship patterns in different contexts for benchmark generation are among
the future directions to explore.
Chapter 6 Utilization of RDF-h for Biomedical Applications

One important motivation for querying graph structured RDF data is its popularity in real applications. In this chapter, we focus on demonstrating the querying power and efficiency of RDF-h for biomedical applications [QO-4].

6.1 Motivation

In post-genomic biology, networks are commonly used to model molecular interactions, as well as high-level associations among various biological entities. These entities include biomolecules, ligands, cellular functions, functional modules, biological processes, tissues, organisms, and phenotypes. Networks are useful to represent a broad range of interactions among biomolecules, including protein-protein interactions (PPIs) [MP+13], gene co-expression [CB+14], transcriptional regulation [HK+07], metabolic pathways [PG+13], genetic interactions [TL+04], and signaling pathways [RT+14]. Higher level associations represented by networks include gene-disease associations [GC+07], clinical similarity or co-morbidity of diseases [GC+07], disease-drug associations [VM+11], molecular response to drugs [VM10], functional annotation of genes and proteins, and evolutionary relationships among molecules and organisms [LQ14].
The current state-of-the-art in the querying and analysis of disparate interaction and association data is limited to querying each type of data in isolation, or downloading different datasets in bulk and joining them in house for specific analysis and mining tasks. In other words, it is not straightforward for a researcher to identify or infer indirect associations among biological entities by incorporating data in multiple forms. Recently, heterogeneous network models that incorporate multiple types of interactions and associations have been shown to be effective in the identification of unknown relationships among biomolecules, biological processes, diseases and drugs. The applications of such integrative models include disease gene prioritization [LP10], drug repositioning [CL+12], and functional annotation of proteins [CP+14].

We demonstrate the promise of representing heterogeneous network models as RDF graph and utilize RDF-h to facilitate exploratory querying of integrated biological networks. Here, the term “integrated biological network” refers to the collection of all known functional, physical and statistical interactions, as well as associations among biological entities. RDF is the first W3C standard for enriching information resources on the web with detailed descriptions (i.e. Meta data). It is the commonly used data model for the linked data, and knowledge bases that are shared and exchanged on the web. An RDF dataset consists of a set of triples, in the form (s,p,o), stating that a subject s has the property p whose value is the object o. RDF data can also be visualized as a graph where subjects and objects are nodes and properties (predicates) are edges. Unique identifiers (URI’s) can be used for subjects, properties or objects to uniquely refer to entities, relationships or concepts. Literals can also be used for objects [KM14].

RDF can easily represent wide range of data and information from structured, semi-
structured, or unstructured sources. Thus, it enables seamless interoperability and integration of the data on the web. Since each RDF triple (edge in graph representation) corresponds to a binary predicate, it lends itself for reasoning and inference based applications as well. Traditional approaches for biomedical applications require querying different biology datasets (e.g., UniProt, MeSH, OMIM, Reactome) using a relational database to build a knowledge base by integrating various query results. In comparison, using an integrated RDF dataset offers several advantages. Namely, creating a knowledgebase using a relational database requires parsers for each dataset, designing a schema, tuning the system. Typically, the file formats, and schema changes occur frequently, which requires costly updates for the parsers, the schema, and the queries to keep the system up-to-date and functional. Using RDF, data is integrated seamlessly, queries do not depend on the schema as in relational databases, and the integrated RDF dataset can be queried directly for interactions and associations between disparate biological entities for information coming from different sources.

6.2 Query Examples

For a broad range of biomedical applications, researchers query large public databases to interpret their findings, to verify their predictions, or to identify relationships that will help develop novel hypotheses. Common queries that are utilized by biomedical scientists include the exploration of various types of information on a single protein, enrichment analyses for sets of genes or proteins, and identification of genes/protein associated with a given phenotype, function, process, tissue, or drug. However, the current state-of-the-art in integrated mining of biological data shows that disparate databases, when considered
together, contain information that cannot be directly extracted by such queries [LP10, CL+12, CP+14]. Inspired by the success of integrative data mining efforts, we propose that queries that integrate multiple types of association and integration data will enable scientists to more effectively explore indirect relationships among biological entities. Such queries include the following:

- **Overlap in the molecular bases of diseases**: The overlap in the identity of genes that are associated with different diseases may be useful in discovering unknown relationships among different diseases [GC+07]. Motivated by this observation, biomedical scientists may be interested in searching for all diseases that share at least a number of gene associations with a disease of interest. Such a question can be addressed

---

**Figure 24. Query Examples and Motivation**

(blue nodes represent user specified entities, red nodes represent query output entities; double line edges represent path connections)
by a graph template matching query shown in Figure 24(a). In this query, the user specifies a disease (QD) and queries for all diseases (TD) such that a protein associated with QD is also associated with TD.

- **Shared molecular interactions between diseases**: Diseases with similar molecular etiology may not necessarily overlap in terms of the identity of gene associations, but the relationship may be revealed through molecular interactions among these genes [LK12]. This provides systems level insights into the shared molecular mechanisms among different diseases [MS15]. Such relationships can be discovered using a query like the one shown in Figure 24(b). In this query, the user specifies a disease (QD) and queries for all diseases (TD) such that a protein associated with QD interacts with a protein associated with TD.

- **Shared molecular interactions between diseases and drugs**: Drug repositioning has recently become a prominent application in computational biology [LZ+15]. This is due to the need to repurpose established drugs to eliminate cost, as well as the opportunity provided by omic data to discover unknown relationships between biomolecules targeted by drugs and biological processes involved in pathogenesis. Ability to query for drugs that share a number of molecular interactions with a given disease (i.e., molecular interactions between the targets of the drug and the genes associated with the disease) can provide an excellent starting point for identifying candidate drugs for repositioning [VM+11]. A graph template matching query that can be used for this purpose is shown in Figure 24(c). In this query, the user specifies a disease (QD) and queries for all drugs (TU) such that a protein associated with QD interacts with a protein associated with TU.

- **Network schemas**: Integrated mining of molecular interaction networks and functional annotations led to the identification of network schemas, i.e., small subgraphs
of functional terms that recur frequently in molecular interaction networks [BN08]. These network schemas provide insights into conserved functional modules and the design principles of cellular networks [PK+07]. A graph template matching query such as the one in Figure 24(d) can be used to search for network schemas involving specific biological processes or molecular functions. In this query, the user specifies a specific molecular function (QF) and queries for all proteins associated with QF that interact with each other.

- **Shared pathways between diseases**: Organization of systems biology knowledge in the form of pathways provides well-established, reliable, and tractable access to state-of-the-art knowledge on biological processes. For this reason, incorporation of pathway data in analyzing the relationship among phenotypes provides information that is complementary to high-throughput interaction data that is organized into networks [KOY12]. For example, identification of shared pathways among different diseases can be useful in understanding similarities in disease development and progression [PK+07]. A sample graph template matching query that can be used for this purpose is shown in Figure 24(e). In this query, the user specifies a disease (QD) and queries for all diseases (TD) such that a pathway that contains a protein associated with QD also contains a protein associated with TD.

As demonstrated by these examples, the ability to seamlessly query interactions and associations among biological entities enables biomedical scientists to get quick answers to a broad range of sophisticated questions on the relationships between these entities. While the examples above are mostly limited to mining indirect associations, it is also possible to use the proposed framework to identify relationships with mechanistic biological interpretations. For example, experimental data derived from gene knock-out or
RNA interference experiments are commonly used to identify associations among gene and proteins in terms of their effect on phenotype [BC+10, VZ14]. For signaling proteins, strong positive or negative correlation between two proteins’ influence on the phenotype may be indicative of common downstream effects [FP02]. Scientists can quickly discover potential candidates for these common effects by querying for paths that go through these proteins and converge into the same node in the integrated network of protein-protein and transcriptional regulatory interactions. Today, the most common way of interpreting observed experimental associations among a group of proteins is to identify subnetworks that connect the proteins of interest, e.g., using Steiner tree based algorithms [BB10].

Biomedical researchers often use commercial software (e.g., Ingenuity Pathway Analysis, Pathway Studio) that do not provide algorithmic transparency or clearly defined criteria for the identified subnetworks. To this end, semantically meaningful queries that integrate multiple data types can generate significant mechanistic insights into the relationships among proteins of interest and provide the researchers with new ways of thinking about their research questions.

While all of the sample queries listed above can be currently processed by downloading bulk datasets from multiple databases and subsequent processing and joining of these datasets, this is often time-consuming and challenging for many biomedical scientists. Therefore, the main contribution of this study is the development of a querying framework that provides a “shortcut” to data integration in the bioinformatics pipeline. This is useful for two types of applications: 1) “targeted” queries, that is when the researcher is interested in identifying new associations for one more specific entities (e.g., a group of genes, a particular disease, a particular drug). 2) “high-throughput” queries
for mining tasks that is when the researcher is interested in identifying all associations that exhibit a specific pattern (e.g., all disease-drug pairs that share a reasonably large number of interactions).

6.3 Datasets

The UniProt Knowledgebase (UniProtKB) [UN15] is a central hub of protein information which provides an integrated view of association and interaction data from different biomedical datasets. One important motivation of UniProtKB is to allow users query the related but dispersed information across disparate protein related datasets. Each protein entry recorded in UniProtKB provides a variety of information related to this protein including protein and gene names (mnemonic name, structured name and alternate names), protein sequences, protein function, catalytic activity, co-factors, subcellular
localization, patterns of expression, protein–protein interactions, and disease association.

Besides the rich information provided for each protein, another advantage of UniProtKB is its high update rate and availability in different formats. UniProtKB data is released every 4 weeks to provide the most up to date protein information in multiple formats including plain text, XML, RDF and GFF.

The decisive factor to choose UniProtKB rather than another database available in RDF format for our experiment relies on its high quality and accuracy of data integration. Since RDF format has no pre-defined schema, RDF data is designed to integrate with ease by combining the triples from different sources directly if unified resource identifiers are utilized. For UniProtKB, each entry undergoes both automated and manual checks to ensure the high accuracy and consistency of the data before it is integrated. The automated check is performed through a quality control software to ensure the correctness of syntax and verification of different biological rules for the entry. Besides this, the manual review process provides extra effort to ensure that all relevant literature, annotation and analysis results are included. As the correctness of the querying results across multiple datasets is determined by the lowest quality data integrated, the high quality standards provided by UniProtKB is essential to provide high confidence of querying results in the experiments.

**6.3.1 Integrated Datasets**

In this section, we describe the three integrated datasets in UniProtKB which are extracted and queried in our experiments reported in the next section: IntAct (protein–protein interaction), Reactome (pathway), and OMIM (disease and phenotype).
6.3.1.1 IntAct

IntAct [KA+12] provides open-source molecular interaction data populated by interactions curated from the literature, as well as from direct data depositions. The information within the IntAct database primarily consists of protein–protein interaction (PPI) data. An important aspect of the IntAct dataset is that each entry in IntAct is peer reviewed by a senior curator, and not released until accepted by that curator. UniProtKB database is readily integrated with the IntAct database to provide protein–protein interaction data. In order to meet the required quality standard of UniProtKB, only a subset of high quality interactions are imported from IntAct based on a statistical scoring system. A score threshold is chosen by UniProtKB to exclude binary interactions supported by only one experimental observation. In addition to the score-based filter, a set of defined rules are utilized to exclude certain types of data, such as interactions observed in larger complexes, or interactions that have not been experimentally validated. By using these strict criteria, only experimentally validated binary interactions supported by multiple observations are imported into UniProtKB.

6.3.1.2 Reactome

Reactome [CM+14] is a manually curated open-source human pathway and reaction dataset. In order to provide a unified identifier, Reactome merges pathway identifier mapping, over-representation and expression analysis tools into a single portal. Reactome uses UniProtKB protein identifiers to provide a list of pathways in which the protein functions. Compared with the pathway annotation provided by UniProtKB directly, the cross referenced Reactome pathways provide more complete information for each protein.
6.3.1.3 OMIM

Online Mendelian Inheritance in Man (OMIM) database [HS02] is utilized in UniProtKB to provide disease/phenotype information for disease annotations associated with proteins. OMIM is a comprehensive, authoritative and timely knowledgebase of human genes and genetic disorders. Each OMIM entry has a full-text summary of a genetically determined phenotype. UniProtKB carefully links the OMIM entry with the protein entry and describes the natural variant(s) of the protein sequence potentially associated with disease according to the scientific literature.

6.3.2 Data Extraction

The data utilized in our experiments is a subset of triples from the UniProtKB RDF dataset. We focus on only human proteins that currently have active entries. The UniProtKB raw data, downloaded from the UniProt website on 4-10-2015, contains about 150 million triples from 971,583 (both reviewed and unreviewed) protein entries. Note that different protein isoforms are represented as different protein entries in UniProtKB. In order to provide a more concise RDF graph to support efficient signature-based indexes, only protein entries associated with at least one of the following statements are extracted: disease annotation, function annotation, PTM annotation, cofactor annotation, subunit annotation and protein interaction. For each protein entry, the following properties are extracted: protein names (mnemonic name, recommended name and alternate name), protein organisms (including the taxonomy information), protein keywords, protein tissues, protein gene information (including different gene labels), and protein pathway information (Reactome pathway associated with the protein). The extracted RDF graph
contains 89,915 proteins (including different protein isoforms), 4,211 diseases, 1,278 pathways, 18,243 interactions and 35,063 annotations. The size of the extracted RDF graph is about 11.6 million triples including 9 million triples from the taxonomy data. Data extraction can be systematically done from any version of the UniProtKB RDF graph, and more types of annotations can be extracted by changing the specification of the data extraction process.

6.4 Experimental Analysis

The proposed framework is implemented with Visual C# 2010 and SQL Server 2008. All experiments were performed on a 2.93GHZ Intel(R) Xeon machine with 48GB RAM running Windows Server 2008 R2. The average space needed for the NI index is $O(N(\mu/2)^{d_{max}/m})$, where $N$ is the number of vertices in $G$, $\mu$ is the average node degree, $d_{max}$ is the maximum hops of neighbors indexed and $m$ is the binning factor. The NI index with a larger $d_{max}$ value results in higher pruning power and ability to handle connection edges with large distance constraints at the cost of requiring more storage space. By using the IDMap index to hash the RDF labels into IDs, the 2 hop NI index achieves a similar size as the original RDF graph while 3 hop NI index is 8 times larger. As we explain in Section 3.2, 2 hops NI index can evaluate connection edges with distance constraints up to 4 hops efficiently which is sufficient for all proposed queries. Here, we decide to use 2 hops NI indexes for graph template matching.

In this section, we primarily investigate the ease of utilization of GBE query templates to specify integrated queries and the strength of these queries in discovering interesting patterns across the integrated network from disparate biological entities. For this purpose,
we focus on 10 queries that require integrated querying across multiple interaction and/or association datasets. We categorize these 10 queries into two groups: 1) single protein patterns: querying the relationships among one protein and other types of resources; 2) multiple protein patterns: querying protein-protein joining based on protein-protein interactions, shared pathways, shared diseases, or shared function. Rather than displaying the results in tabular form, we export query results into Cytoscape [LF+10] to produce meaningfully summarized graphs. For each query, the query results can produce multiple summarized graphs by specifying different relationships among various types of resources.

6.4.1 Single Protein Patterns

Single protein patterns focus on querying relationships between one protein and other types of resources across different datasets. Four queries are proposed as single protein patterns:

Q1. Finding pathways that contain at least one protein associated with “Breast Cancer”.

Q2. Finding molecular functions that are associated with at least one protein associated with “Breast Cancer”.

Q3. Finding cancers that are associated with at least one protein that is also associated with “Breast Cancer”. (Motivated by Figure 24 (a))

Q4. Finding tissues that are associated with at least one protein which is associated with “Breast Cancer”.

To evaluate these four queries, at least two types of biological resources/entities need to be integrated: protein resource (Uniprot) and disease resource (OMIM). UniProtKB has already linked protein resource with disease information which makes some of queries
solvable through manual effort, e.g. one can search all protein entries associated “Breast Cancer” on Uniprot website and manually check all these entries to find any other cancers associated. As UniProtKB also provides a beta SPARQL endpoint which allows user to specify SPARQL queries, some of these queries can also be specified in SPARQL. However, both manual check and specifying the complete SPARQL queries requires...
SELECT 
?Cancer
WHERE 
{
?Protein1  #type <http://purl.uniprot.org/core/Protein>
#annotation disease_annotation1
#interaction Interaction1
?Protein2  #type <http://purl.uniprot.org/core/Protein>
#annotation disease_annotation2
#interaction Interaction2
disease_annotation1 #disease ?disease1
disease_annotation2 #disease ?disease2
?disease1  #prefLabel "Breast Cancer"
?disease2  #prefLabel ?Cancer
Interaction1  #participant ?EBI1
#participant ?EBI2
Interaction2  #participant ?EBI1
#participant ?EBI2
FILTER regex(?Cancer, "*Cancer.*")
}

(a) SPARQL

(b) Complete Query Template

(c) GBE Query Template

Figure 27. Query Specifications of Q5

significantly more effort compared with using GBE query templates in our framework.

In Figure 26, we illustrate three possible ways to specify Q3. One can easily observe
that both alternatives of specifying the complete query template (Figure 26 (b)) and the SPARQL query (Figure 26 (a)) need not only strong database background but also a good understanding of the underlying graph structure of UniProt RDF graph. In comparison, by using connection edges and partial keywords, GBE query templates can be much simpler to formulate. As shown in Figure 26 (c), the GBE query template uses connection edges to avoid specifying the intermediate annotation node and the disease ID node in identifying the relationship between a protein and disease name. With a little background of UniProt RDF dataset, the GBE query framework provides simpler query specifications as demonstrated by the abstract queries in Figure 24. Note that, using connection edges may yield more results compared to specifying all edges in the complete graph template. For all these four queries, the simplified GBE query templates return the same set of results as the complete graph template on the extracted UniProtKB graph. The GBE framework also supports displaying the results in graph templates. Thus, the researchers can choose a subset of the results they are interested, and display them as graph templates to perform exploratory search. Due to space limits, the details of using the GBE framework to display query results as templates to perform exploratory search is not included here.

Q1 returns 60 Reactome pathways potentially associated with “Breast Cancer”. 13 protein function annotations associated with proteins which are associated with “Breast Cancer” are found in Q2. Q3 discovers 5 cancers that have overlapping molecular bases with “Breast Cancer”. 23 tissues are identified in Q4 related to proteins associated with “Breast Cancer”. As we explain in Section 2, identification of diseases with overlapping molecular bases may be useful in identifying unknown disease-disease relationships. Here, we display the results of Q3 in Figure 5 using Cytoscape. In Figure 5 (a), we display the
connections among proteins and diseases identified in the query results of Q3 by making the disease and the protein in each query result as the source and target in Cytoscape. To further provide more clear relationships between “Breast Cancer” and other cancers, another summarized graph is produced in Figure 28 (b) by making the two diseases in each query result as the source and target in Cytoscape. The width of the edge between two diseases is determined by the number of proteins shared by this disease pair. The size of the cancer node is based on the number of shared proteins. The more proteins shared between one disease and “Breast Cancer”, the larger disease node is (same for edge width, the more shared proteins, the wider the edge is). One can observe that “Pancreatic Cancer” and “Breast Cancer” have strong shared molecular bases (3 shared proteins) and are more likely to relate with each other. Such a network is utilized in [GC+07] which can be constructed via a single query in our framework.
Figure 29. Disease-Disease Relationships for Q5

Figure 30. Full Visualization of Query Results for Q5
6.4.2 Multiple Protein Patterns

Compared to single protein patterns, multiple protein patterns are more complex for query specification and evaluation. We consider three types of protein-protein joining conditions: protein-protein interaction, shared functions and shared pathways. Some of the multiple protein pattern queries also require joining proteins based on multiple criteria.

6.4.2.1 Protein-Protein Interaction

Protein–protein interactions reveal functional relationships between genes. Though a large number of protein–protein interaction datasets are available, no direct way is proposed for querying patterns of protein–protein interactions related with other types of biological entities, e.g. certain phenotypes. By using the extracted UniprotKB graph, we demonstrate the potential power of seamlessly querying the integrated biology datasets.

We consider two queries here:

Q5. Finding cancers associated with at least one protein that interacts with another protein associated with “Breast Cancer”. (Motivated by Figure 24 (b))

Q6. Finding all pairs of cancers associated with pairs of proteins that interact with each other. (Motivated by Figure 24 (b))

Specifying queries of multiple protein patterns is more difficult compared to queries involving single protein patterns. Similar to Figure 26 of Q3, all three possible ways (SPARQL, complete graph template and simplified GBE template) to specify Q5 are shown in Figure 27. Note that the binary interaction between two proteins are identified by two EBI observations which is the requirement of UniprotKB dataset.

Q6 is a more general form of Q5 as Q6 tries to identify all pairs of cancers with shared
molecular interactions while Q5 specifies one disease as “Breast Cancer”. Complete visualization of query results for Q5 is shown in Figure 30. The width of the edge between two proteins (represents the binary interaction between two proteins) is proportional to the
number of disease pairs shared this interaction. Four protein-protein interaction patterns are identified to connect different cancers with “Breast cancer”. Compared with Figure 28 (a), one can observe that there is no edge between BRCA1 protein and “Breast Cancer” in Figure 30. This is a result of graph isomorphism matching of the query template as each query node should match a unique node in the RDF graph. Since there is no protein associated with other cancers which interacts with BRCA1 protein, no query results contain an edge between BRCA1 protein and “Breast Cancer” for Q5. The summarized disease-disease relationships identified by Q5 is shown in Figure 29. Similar to Figure 28 (b), the edge width represents the number of protein-protein interactions shared by the respective disease pair. For Q6, the complete disease-disease relationships among all pairs of cancers is shown in Figure 31. These two queries are motivated by Section 2, Figure 24 (b) as finding shared molecular interactions between diseases.

6.4.2.2 Shared Functions

Identifying network schemas requires protein-protein joining through both shared functions and protein-protein interactions as shown in Section 2, Figure 24 (c). We consider one query:

Q7. Finding pairs of proteins which have general function description as “Component of the Mediator complex” and interact with each other. (Motivated by Figure 24 (c))

The query results are displayed in Figure 32 using Cytoscape. Three groups of proteins are identified where proteins closely interact with each other in the same group.
6.4.2.3 Shared Pathways

Understanding the roles of proteins in higher order interconnected pathways is critical to understanding molecular reactions at the cellular level. Some conceptualizations of protein-protein interactions also consider proteins in the same pathway interact with each other directly or indirectly. Identifying protein-protein patterns with shared pathways provides complementary information to binary protein-protein interactions. Here, we consider three queries:

Q8. Finding any pair of proteins that are involved in the same pathway and are both associated with “Breast Cancer”.

Q9. Finding cancers associated with a protein that is involved in the same pathway with a protein associated with “Breast Cancer”. (Motivated by Figure 24 (e))

Q10. Finding two proteins that are involved in the same pathway and interact with each other such that one protein is associated with “Breast Cancer” and the other protein is associated with another cancer.

The query results of Q8 are visualized in Figure 33. We observe that there are two groups of proteins where the proteins in each group are closely related with each other through shared pathways. Four proteins associated with three cancers which interact with each other and share three common pathways are identified in Q10 as shown in Figure 34. Note that “Breast Cancer”, “Pancreatic Cancer” and “Breast-Ovarian Cancer” are also observed to share molecular interactions in Figure 29 and Figure 31. The query results of Q9 returns 24 distinct proteins, 31 pathways and 15 cancers (not include “Breast Cancer”). Q9 is motivated by Section 2, Figure 24(e) as finding diseases with shared pathways.
6.4.3 Query Evaluation Efficiency

The neighborhood signature index is considered as a strong candidate to evaluate GBE.
templates as it can: 1) reduce the unnecessary candidates in subgraph isomorphism matching, and 2) handle connection edges with short distance constraints efficiently. By indexing all neighbor node IDs in the NI index, the connection edges can be evaluated efficiently. Query running time and the number of matching results for all 10 queries are shown in Table 1. We find that single protein pattern queries are more efficient to evaluate compared with multiple protein pattern queries. For single protein pattern query, Q2 requires the longest running time as the component of matching any protein associated with function annotations results in a large number of matching candidates which leads to a large number of connection checks. Q6 is the most time consuming multiple protein pattern query as each decomposed component is small and all components are connected through connection edges. Compared with Q6, Q5 requires less effort as one of the cancers is specified as “Breast cancer” which significantly reduces the intermediate matches. Except for Q6, all queries can be evaluated in less than thirty seconds or so which is good enough to support real time biomedical applications.

Table 4. Query Running Times for all Queries

<table>
<thead>
<tr>
<th>Query ID</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Results</td>
<td>72</td>
<td>13</td>
<td>10</td>
<td>48</td>
<td>25</td>
</tr>
<tr>
<td>Run Time</td>
<td>3.86 s</td>
<td>10.26 s</td>
<td>1.42 s</td>
<td>1.06 s</td>
<td>23.31 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query ID</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Results</td>
<td>67</td>
<td>20</td>
<td>32</td>
<td>158</td>
<td>8</td>
</tr>
<tr>
<td>Run Time</td>
<td>156.4 s</td>
<td>15.71 s</td>
<td>15.79 s</td>
<td>22.09 s</td>
<td>31.74 s</td>
</tr>
</tbody>
</table>

6.5 Conclusions and Comments

We have demonstrated that, using simple graph templates to query an integrated RDF knowledge base, graph template matching based querying is a promising tool to express
sophisticated questions, and discover hidden connections among biological or biomedical entities from diverse datasets. For data to be published or exchanged in the web, RDF is increasingly being adopted, and vast amounts of RDF data are already available. The availability of ever increasing amounts of RDF data in various fields of biomedical applications from public and private sources, adds even more to the potential of querying integrated RDF data sets for new insights and discovering knowledge about hidden associations among biological and medical entities.
Chapter 7 Optimizations for RDF-h

Utilization of RDF-h for biomedical applications demonstrates its effectiveness in both specifying complex query templates and integrated querying of disparate data from different sources. It satisfies the ease of use criteria for biology researchers with little database knowledge by providing an intuitive interface to propose graph template queries. By using UniprotKB dataset, we demonstrate that RDF-h provides an efficient way to discover hidden relationships among different biological entities that cannot be readily obtained by other query techniques. Despite these advantages offered by RDF-h framework, we also identify several critical issues which need to be addressed properly in order to provide better support for querying biomedical RDF data. This leads to an improved design of indexes for RDF-h and two optimized connectivity check algorithms which will be introduced in this chapter.

7.1 Motivation

Properties of integrated querying of biomedical data motivate the following three optimization directions for improving RDF-h framework.

1. Handling frequent updates:

Most biomedical datasets update routinely to adapt new discovered knowledge from user uploads, new published materials and new experimental results, e.g. UniProt is updated every four weeks. These routine updates often affect small amount of existing data
and insert some new facts cured from new research. However, current design of RDF-h doesn’t support incremental updates and all indexes needs to be rebuilt if any updates occur. Considering the large size of the entire dataset and frequency of routine updates, rebuilding the indexes from scratch is too costly to maintain the usability of RDF-h framework.

2. Improved connectivity check performance:

The query optimization and query evaluation of RDF-h framework originally focus on improving the query performance of single component queries considering random queries. Random query generation is also utilized in experiment analysis. However, the integrated querying of interactions and associations among different biological entities across datasets requires heavy utilization of connection edges to simplify query template specification. This leads to queries with small components connected by connection edges. As query time for matching these small components is relatively short, the query optimization should focus on improving the performance of connectivity check process.

3. Index space compression for NI index:

Signature-based neighborhood index is space costly as it stores the breadth first search results up to certain hops for each node. By hashing the RDF labels into simple integer IDs, NI index is much more space efficient compared with traditional neighborhood indexes, e.g. indexes for Spath and GraphQL. However, the index space needs further optimization for biomedical data as the biological entities are closely connected by different biology networks. As we state in chapter 3, the average space of NI index is \( O(N\left(\frac{\mu}{2}\right)^{d_{max}}) \) where \( N \) is the number of vertices in \( G \), \( \mu \) is the average node degree and \( d_{max} \) is the maximum hops of neighborhood index. Since most biological entities have a large node degree, the size of 3 hop NI index is over 2 GB for the extracted Uniprot dataset (about 0.4
GB). As the Uniprot dataset keep increasing its size and more aspects of the Uniprot dataset need to be extracted to answer sophisticated biology queries, better compression of the neighborhood index is necessary.

To address these three aspects, we propose ByteMap and NB (Neighborhood Byte) indexes which replace the integer IDs utilized in IDMap and NI indexes with binary IDs. Two optimized connectivity check algorithms are then proposed to maximize the performance benefit from NB index. The design of the ByteMap and NB indexes is also optimized for main memory data processing in a similar way as BitWeaving and ByteSlice.

7.1.1 BitWeaving

In order to provide real-time query template matching of RDF-h framework, storing the indexes in main memory is necessary due to the complexity of subgraph isomorphism and evaluation of connection edges. Main memory database management is an active research area in database research community [LP13, BA+13, KN+12, BT+13, BH+09, WP+09, BL+11, FL+15]. Here, we focus on a recent technique, called BitWeaving [LP13], which is designed to exploit the parallelism at the bit level for in memory data processing. The key operation supported in BitWeaving is predicate scan which takes as input a list of \( n \) k-bit codes and a predicate with a basic comparison, e.g. \( (=, \neq, <, >, \leq, \geq, \text{Between}) \) on a single column. An important goal is to run predicate scans at the speed of the processing units, and exploit all the functionality that is available inside modern processors. As the most important operations in RDF-h are predicate scans based on ID intervals which can be represented as bit codes, the optimization provided in BitWeaving is also applicable here.
Figure 35. BitWeaving Storage Layout

Figure 36. BitWeaving Predicate Scan & Early Pruning
We illustrate the storage layout and two optimizations for predicate scan on an integer column of BitWeaving technique in Figure 35 and Figure 36. The integers are recorded as four bit codes and stored in vertical partitions. The design is utilized to exploit the SIMD (single instruction multiple data) instruction provided by processor. In other words, it stores one bit for multiple integers/codes (8 bits in Figure 35 and 36, defined as a word in BitWeaving) as a single memory chunk which is processed at one run. In Figure 35, the significance of each bit in the 4 bit representation of an integer is indicated. When the predicate scan is processed on these integers, the 4-bit codes are scanned according to the significance of each bit in four runs. If certain comparison is performed (< 5 in Figure 36), the significant bit can be utilized to prune the candidates at early runs. As an example, the first integer c1 is coded as 1010. In the first run of the predicate scan, we found c1’s first significant bit is 1 which is larger than the first significant bit of 5 (which is 0). We can safely label c1 as not-matched without checking other bits. In the same manner, we can determine all the predicate scan results in the third run which means we can achieve an early termination for this predicate scan. This specific storage layout is proposed as Vertical Bit-Parallel (VBP) which is good for range based predicate scan. However, the reconstructing/looking-up a value under the VBP layout is expensive as the bits of a value are spread across different words. Another storage layout proposed in [LP13] is Horizontal Bit-Parallel (HBP). In Horizontal Bit-Parallel (HBP), we can store multiple codes in the same memory chunk which can achieve paralleled look ups. These two storage layouts provide a tradeoff between the look up performance and scan performance. To provide a storage layout which can achieve both advantages of efficient look ups and scans, ByteSlice [FL+15] technique stores all codes as vertical bytes that fully leverages SIMD
data-parallelism. By checking byte at a time, ByteSlice can achieve both early termination and parallel look ups at the same time.

### 7.2 ByteMap & Neighborhood Byte Indexes

![ByteMap](image1)

#### Node & ID Table

<table>
<thead>
<tr>
<th>Node</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://purl.uniprot.org/diseases/426">http://purl.uniprot.org/diseases/426</a></td>
<td>0x04ED7700</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/diseases/4270">http://purl.uniprot.org/diseases/4270</a></td>
<td>0x04ED7800</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/evoc/0100049">http://purl.uniprot.org/evoc/0100049</a></td>
<td>0x04F03600</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/evoc/0100050">http://purl.uniprot.org/evoc/0100050</a></td>
<td>0x04F03700</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/keywords/389">http://purl.uniprot.org/keywords/389</a></td>
<td>0x050D2100</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/keywords/38">http://purl.uniprot.org/keywords/38</a></td>
<td>0x050D21C0</td>
</tr>
<tr>
<td><a href="http://purl.uniprot.org/keywords/390">http://purl.uniprot.org/keywords/390</a></td>
<td>0x050D2200</td>
</tr>
</tbody>
</table>

**Figure 37. ByteMap**

![NB Index](image2)

#### NB Index Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Distance</th>
<th>Sigbyte</th>
<th>Flag</th>
<th>Count</th>
<th>IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04ED77</td>
<td>1</td>
<td>0x00</td>
<td>0</td>
<td>3</td>
<td>0x5352B823D345</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x02</td>
<td>0</td>
<td>1</td>
<td>0x37AE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x03</td>
<td>0</td>
<td>2</td>
<td>0x30BA96D5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x04</td>
<td>0</td>
<td>1</td>
<td>0xDF9A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x05</td>
<td>1</td>
<td>3</td>
<td>0x0D21C0010500016A400</td>
</tr>
<tr>
<td>0x04ED77</td>
<td>2</td>
<td>0x01</td>
<td>0</td>
<td>2</td>
<td>0x27F73088</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x02</td>
<td>0</td>
<td>1</td>
<td>0xB82B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x03</td>
<td>0</td>
<td>1</td>
<td>0xA377</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x04</td>
<td>0</td>
<td>1</td>
<td>0xDF99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x05</td>
<td>0</td>
<td>2</td>
<td>0xF5E0F92</td>
</tr>
</tbody>
</table>

**Figure 38. NB Index**

Inspired by BitWeaving technique, we re-design the indexes in RDF-h framework as ByteMap and Neighborhood Byte (NB) indexes.
ByteMap Index maps RDF labels into byte IDs in lexicographic order. In other words, the ID of \( l_i \) should be smaller than the ID of \( l_j \) if the label \( l_i \) comes before the label \( l_j \) in lexicographic order. Each label ID contains 4 bytes (representing a 32 bit integer) as shown in Figure 3. The last byte is designed to handle the updates and its first bit is utilized as the flag bit to indicate the utilization of the last byte. The three byte IDs allow ByteMap indexing RDF graphs with 16 million nodes. If the RDF graph has more than tens of millions of nodes, the last byte can also be utilized to handle extremely large graphs. When a new node is insert into the ByteMap, it locates the node label with the largest ID coming before it in lexicographic order. Then, it sets the flag bit in the last byte as 1 and uses the last byte to assign an ID. The strategy to assign IDs to new nodes will be explained in Section 7.4.

Similar as the NI index, the NB index materializes the results of BFS with one or more hops for the nodes in the graph. Rather than grouping the neighbors into ID intervals using the binning factor \( m \) in NI index, the NB index groups IDs of neighbors by its first byte labeled as significant byte shown in Figure 38. Each index entry for node \( n_i \in G \) contains six columns:

- ID of \( n_i \) (if the flag bit is 0, the first three bytes of the byte ID is utilized);
- Distance: Distance is the length of the shortest path from \( n_i \) to the indexed neighbor node;
- Significant byte of the neighbor IDs;
- Flag bit: Indication of the utilization of the last byte of the byte IDs.
- Number of indexed neighbor nodes;
Neighbor node IDs without Sigbyte.

The neighbor nodes sharing the same distance and significant byte (first byte of the byte ID) are grouped in one index entry, ordered by their IDs. The flag bit of an index entry is set as 1 if any ID of the neighbors utilize the last byte of the byte ID. Otherwise, the flag of an index entry is set as 0 and all neighbor IDs utilize the first three bytes of the byte ID. The maximum indexed distance is defined as $d_{max}$.

### 7.2.1 Index Space Efficiency

Compared with the Neighborhood Interval (NI) index, the Neighborhood Byte (NB) index is much more space efficient for three reasons:

1. Most node IDs indexed in NB index only use the first three bytes of the 4 byte ID. Compared with the integer IDs utilized in NI index, utilizing three bytes to index nodes saves index space.

2. The utilization of significant byte to group neighbor IDs into the same index entry achieve a higher space compression. As the significant byte is shared by all neighbors, the IDs of neighbors indexed in each entry don’t need to store the first byte of the byte IDs. Since most index entries contain more than one indexed neighbor, the NB index is much more space efficient as the significant byte is stored once for all neighbors in an index entry.

3. As the neighbor IDs indexed in NI index are integers, it requires an extra delimiter to identify ID for each neighbor in the IDs column. Since the IDs of NB index are fixed length, we can separate the ID for each neighbor from a binary array which stores all IDs of neighbors. As shown in Figure 38, neighbor IDs can be separated...
from the IDs column as they either contain two bytes (flag is 0) or three bytes (flag is 1). Thus, the neighbor IDs indexed in the NB index doesn’t need the delimiter.

For different indexed distances and datasets, NB index can achieve an overall 80% space saving compared with NI index which is demonstrated in Section 7.5.

7.3 Optimized Connectivity Check

The most important operation for connectivity check is to find the intersection of the sets of the neighbor IDs of two nodes $n_i$ and $n_j$ using the NB index. The design of NB index provides an efficient neighbor ID equality scan similar as the Vertical Bit-Parallel (VBP) storage in BitWeaving technique. As each neighbor ID composes four bytes (32 bit), finding the intersection of the sets of neighbor IDs from two ID lists can be performed in a hierarchical way where each byte is checked in the order of its significance. If any byte of the two neighbors are not equal, we can achieve an early termination.

As we explained in chapter 3, connection edges can be either inside of one query component or joining two different components. The number of connectivity checks for a connection edge inside component is linear with the size of the component candidate set, while the number of connectivity checks for a connection edge between components depends on the product of the sizes of components’ candidate sets. For a connection edge inside a component, the connectivity check process should focus on optimizing the hierarchical scan of neighbor IDs for each component candidate. On the other hand, the connectivity check process should provide a way to store the index entry scan results for each distinct candidate node to avoid scanning the same node multiple times when evaluating the connection edge between components. Especially for connection edges
between two components when the sizes of candidate sets of both components are large, storing the index scan results improves the connectivity check performance significantly. Thus, we propose two connectivity check algorithms, merge check algorithm for connection edge inside a single component and cached check algorithm for connection edge between two components.

**Algorithm 1**: MergeCheck \((C_{G_c}, i, j, d_c, NB)\)

**Input**: Candidate Set of Component \(G_c\): \(C_{G_c}\), Connection edge between \(q_i\) and \(q_j\), Distance Constraint \(d_c\), Neighborhood Byte Index \(NB\)

**Output**: Candidate Set of Component \(C_{G_c}'\) passing Connectivity Check

1. Foreach candidate match \(c\) in \(C_{G_c}\)
2. If (Merge\((c[i], c[j], d_c, NB)\))
3. Add \(c\) to \(C_{G_c}'\)
4. Return \(C_{G_c}'\)

bool Merge \((n_i, n_j, d_c, NB)\)

5. Index Entries of \(n_i, n_j\) as \(NB_i\) and \(NB_j\) (order by \(Entry.Sig, Entry.Dis\))
6. \(e_i = NB_i.count, e_j = NB_j.count, l = 0, m = 0,\)
7. \(d_1 = Ceil\left(\frac{d_c}{2}\right), d_2 = Ceil\left(\frac{d_c}{2}\right) - d_c\)
8. While \((l < e_i \& m < e_j)\)
9. While \((l < e_i \& NB_i[l].Dis > d_1 \& NB_i[l].Dis < 0)\)
10. \(l++\)
11. While \((m < e_j \& NB_j[m].Dis > 0 \& NB_j[m].Dis < d_2)\)
12. \(m++\)
13. If \((NB_i[l].Sig < NB_j[m].Sig)\)
14. \(l++\)
15. Else If \((NB_i[l].Sig > NB_j[m].Sig)\)
16. \(m++\)
Else If $(NB_i[l].\text{Sig} == NB_j[m].\text{Sig})$

Add $NB_i[l].\text{IDs}$ to $L_i$, Add $NB_j[l].\text{IDs}$ to $L_j$

While($NB_i[l].\text{Sig} == NB_i[l+1].\text{Sig} \&\& NB_i[l+1].\text{Dis} \leq d_1$)

Add $NB_i[l+1].\text{IDs}$ to $L_i$

$l++$

While($NB_j[m].\text{Sig} == NB_j[m+1].\text{Sig} \&\& NB_j[m].\text{Dis} < 0$)

Add $NB_j[m+1].\text{IDs}$ to $L_j$

$m++$

Equal Scan $L_i$ with $L_j$ as $\psi_{l,m}$ (using VBP)

If ($\psi_{l,m} != \phi$)

Return true

Else $l++, m++$

Return false

7.3.1 Merge Check Algorithm

Merge Check algorithm provides an efficient index scan process to check whether the neighbors of nodes $n_i$ and $n_j$ are intersected based on the distance constraint. The index entries of nodes $n_i$, $n_j$ are ordered by the significant byte and distance in the neighborhood byte index. The neighbor IDs in each index entry are also sorted. By this design, the neighbor ID equality check between two lists of index entries can be performed in a hierarchical merge process (line 7-20). For each index entry, it first checks whether the entry is in the distance constraint (line 8-11). Then, it checks whether the significant bytes of the two entries are equal. If the significant bytes of the two entries are not equal, it skips the entry with smaller significant byte (line 12-15). If the two entries have the same significant byte, it then checks all bytes of neighbor IDs in the order of its significance. If
all bytes of the two neighbors are equal, the algorithm returns true. If there are no more index entries in either list, it returns false to indicate no common neighbors are found (i.e. the intersection of the sets of the neighbors of the two nodes is empty).

The Merge Check algorithm has the benefit of fully utilizing the four byte IDs in NB index. By grouping the neighbor IDs with the same significant byte in one entry, the equality check can skip all neighbors in an entry if the significant byte of two index entries are not equal. As the neighbor IDs of the index entries are stored using the Vertical Bit-Parallel layout, equality scans for neighbor IDs are processed similar as the BitWeaving technique which achieves early termination. As it is demonstrated by the experimental results (Section 7.6.2), Merge Check algorithm is very efficient.

7.3.2 Cached Check Algorithm

Though Merge Check algorithm provides efficient connectivity check for connection edges inside a component, it does not perform well for connection edges between components especially when the sizes of candidate sets of both components are large. As evaluation of connection edges between components requires nested joins on each pair of the component candidates, the index entries of the same node are scanned multiple times during the connectivity checks. The limitation of merge check is no intermediate scan results are stored to facilitate further process, e.g. indexes entries violating the distance constraints (line 8-11 in Algorithm 1) and index entries with same significant byte which needs to be combined (line 18-23 in Algorithm 1).

<table>
<thead>
<tr>
<th>Algorithm 2: CachedCheck $(C_{G_{c1}}, i, C_{G_{c2}}, j, d_c, NB)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Candidate Set of Component $G_{c1}$: $C_{G_{c1}}$, Candidate Set of Component $G_{c2}$: $C_{G_{c2}}$</td>
</tr>
</tbody>
</table>
$C_{G_{c_2}}$, Connection edge between $q_i$ in $G_{c_1}$ and $q_j$ in $G_{c_2}$, Distance Constraint $d_c$, NB Index $NB$

Output: Candidate Set of joint Component $C_{G_c'}$ by Connectivity Check

1. Foreach candidate match $c_1$ in $G_{c_1}$
   2. If (!CachedSig.contains($c_1[i]$))
      3. Index Entries of $c_1[i]$ as $NB_i$ (order by Entry.Sig, Entry.Dis)
      4. Foreach Index Entry $e$ in $NB_i$
         5. If ($e$.Dis $\leq$ $Ceil\left(\frac{d_c}{2}\right)$ & $e$.Dis $>$ 0)
            6. If ($M_{iDS}$.contains ($e$.Sig))
               7. Add $e$.Sig to $L_{sig}$, $M_{iDS}[e$.Sig].add($e$.IDs)
            8. Else
               9. Add $e$.Sig to $L_{sig}$, Add $e$.Sig, $e$.IDs to $M_{iDS}$
      10. Add $c_1[i]$, $L_{sig}$ to CachedSig
      11. Add $c_1[i]$, $M_{iDS}$ to CachedIDs
   12. Foreach candidate match $c_2$ in $C_{G_{c_2}}$
      13. If (!CachedSig’.contains($c_2[j]$))
         14. Index Entries of $c_2[j]$ as $NB_j$
         15. Foreach Index Entry $e'$ in $NB_j$
            16. If ($e'$.Dis $\geq$ $Ceil\left(\frac{d_c}{2}\right) - d_c$ & $e'$.Dis $<$ 0)
                17. If ($M'_{iDS}$.contains ($e$.Sig))
                   18. Add $e'$.Sig to $L'_{sig}$, $M'_{iDS}[e'$.Sig].add($e'$.IDs)
                19. Else
                   20. Add $e'$.Sig to $L'_{sig}$, Add $e'$.Sig, $e'$.IDs to $M'_{iDS}$
      21. Add $c_2[j]$, $L'_{sig}$ to CachedSig’
      22. Add $c_2[j]$, $M'_{iDS}$ to CachedIDs’
   23. Foreach candidate match $c_1$ in $G_{c_1}$
   24. Foreach candidate match $c_2$ in $C_{G_{c_2}}$
   25. Intersect CachedSig[$c_1[i]$] with CachedSig’[$c_2[j]$] as $\psi_{i,j}$
Foreach Sigbyte $s$ in $\psi_{i,j}$

Equal Scan CachedIDs $[c1[i]][s]$ with CachedIDs’ $[c2[j]][s]$ as $\psi_s$

If ($\psi_s \neq \phi$)

Join $c1$, $c2$ as $c'$, Add $c'$ to $C_{G_{c'}}$

Return $C_{G_{c'}}$

In order to address this limitation of Merge Check algorithm for connectivity checks between components, we propose a new algorithm, called Cached Check algorithm. This algorithm provides an efficient way to store the intermediate index scan results for each distinct node which can be utilized in accelerating the connectivity check results. Cached Check algorithm (Algorithm 2) begins with scanning index entries for all matched nodes for $q_i$ and $q_j$. For each distinct matched node, it scans the index entries by checking the distance constraint and caches the list of significant byte and all IDs associated with each significant byte (line 1-22). For connectivity checks between each pair of the matched nodes, it first locates the cached results in memory and then it finds all the intersected significant bytes (line 25). Finally, it perform the equal scan of all the neighbor IDs for each intersected significant byte (line 26-27). If it identifies any intersect neighbor, it joins the two candidate matches as the candidate matches for the $G_{c'}$.

Compared with Merge Check algorithm which skips index entries with smaller significant byte, Cached Check algorithm needs scan all neighbor IDs for each distinct matched node. On the other hand, it avoids scanning the index entries of the same node multiple times by utilizing more memory to store intermediate scan results if both components have many matched candidates. As the neighbor IDs required by the connectivity checks are cached in memory using the Vertical Bit-Parallel layout, the
equality scan for neighbor IDs are processed efficiently. For the queries with connection edges between components, most of the components are small (as the query templates become disconnected smaller pieces) with a larger number of matched candidates. Thus, these queries benefit more from the Cached Check algorithm due to savings of repetitive index scans for the same node.
7.4 Query Framework based on Neighborhood Byte Index

In general, the query framework design for neighborhood byte index remains the same as RDF-h for NI index. The query framework begins with separating connection edges from query templates, which may result in a template with several connected components. Then, it uses the ByteMap Index to find matching candidates for each query node by checking the partial keywords. Neighborhood check process is then selectively utilized to prune the set of candidates for each query node. Each connected component is decomposed into a set of D-trees. All matching candidates for each decomposed D-tree are generated, and then joined together to get the matching results for each component.

Neighborhood check process is slightly changed from the original algorithm as it utilizes the Vertical Bit-Parallel layout to check neighbor IDs for each given keyword. As the keywords are matched to consecutive ID intervals (keywords are specified as prefixes of the RDF labels), the most important operation for neighborhood check is to perform predicate range scans of the neighbor IDs. The process of the neighborhood check remains the same for neighborhood byte index. It first identifies the value pairs as \{Distance, Count\} for each partial keyword associated with each query node. By checking the partial keywords one by one, the neighborhood check process then scans the NB index to check whether all partial keywords are uniquely contained by the neighbors of the candidate node. The neighbors of the candidate node contain the partial keyword if the number of neighbors which match the partial keyword exceeds the Count within the Distance. We demonstrate the neighborhood check process in Figure 40 which checks the candidate node 0x04ED77 (<http://purl.uniprot.org/diseases /426>) for query template in Figure 39. To pass the neighborhood check, the candidate node 0x04ED77 should have one neighbor which
matches partial keyword <…/keywords/*> within 1 hop, and have one neighbor which matches keyword "Ectodermal dysplasia" within 2 hops. To check for the partial keywords, first the significant bytes of all neighbors of node 0x04ED77 are checked, and the significant bytes intersecting with the ID intervals of the partial keywords are identified. As we shown in Figure 40, the neighbor check achieves an early termination for partial keyword <…/keywords/*> as it identifies one match in the first byte scan of the neighbor IDs.

7.5 Handling Dataset Updates

Another benefit of utilizing neighborhood byte index is to achieve incremental updates which avoids rebuilding the neighborhood index for the whole RDF dataset when small amount of data are changed. Most real datasets require routine updates to include the most current results and offer downloads for the updated subsets directly. Updates for an RDF dataset are often represented as a set of triple statements. The advantage of using the four byte IDs in neighborhood byte index is the last byte can be utilized to handle newly inserted nodes without influence the most of the index entries.

The incremental update algorithm is shown in Algorithm 3. It begins with scanning the set of newly inputted triples and identifying the list of newly inserted nodes without ID assignment (line 1-2). For each node in the list, it locates the two consecutive node IDs in ByteMap index where the new node label should be inserted in between in lexicographic order. The new ID is assigned as the median of the consecutive node IDs to avoid frequent reordering of the existing ByteMap index (line 3-4). As an example in Figure 37, <…/keywords/38> should be inserted between nodes 0x050D2 100 and 0x050D2200. As
the flag bit is set to 1 to indicate the utilization of the last byte, the last byte of assigned ID is 0x80+0x40=0xC0. If many nodes should be inserted between the same consecutive node IDs, it performs equal insertions, i.e. the ID intervals among the assigned IDs are equal. The next step is to build neighborhood indexes for each node in the set of new triples. If the node is assigned with new ID, the new index entries are inserted into the NB index. Otherwise, the new index entries overwrite the old indexes for the nodes in the original graph (line 6-17). The last step is the backward propagation for the neighbor nodes indexed in the new entries to complete the incremental updates. For each neighbor node, the backward propagation checks whether the indexed node ID should be updated in the index entries for the neighbor node due to the updates (line 18-21).

Algorithm 3: IncreUpdate ( T,BID, NB)

Input: Set of Updated Triples T, ByteID index BID, NB Index NB

Output: Updated ByteID index BID and NB Index NB

1 Foreach triple t in T
2 If BID not contain the label l in t
3 Locate consecutive IDs i₁ and i₂ in BID where l₁ (label of i₁) < l and l₂ (label of i₂) > l in lexicographic order
4 Assign the median value of i₁ and i₂ as i to l and insert into BID
5 Set of new index entries as E
6 Foreach triple t in T
7 Find the ID iₛ for t_subject and ID iₒ for t_object
8 If E not contain index entries for ID iₛ
9 Build index for iₛ as Eₛ
10 If NB not contain index entries for ID iₛ
11 Insert Eₛ to NB, Add Eₛ to E
12 Else Overwrite Eₛ in NB, Add Eₛ to E
If $E$ not contain index entries for ID $i_o$

Build index for $i_o$ as $E_o$

If $NB$ not contain index entries for ID $i_o$

Insert $E_o$ to $NB$, Add $E_o$ to $E$

Else Overwrite $E_o$ in $NB$, Add $E_o$ to $E$

Foreach index entry $e$ in $E$

Foreach neighbor ID $i$ in $e.IDs$

If $NB_i$ (Index Entries of ID $i$) not contain $e.ID$

Update $NB_i$ with $e.ID$

One challenge of updates is to resolve the conflicted IDs for new inserted nodes, i.e. there is no IDs can be assigned for the newly inserted node between the specific consecutive node IDs. In that case, a reshuffle of the node IDs happens. The reshuffle reorders the last byte of all node IDs which have the same first three bytes as the consecutive node IDs. As the last byte can be utilized to handle 128 insertions between each consecutive node IDs in the original dataset, the possibility of reshuffling is very small since the updates are relatively small as compared with the whole dataset. When the accumulated update size reaches certain percentage of the complete dataset (e.g. 20%), the NB index are rebuilt from scratch.

7.6 Experimental Results

We evaluate the optimizations of RDF-h for two aspects: (i) index space comparison with NI index for different datasets; (ii) query evaluation analysis for the biomedical queries. The optimized RDF-h framework is implemented with Visual C# 2010 and SQL Server 2008. All experiments are performed on a 2.93GHZ Intel(R) Xeon X5570 machine with 48GB ram running Windows Server 2008 R2.
7.6.1 Index Space Comparison
As we explain in Section 7.2.1, neighborhood byte index is more space efficient compared with neighborhood integer index. The significant byte which is utilized to group the neighbor IDs into one index entry saves a lot of space if the indexed node has many
neighbors sharing the same significant byte. Indicated by the flag bit, all neighbor IDs indexed are represented as the byte arrays with fixed length (two bytes if the flag bit is 0 and three bytes if the flag bit is 1). The fixed length byte IDs allow the query framework to separate each neighbor ID from a long byte array without using any delimiter. The index space comparison results are shown in Figure 41 (for Uniprot dataset) and Figure 42 (for LUBM dataset). One can observe that the IDMap index and the ByteMap index are equivalent in space since the IDs in the IDMap index are 32 bit integer while the IDs in the ByteMap index are 4 byte bit arrays. Four versions of the neighborhood indexes are compared: 3 hop neighborhood index, 2 hop neighborhood index, 1 hop neighborhood index and vertex cover neighborhood index (the vertex cover neighborhood index uses 2 hop neighbors for nodes in the vertex cover set, and 1 hop neighbors for all other nodes). As shown in the results, the neighborhood byte index achieves higher compression rate if more hops of neighbors are indexed as more neighbors with the shared significant byte are indexed in each entry. Overall, the neighborhood byte index use less than half of the index space compared with the neighborhood integer index for both datasets in all versions.

7.6.2 Query Evaluation Analysis

We use the same ten biomedical queries for Uniprot dataset (Chapter 6, Section 6.4) that we used in experimental results using NI index to demonstrate the efficiency of utilizing neighborhood byte index in RDF-h. All these biomedical queries are proposed as small components connected by connection edges. The query performance highly depends on the efficiency of the connectivity check algorithm. To compare different connectivity check algorithms, three query plans are evaluated:
Figure 45. Memory Cost for Cached Check

Figure 46. Query Comparison (in Percentage)
1. Integer check algorithm: connectivity check algorithm for neighborhood integer index which provides similar cache mechanism as cached check algorithm;

2. Merge check algorithm: all connection edges are processed based on merge check process without using any cache;

3. Merge check + Cached check algorithm: connection edges inside component are processed by merge check algorithm and connection edges between components are process by cached check algorithm).

The query run time results are shown in Figure 43 and Figure 44. For single protein patterns (Q1 to Q4 in Figure 43), most of the queries contain only one connection edge between small protein component (e.g. protein with function annotation, protein with Reactome pathway…) and a specific disease node “Breast cancer”. The number of connectivity checks is the number of candidates of the small protein component since the disease node “Breast cancer” has exactly one match. As a result, most of queries are not benefit from cached check algorithm which caches the index scan results compared with merge check algorithm which skips the index entries without matched significant byte. In the opposite, most of the multiple protein queries (Q5, Q6, Q9 and Q10) which contain many connection edges achieve performance improvement by using the cached check algorithm as shown in Figure 44. For Q5, the query performance with merge check algorithm performs even worse compared with integer check algorithm based on neighborhood integer index. Another important aspect of the integer check and cached check algorithm is memory cost for caching the index scan results. As the cached results are stored temporally during the query evaluation, the memory cost is not a critical issue. However, it becomes a big memory overhead if many queries are executed at the same
time. The memory cost for cached check algorithm of the ten biomedical queries is shown in Figure 45 (memory cost for integer check algorithm has similar results). One can observe that queries (e.g. Q5, Q6) benefit significantly from the cached check algorithm require more memory to store the intermediate results. Repetitively scanning these results becomes time consuming for the merge check algorithm. As a result, the effectiveness of caching the intermediate results leads to much more efficient query evaluation for the merge check + cached check algorithm.

As we can observe from the query evaluation results, merge check algorithm which enables early skip by checking the significant byte in neighborhood byte index is efficient to handle connection edges in most queries. However, selectively utilizing cached check algorithm to process connection edge between components which both have a large number of candidates is beneficial. To achieve better performance, the optimized algorithm is proposed as hybrid check algorithm which chooses to utilize cached check algorithm if all following conditions are true: 1) the connection edge is between components; 2) both components has more than $\phi_1$ candidates and 3) the product of the sizes of components’ candidate sets are larger than $\phi_2$. Thresholds $\phi_1$ and $\phi_2$ can be set using a sample set of user input queries similar as the parameter tuning process in RDF-h. For Uniprot dataset, we simply set $\phi_1$ as 2 and $\phi_2$ as 1,000 for the ten biomedical queries as no user input queries are available. The query comparison results between integer check algorithm, merge check algorithm, merge check + cached check algorithm and hybrid check algorithm are shown in Figure 46. For most of the queries (except Q2, Q4 and Q5), the query performance of hybrid check algorithm can save more than 50% of the run time compared with integer check algorithm. Hybrid check algorithm provides the best performance
among the four algorithms for all queries.

7.7 Conclusions and Comments

ByteMap and neighborhood byte indexes are proposed for RDF-h query framework in this project. Compared with the IDMap and neighborhood integer indexes, ByteMap and neighborhood byte indexes have three major advantages: 1). higher index space efficiency; 2). incremental update for routine dataset updates; 3). optimized connectivity check algorithms for better query performance. By using the Vertical Bit-Parallel storage, the neighborhood check and connectivity check processes based on neighborhood byte index achieve early termination and skip of unmatched index entries similar as the BitWeaving and ByteSlice techniques. For datasets with millions of nodes, the last byte of the byte IDs are utilized to handle newly inserted nodes where the original nodes are indexed with the first three bytes of the byte IDs to save index space. The utilization of flag bit and the byte IDs provides an easy way to separate neighbor IDs from a long byte array stored in each index entry. Merge check algorithm and cached check algorithm are proposed to maximize the benefit of neighborhood byte index for processing connection edges. Based on the experimental results, an improved connectivity check algorithm, hybrid check algorithm is proposed which selectively utilizes cached check algorithm by evaluating the complexity of processing a connection edge between components. In experiments, we focus on demonstrating: 1). higher space efficiency of neighborhood byte index compared with neighborhood integer index; 2). better query performance for the biomedical queries using the new proposed connectivity check algorithms. For index space efficiency, neighborhood byte index can save more than 80% of index space for Uniprot dataset and save more than
60% of index space for LUBM dataset. For query performance evaluation, hybrid check algorithm outperforms integer check algorithm for all queries and saves more than 50% of query run time for most of the queries.
Chapter 8 Future Works

Overall, RDF-h provides an effective framework for querying graph structured RDF data. Though we propose many query optimization strategies to make RDF-h framework more efficient, there are still some issues open for future research. Here, we focus on two main topics which can highlight the future directions for RDF-h: 1) handling relational based functionalities, e.g. aggregations on numerical values; 2) distributed memory cloud to handle billion edge graphs.

8.1 SPARQL Hybrid

Graph template matching is an effective way to query RDF data. However, a big limitation of graph template matching is that it is not natural to support relational based functionalities, such as group by, filter, aggregation and comparison among variables. These relational based functionalities are commonly utilized to provide statistical analysis in decision making process; and thus they are important for real applications. In order to handle both relational based and graph based (e.g. connection edges) functionalities efficiently, different types of indexes are required. Neighborhood signature index is good for evaluating connection edges while not useful for aggregations. Indexing permutations of three dimensions of RDF triples can support aggregation but not effective in answering connection edges. By analyzing the characteristics of a given RDF dataset, a hybrid storage and index plan can be proposed to benefit both relational based and graph based functionalities.
Figure 47. SparQL Hybrid Query Process

Figure 48. RDF Schema Graph
There are two common types of nodes in a RDF graph: resource (URI) nodes (blank nodes are treated same as resource nodes in our work) and literal nodes. This also leads to different types of edges: relationship edges (between two resource nodes) and attribute edges (between resource node and literal node). One important rule for RDF data is that Subject can only be resource (URI) nodes which means there is no out-going edges for all literal nodes. Thus, building graph based index for attribute edges is not benefit as there is no path across the literal nodes. On the other hand, relational based functionalities are closely related with literal nodes and attribute edges. Moreover, most RDF datasets come with predefined type systems described by the predicate “rdf:type”. As analyzed in [QO-2, DK+11], resources of the same type often share the same set of attribute predicates, and show similar relationship patterns. RDF datasets can also come with RDF-Schema documents and ontologies which describe classes, class hierarchy and important predicates. By using all these type information, the hybrid storage and index plan can be further optimized.

SPARQL hybrid proposes a hybrid storage and index plan by utilizing schema graphs. Schema graph is an abstract visualization of the given RDF graph considering its schema documents which can be utilized as a standard way to define the characteristics of an RDF dataset. The schema graph describes the set of classes and predicates used in the given RDF dataset as shown in Figure 47. Resources are classified into different classes which are often identified by predicate rdf:type and may be described by URI references or predicates. By analyzing the schema graph and the features of the given RDF dataset, SPARQL hybrid stores attribute edges in the extended property tables and stores relationship edges as graphs. Indexes to accelerate partial matching of keywords and
aggregations can be built for the extended property tables while neighborhood signature index can be built to accelerate graph isomorphism matching and efficient evaluation of connection edges. This hybrid plan can provide both index space efficiency and query performance improvements.

The challenge of this work is there is no systematical way to build a schema graph for any given dataset which can be utilized in query processing. In other words, the process of building the schema graph should adapt to a large granularity of prior knowledge (related documents) that a RDF dataset may associate with. As the worst case, a RDF dataset may come without any documents and the only prior knowledge is that there are two types of nodes and edges based W3C standard definitions. As an opposite, if a RDF dataset is transferred from a relational database with strict schema, we may have accurate documents which can provide us complete underlying data structure. Thus, making best effort to utilize different granularity of prior knowledge to improve query performance is the major challenge.

8.2 Distributed Framework

The requirement of handling RDF graphs with billion triples necessitates the distributed design of the RDF-h query framework. Current representatives of distributed query infrastructures include: Hadoop and Map Reduce [RS11, RS10, DS10, DQ+10, JO+10], Dryad [IB+07], trinity memory cloud [SW+13], Pregel [MA+10, SW14, TB+14, HD+14] and Spark [ZC+10, AX+15].

Hadoop stores data using HDFS. HDFS partitions data into large horizontal partitions and each partitioned block receives a unique ID. The HDFS blocks get distributed and
replicated over the cluster. Query evaluation of Hadoop is achieved by Map Reduce. Map Reduce assigns a mapper at every data node which has data to be processed. Each mapper reads one of the HDFS blocks and breaks that HDFS blocks into records. The output intermediate results is collected on the local disks. Intermediate results are redistributed over different data nodes and reducer process the intermediate results to generate final results.

Dryad is another distributed query infrastructure providing similar functionalities as Hadoop. The benefit of Dryad is that it can support SQL queries directly. SQL queries are automatically transferred as Communication Graphs. Dryad also provides runtime graph refinement. For communication graph with aggregation (i.e. common downstream node), it adds internal local execution nodes for a large single output which can benefit from the local storage or servers in the same rack. Compared with Hadoop, Dryad is more flexible to specify the workload and data communication.

Trinity memory cloud is targeted at in memory graph manipulation. The in memory graph is stored as Cell structs which contain attributes of each node and in-links and out-links of its neighbors. The Cell structs can communicate through message passing, both synchronized and asychronized. Cell structs in main memory are stored as memory chunks of key value pairs represented in binary blob. Global address table is built which contains the chunks for a specific key.

Pregel is a vertex-centric programming model based on BSP (Bulk Synchronous Parallel) architecture. A vertex in Pregel commonly contains: (i) attributes: id and value, (ii) its outgoing edges and their attributes: destination vertex id and edge value, and (iii) a logical inbox in which it receives messages from other vertices. Pregel computation is
executed through synchronous iterations of asynchronous computation. During sequence of super steps, each vertex executes a user-defined function. Vertices work independently and asynchronously in each super step, while at the end of a super step, a global synchronization occurs among vertices in order to exchange messages.

Spark extends Hadoop with a new abstraction called resilient distributed dataset. Resilient distributed dataset is immutable collections partitioned across cluster. It is created by transforming data in stable storage using data flow operators, e.g. map, filter, group by. The resilient distributed dataset can be rebuilt if a partition is lost. Current version of Spark extends its functionality to support SQL queries. It proposes an optimizer framework, called Catalyst for manipulating trees of relational operators.

Though there are many distributed query infrastructures available, most of them are not natural to support graph template matching. Since graph template matching is a NPC problem, the distributed version of RDF-h focuses on in memory graph manipulation similar as trinity memory cloud.

The basic memory cell in distributed RDF-h contains the attributes and neighborhood byte index of each node. The nodes in the RDF graph is partitioned into different groups and each group of nodes is assigned to a server. In order to minimize the communication cost among different servers during query evaluation, it’s important to answer most of the queries locally, i.e. most of the queries results are contained in a single server. Since the neighborhood byte index for each node contains 3 hops neighbor information, the queries with size 3 or smaller can be answered locally. For queries with larger size, the matching result may across different servers if the candidates for a query node is a border node (a border node is a node connecting any node in other servers). To minimize the possibility
that a query result containing edges across different servers, it is benefit to find the k-way graph partitioning with minimal edge cut. A popular approach to find the k-way graph partitioning with minimal edge cut is multilevel graph partitioning. Multilevel graph partitioning begins with iteratively shrink the graph by contracting edges. The shrinking process is repeated until the number of remaining vertices is small enough to perform an expensive initial partitioning algorithm. At last, the contracted edges are released gradually and a local optimization search algorithm is performed to improve the edge cut size. Here, we use METIS [LK+13] (serial graph partitioning and fill-reducing matrix ordering) to handle the graph partitioning task. Once the graph is partitioned into different groups of nodes, each group is assigned with a server ID and the memory cells of the nodes in the group is loaded into main memory of the assigned server. Other optimizations can be utilized to handle queries across different servers to reduce the communication cost.

Storing the graph in a distributed way is only the first step. The main challenge of this work is to achieve parallel processing of graph template matching. Validating the results and improving the graph partitioning strategy are also important issue to check in future work.
Chapter 9 Conclusions

In this thesis, we propose a flexible framework for querying graph structured RDF data, called RDF-h, which selectively uses the signature-based pruning based on the characteristics of RDF datasets and query templates. RDF-h utilizes parameters we defined to describe the dataset characteristics and the complexity of query templates to assess the effectiveness of signature-based pruning techniques in query evaluation. The query templates utilized in RDF-h also allow flexible connection edges, and partially entered keywords. Compared with the state of art graph template matching techniques, such as Spath [4] and STWIG [9], RDF-h shows improved query performance since it combines their advantages and chooses additional signature-based pruning if it is evaluated as effective. With the parameter tuning process, RDF-h can automatically capture frequent user query patterns and be adjusted to maximize the benefits of the signature-based pruning. This makes RDF-h more effective in domain specific real applications. Based on our RDF dataset characteristics analysis, we can also identify datasets with respect to the expected level of performance gains from signature-based pruning.

By querying the integrated biomedical RDF knowledge base UniprotKB, RDF-h is demonstrated as a promising tool to express sophisticated questions, and discover hidden connections among biological or biomedical entities. The availability of ever increasing
amounts of RDF data in various fields of biomedical applications from public and private
sources, adds even more to the potential of querying integrated RDF data sets using RDF-
h for new insights and discovering knowledge about hidden associations among biological
and medical entities.

Additional optimizations using the new ByteMap and neighborhood byte indexes are
also proposed for RDF-h query framework. The optimizations achieve three important
goals highlighted in querying biomedical data: 1) support incremental update for routine
dataset updates; 2) reduce the index space requirement; and 3) improved connectivity
check algorithms for better query performance. The optimized RDF-h framework utilizes
the Vertical Bit-Parallel in memory storage plan which achieves early termination and skip
of unmatched index entries for both neighborhood check and connectivity check processes.
The experimental results shows that neighborhood byte index can save more than 80% of
index space for Uniprot dataset and save more than 60% of index space for LUBM dataset
compared with the original neighborhood integer index. The optimized hybrid check
algorithm also outperforms the integer check algorithm for all queries on Uniprot dataset
and saves more than 50% of query run time for most of the queries.
Bibliography


[QY+09] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Yufei Tao. Querying Communities in Relational Databases. ICDE ’09, 2009.


[ZL09] S. Zhang, S. Li, and J. Yang. GADDI: Distance Index based Subgraph


2014.


[SW13] Bin Shao, Haixun Wang, and Yatao Li, Trinity: A Distributed Graph Engine on a Memory Cloud, SIGMOD 2013.


