Graph Grammar Based RDF Benchmarking System

by

Jiaping Xu
B.Sc., Zhengzhou University, 2005
M.Eng., Tongji University, 2008

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Jiaping Xu, 2017
University of Victoria

All rights reserved. This project may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Graph Grammar Based RDF Benchmarking System

by

Jiaping Xu
B.Sc., Zhengzhou University, 2005
M.Eng., Tongji University, 2008

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Daniel Hoffman, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Daniel Hoffman, Co-Supervisor
(Department of Computer Science)

ABSTRACT

In this thesis, we propose a graph grammar based RDF (Resource Description Framework) benchmarking system. We generate the graphs based on two types of graph grammars, Generalized Node Replacement Recursive Graph Grammar (GNRR) and Generalized Edge Replacement Recursive Graph Grammar (GERR). We provide algorithms that generate the RDF database and queries from GNRR or GERR graph grammars of a certain domain. We perform two experiments to show the efficiency of our benchmarking system and provide benchmark results against Jena TDB 3.2 and Sesame 2.9.0.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vi

List of Figures vii

Acknowledgements viii

Dedication ix

1 Introduction 1
- 1.1 Overview 1
- 1.2 Contribution 2
- 1.3 Related Works 2
  - 1.3.1 Grammar-based Testing 2
  - 1.3.2 RDF Benchmarks 2
- 1.4 Thesis Organization 3

2 Background 4
- 2.1 RDF Databases 4
- 2.2 SPARQL Queries 5

3 Graph Grammars for RDF Graph and Query Generation 8
- 3.1 Graph Grammar Definition 8
  - 3.1.1 Generalized Node Replacement Recursive Graph Grammar (GNRR) 9
  - 3.1.2 Generalized Edge Replacement Recursive Graph Grammar (GERR) 11

4 Graph Grammar-based RDF Data Store Benchmarking System 13
4.1 Data and Query Generator 13
4.1.1 Generator based on GNRR Graph Grammar 14
4.1.2 Generator based on GERR Graph Grammar 15
4.2 Query Performance Comparison 19

5 Experiments 20
5.1 System Environment 20
5.2 GNRR Graph Grammar based Benchmark for Twitter 20
5.2.1 Graph Grammar 21
5.2.2 Generated RDF Graphs and Queries 24
5.2.3 Benchmark Results 25
5.3 GERR Graph Grammar based Benchmark for G-tetrad 29
5.3.1 Graph Grammar 29
5.3.2 Generated RDF Graphs and Queries 30
5.3.3 Benchmark Results 31
5.4 Experiment Conclusion 33

6 Conclusions and Future Work 34

Bibliography 36

List of Tables

Table 5.1 Benchmark Datasets and Queries from Twitter's GNRR Graph Grammars 25
Table 5.2 The load times to Jena and Sesame - Twitter's Datasets 26
Table 5.3 Benchmark results for Twitter Dataset 1 26
Table 5.4 Benchmark results for Twitter Dataset 2 27
Table 5.5 Benchmark results for Twitter Dataset 3 27
Table 5.6 Benchmark results for Twitter Dataset 4 28
Table 5.7 Benchmark results for Twitter Dataset 5 28
Table 5.8 Benchmark Datasets and Queries from G-tetrad's GERR Graph Grammars 31
Table 5.9 The load times to Jena and Sesame - G-tetrad's Datasets 31
Table 5.10 Benchmark results for G-tetrad Dataset 1 32
Table 5.11 Benchmark results for G-tetrad Dataset 2 32
Table 5.12 Benchmark results for G-tetrad Dataset 3 32
Table 5.13 Benchmark results for G-tetrad Dataset 4 33
Table 5.14 Benchmark results for G-tetrad Dataset 5 33

List of Figures

Figure 2.1 A simple RDF statement 5
Figure 2.2 A set of RDF triples in N-Triples format 5
Figure 2.3 A simple RDF query 6
Figure 2.4 An RDF query with multiple matches 7

Figure 3.1 (a). An example of a GNRR graph grammar (b). A graph generation process 10
Figure 3.2 (a). An example of a GERR graph grammar (b). A graph generation process 12

Figure 4.1 A graph grammar representation in an input file 15
Figure 4.2 A graph grammar representation in a input file 17

Figure 5.1 The graph model of Twitter 21
Figure 5.2 The decomposed graph model of Twitter 22
Figure 5.3 The graphical representation of Twitter's GNRR Graph Grammar 23
Figure 5.4 The text Representation of Twitter's GNRR Graph Grammar . 24
Figure 5.5 A set of queries generated from Twitter's GNRR graph grammars 25
Figure 5.6 The graphical representation of G-tetrad's GERR Graph Grammars 29
Figure 5.7 The graphical representation of G-tetrad's GERR Graph Grammars 30

ACKNOWLEDGEMENTS

I would like to expression my gratitude and appreciation to my supervisors Dr. Alex Thomo and Dr. Daniel Hoffman for their guidance and support. Without their valuable assistance, this work would not have been completed.
I would like to express my profound gratitude to my parents and to my husband Siyuan Xiang for providing me with continuous support and encouragement throughout my years of study. This accomplishment would not have been possible without them.

DEDICATION

I would like to dedicate this work to my husband and my parents.

Chapter 1

Introduction

1.1 Overview

The Resource Description Framework (RDF) is a framework recommended by the World Wide Web Consortium (W3C) for representing information on the web. RDF data consists of triples of the form subject-predicate-object, e.g. (Shakespeare, wrote, Hamlet) and (Shakespeare, lived-in, London). These triples define a graph where the nodes are the subjects and objects of the RDF triples e.g. (Shakespeare, Hamlet, London) and the edges are the predicates e.g. (wrote, lived-in). Subjects, objects, and predicates are also called *resources*.

An RDF store (or triple store) provides a mechanism for storing and querying data in RDF format. A number of RDF data stores have been actively developed since the early 2000s, e.g., Jena, Sesame, Virtuoso, BigOWLIM, and RDF Gateway. These RDF data stores provide SPARQL as a query language on RDF databases. There have been a lot of SPARQL benchmark methodologies proposed to evaluate the performance of SPARQL queries. However, most of them are limited to the structure of a certain area, e.g., e-Commerce, university, and social networking; therefore they are not variable enough to supply a measure of the RDF data store performance for a different domain. In this project, we propose a benchmarking system that generates RDF databases and queries based on graph grammars.

1.2 Contribution

In this project, we propose and implement a benchmarking system for RDF data stores. We generate graphs using Node Replacement and Edge Replacement graph grammars, namely, Generalized Node Replacement Recursive Graph Grammar (GNRR) and Generalized Edge Replacement Recursive Graph Grammar (GERR) which are generalized versions of NRR and ERR graph grammars[18]. We represent the graph grammar in a text file in which each production has a probability to be applied when choosing a production rule with a specific left side symbol. A set of parameters is defined to limit the size of the graph and number of the queries. We use GNRR and GERR in two different experiments to demonstrate that our system can generate RDF databases and queries efficiently. The benchmark results against two well-known RDF stores, Jena and Sesame, are shown at the end of each experiment.

1.3 Related Works

1.3.1 Grammar-based Testing

Grammar-based testing has been proposed since 1970 for automated generation of test cases [14]. In [21], an enhanced context free grammar is proposed to not only generate test cases, but also generate simple test program according to the description of the test. Work [27] describes lava, a domain specific language for specifying production grammars, which are used to generated test suites for the Java virtual machine. [16] presents YouGen, a new grammar-based test generation tool supporting the semantic based tags. Formal grammars are used in these works based on the fact that the test cases in their contexts follow the formation rule of a formal language.

To the best of our knowledge there is still no work that uses grammars for automated testing and benchmarking of RDF data stores.

1.3.2 RDF Benchmarks

SPARQL is the query language for RDF data stores. A lot of effort has focused on building benchmarks to evaluate the current existing RDF data stores. Most of the existing benchmark systems are provided for a specific domain. For example, LUBM [12] consists of a university domain ontology, BSBM [4] is designed upon an e-Commerce use case, and LDBC [9] focuses on the field of social networks.

1.4 Thesis Organization

In Chapter 2, we give the background knowledge of RDF database and SPARQL query. We describe the graph grammar used in our approach in Chapter 3. Chapter 4 introduces the design and implementation of our benchmark system. Chapter 5 shows the experiment details and the benchmark results. We conclude the thesis and discuss the future work in Chapter 6.

Chapter 2

Background

2.1 RDF Databases

The Resource Description Framework (RDF) is a recommendation of the World Wide Web Consortium (W3C), which can be used to represent data in the open web or enterprises.

All the things in the world are referred to as resources in RDF. A resource can be a web resource, such as a web page or a HTML element; a resource can also be an object that is not directly accessible via the Web, such as a printed book. Properties (or predicates) are used to describe relationships between resources or the attributes of the resources. RDF uses Uniform Resource Identifier (URI) to identify resources and properties.

The basic model in RDF is a subject-predicate-object triple which is called an RDF statement. Subjects, objects, and predicates are represented by resources. Figure 2.1 shows graphically a simple triple. The subject is `http://www.w3.org/Home/Lassila`, the predicate is “Creator”, and the object is “Ora Lassila”.

The subject can be a URI or blank node; the predicate has to be a URI; the object can be a URI, blank node, or a literal.

A collection of RDF triples represents an RDF graph, which can be saved in an RDF data store (in the form a set of triples). There are several common serialization formats to represent RDF triples, namely, XML, Notation 3 (or N3), Turtle and N-Triples. In this thesis, we use N-Triples as the representation format of triples. Figure 2.2 shows an N-Triples format RDF graph.

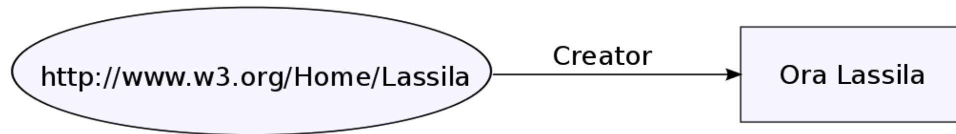

Figure 2.1: A simple RDF statement

```
PREFIX dbpedias:<http://dbpedia.org/resource/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema:<http://schema.org/>

dbpedias: Aristotle        rdf: type   schema: Person .
dbpedias: Alcover          rdf: type   schema: Place .
dbpedias: Alcover          rdf: type   schema: City .
dbpedias: Pablo_Casals     rdf: type   schema: Person .
dbpedias: Pablo_Casals     rdf: type   schema: MusicGroup .
```

Figure 2.2: A set of RDF triples in N-Triples format

2.2 SPARQL Queries

SPARQL is the query language for RDF databases. Most of the RDF data stores implement a SPARQL query engine. SPARQL contains capabilities for querying graph patterns using conjunctions and disjunctions. The results of SPARQL queries can be sets of tuples or RDF graphs [2].

Most SPARQL queries contain a set of triple patterns called "basic graph patterns". Triple patterns are a set of RDF triples in which each of the subject, predicate and object may be a variable. Triple patterns are combined together using conjunctions or disjunctions. When the basic graph patterns match a subgraph of the RDF database, the matched RDF triple elements that can substitute for the variables in the patterns are the query results.

Variables in the triple patterns are prefixed by "?". The variables whose value bindings we would like to output are listed in the select clause of the query, very much like in SQL for relational databases.

The example in Figure 2.3 shows a SPARQL query to find the Birthday of William Shakespeare from the given data graph. The query consists of two parts: the SELECT clause identifies the variables to appear in the query results, and the WHERE

clause provides the basic graph pattern to match against the data graph. The basic graph pattern in this example consists of a single triple pattern with a single variable (?dateOfBirth) in the object position. This query has one solution as shown in Figure 2.3.

Data:

```
<http://dbpedia.org/resource/William_Shakespeare>  <http://dbpedia.org/property/dateOfBirth>  "April 1564"
```

Query:

```
select ?dateOfBirth where
{<http://dbpedia.org/resource/William_Shakespeare>  <http://dbpedia.org/property/dateOfBirth>  ?dateOfBirth }
```

Query Results:

dateOfBirth
"April 1564"

Figure 2.3: A simple RDF query

Figure 2.4 shows an example of multiple matches, in which the result of a query is a solution sequence, corresponding to the ways in which the query's graph pattern matches the data. In the dataset we have five RDF triples describing some people with regard to their name and email address. The query is a conjunction of two basic graph patterns with variables. The result of this query is given in the following table in the figure.

There may be zero, one or multiple solutions to a query. Each solution gives one way in which the selected variables can be bound to RDF terms so that the query pattern matches the data. The result set gives all the possible solutions. In this example, the two subsets of the data provided two matches.

Data:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x  foaf:name  ?name .
    ?x  foaf:mbox  ?mbox. }
```

Query Results:

name	mbox
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

Figure 2.4: An RDF query with multiple matches

Chapter 3

Graph Grammars for RDF Graph and Query Generation

In this chapter, we introduce the graph grammar which we will use to generate RDF databases and queries. We use a Node Replacement Graph Grammar and an Edge Replacement Graph Grammar that generalize the Node Replacement Recursive Graph Grammar (NRR) and Edge Replacement Recursive Graph Grammar (ERR) proposed in [18].

3.1 Graph Grammar Definition

Just like a string grammar can define a string language, a graph grammar provides a mechanism to define a set of graphs. A graph grammar $G = (S, P)$ consists of a starting graph S and a finite set of productions rules P. A production rule is a triple (M, D, E), where M is a mother graph, and D is a daughter graph, and E is some embedding mechanism which contains a finite number of connection instructions. Such a production rule can be applied to a "host" graph H if there is an occurrence of M in H. The production rule is applied by removing this occurrence of M from H, replacing it by (an isomorphic copy of) D, and finally using the embedding mechanism E to attach D to the remainder H^{-} of H [24]. The graph language $L(G)$ is the set of graphs generated from graph grammar G.

Different types of graph grammars can be distinguished based on how the embedding mechanism is defined and applied.

3.1.1 Generalized Node Replacement Recursive Graph Grammar (GNRR)

The GNRR graph grammar is more generalized than the NRR graph grammar defined in [18]. Namely, in the GNRR graph grammar, the right side of a production does not have to contain the left side non-terminal symbol in the same production, which is required in the definition of NRR graph grammar [18]. A NRR graph grammar is a special case of GNRR graph grammars. The following gives the definition of a graph and a GNRR graph grammar.

A labeled graph G is a 6-tuple, $G = (V, E, \mu, \nu, L)$, where
V - is the set of nodes,
$E \subseteq V \times V$ - is the set of edges,
μ: V $\rightarrow$ L - is a function assigning labels to the nodes,
ν: E $\rightarrow$ L - is a function assigning labels to the edges,
L - is a set of labels on nodes and edges.

A generalized node replacement recursive graph grammar is a tuple $Gr = (\Sigma,\Delta,\Gamma,P)$, where
Σ - is an alphabet of node labels,
Δ - is an alphabet of terminal node labels, $\Delta \subseteq \Sigma$,
Γ - is an alphabet of edge labels, which are all terminals,
P - is a finite set of productions of the form (d, D, C), where $d \in \Sigma$ - Δ, which is the left side of the production. D is a labeled graph over Σ and Γ. There should be at least one production with S on the left side, which denotes the starting graph.

C is an embedding mechanism with a set of connection instructions, $C \subseteq V \times V$, where V is the set of nodes of some production rule. A connection instruction in C is of the form (v_i-v_j), where v_i is a node with the label d_r in an instance of D and v_j is a node in the D_r graph of some production rule P_r of the form (d_r, D_r, C_r). Such an instruction says that if there is an instance of D in some host graph, we can replace v_i in the instance of D by graph D_r and the incident edges of v_i will become incident edges of v_j. All the existing edges incident to v_j remain unchanged.

There are two types of productions. If there is at least one non-terminal node in the D graph of P, the graph D is called a non-terminal graph, and the production P is called a recursive production; if all the node labels in the graph D are terminal, the graph D is called a terminal graph, and the production P is called a non-recursive production. The set of connection instructions of a non-recursive production is null.

Figure 3.1-(a) shows a GNRR graph grammar and 3.1-(b) shows how a graph is derived from it. In this graph grammar, $\Sigma = \{a, b, c, d, S, N\}$, which is a set of node labels; $\Delta = \{a, b, c, d\}$, which is a set of terminal node labels; $\Gamma = \{x, y, z\}$, which is a set of edge labels. In the first production, S indicates the starting one-node graph on the left side. The production has connection instructions $(2-1)$ and $(4-2)$. The second production rule with N on the left side, which has a terminal graph on the right side.

The graph generation process in 3.1-(b) is as follows. 1) After applying the first production rule, the graph G_1 is the first graph obtained. Now we can replace either one of the two non-terminal node labeled N by using the production rules for N. 2) The left side N node in G_1 has ID 2 in the instance of D_1, so we use the connection instruction $(2-1)$ and the second production rule for N. The node 1 in D_2 thus replaces the left side N node in G_1 and obtain graph G_2. 3) The right side N node in G_2 has ID 4 in the instance of D_1, so we use the connection instruction $(4-2)$ and the second production rule for N. The node 2 in D_2 replaces node 4 in G_2 and graph G_3 is obtained.

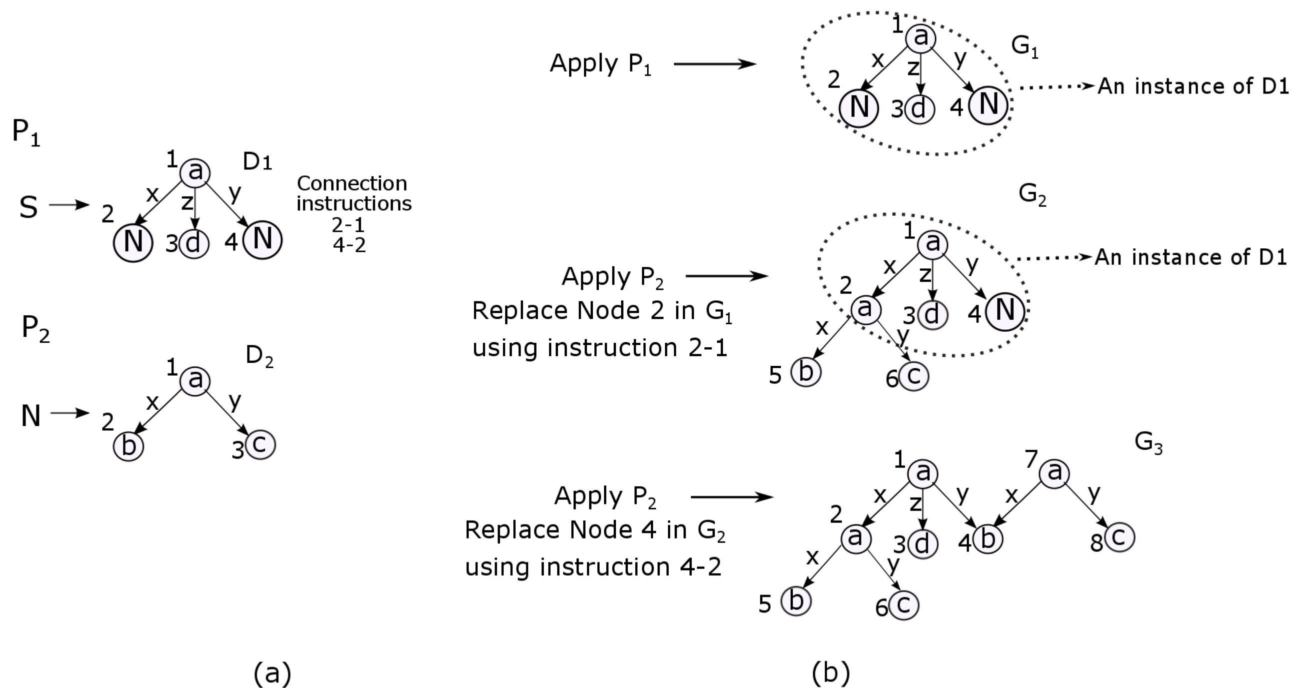

Figure 3.1: (a). An example of a GNRR graph grammar (b). A graph generation process

3.1.2 Generalized Edge Replacement Recursive Graph Grammar (GERR)

A generalized edge replacement recursive graph grammar[18] is a tuple $Gr = (\Sigma,\Gamma,\Omega,P)$, where

Σ - is an alphabet of node labels, which are all terminals,

Γ - is an alphabet of edge labels,

Ω - is an alphabet of terminal edge labels, $\Omega \subseteq \Sigma$,

P - is a finite set of productions of the form (d, D, C), where $d \in \Gamma$ - Ω, which is the left side of the production, G is a graph over Σ and Γ. There should be at least one production with S on the left side, which denotes the starting graph. C is an embedding mechanism with a set of connection instructions, $C \subseteq (V \times V; V \times V)$, where V is the set of nodes of some production rules. A connection instruction in C is of the form (v_i-v_j, v_k-v_l), where there is an edge with the label d_r between node v_i and v_k in an instance of D, and there is an edge between v_j and v_l in the D_r graph of some production rule $P_r(d_r, D_r, C_r)$. Such a instruction says that if there is an instance of D in some host graph, we can replace v_i, v_k in the instance of D with the graph D_r. All incident edges of v_i will become incident to v_j, all incident edges of v_k will become incident to v_l, and all the edges incident to v_j and v_l remain unchanged.

The edge between (v_i, v_k) or (v_j, v_l) is a virtual edge when there is no real edge between them. If the edge between (v_j, v_l) is a virtual edge and the edge label is terminal, the edge and the edge label will be removed between them after the production is applied, and the node v_j and v_l still remains.

There are two types of productions. If there is at least one non-terminal edge label e in the D graph of P, the graph D is called a non-terminal graph, and the production P is called a recursive production; if all the edge labels in the graph D are terminal, the graph D is called a terminal graph, and the production P is called a non-recursive production. The set of connection instructions of a non-recursive production is null.

An example in Figure 3.2 shows a GERR graph grammar and how a graph is derived from it.

The graph generation process in 3.2-(b) is as follows. 1) After applying the first production rule, the generated graph G_1 is the first graph with a non-terminal edge (3, 1) labeled E. 2) The edge labeled E in G_1 is edge (3, 1) is the instance of D_1. To replace it in G_1, we use the second production rule for E and the connection instruction (3-3, 1-2) defined in the first production. The node 3 in G_1 is replaced by

the node 3 in D_2 and the node 1 in G_1 is replaced by the node 2 in D_2. The node IDs are updated after replacement, thus obtaining graph G_2. The ID with a dashed circle indicates the original node ID in the daughter graph D_2.

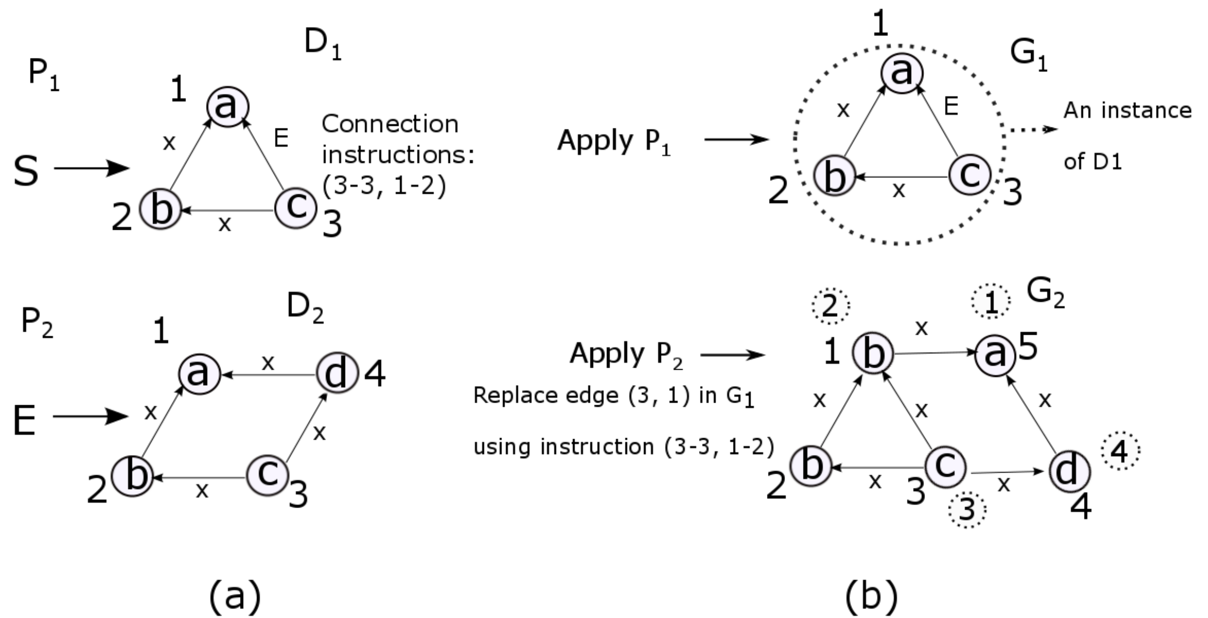

Figure 3.2: (a). An example of a GERR graph grammar (b). A graph generation process

Chapter 4

Graph Grammar-based RDF Data Store Benchmarking System

In this chapter, we will introduce the design of our benchmarking system, especially how we generate the database and queries using GNRR and GERR graph grammars.

4.1 Data and Query Generator

Since an RDF database is a collection of triples, which represents a labeled, directed graph, and a query on RDF data store is also constructed based on a basic graph pattern, we can use graph grammars to generate RDF databases and queries automatically. The databases and queries will be used in benchmarking the RDF data store's performance.

Graph grammars differ in the way how connection instructions are defined. The connection instructions of GNRR and GERR graph grammars emphasize the node overlapping, i.e., GNRR overlap the host graph and daughter graph on one node; GERR overlap the host graph and daughter graph on two nodes. Intuitively, it's like we have a number of separate RDF triples about one or two resources, and we want the RDF triples overlapping on the same resource to form an RDF graph.

Our generator can generate graphs and queries based on either GNRR graph grammar or GERR graph grammar defined in Chapter 3. The choice of the graph grammar should depend on the graph structure of the specific domain.

The productions of the graph grammar will be defined in an input text file, which is the input of the generator.

We have four parameters for a generator.

- *grammar*: the identifier of the used grammar rules
- *max_Iterations*: maximum number of iterations for the recursion. An RDF graph in N-Triple format will be generated until *max_Iterations* is reached.
- *query_Iteration_percent*: Given that the current iteration is *level*, only when *level/max_Iterations* is less than or equal to *query_Iteration_percent*, generate a query from the current derived graph.
- *max_Iteration_nonterminal*: the limit to use non-terminal production only, when the current iteration *level* is less than or equal to *max_Iteration_nonterminal*, only choose non-terminal productions for replacement, and the non-terminal production will be chosen based on the probabilities of all corresponding non-terminal productions; otherwise choose the production based on the probabilities of all productions.

max_Iterations is generally used to control the size of the generated graph; *query_Iteration_percent* determines when a query is generated during the recursion. The basic graph pattern of the generated query will be a subgraph of the generated RDF graph. This will make sure the query has potential value bindings in the RDF graph. *query_Iteration_percent* also limits the size of the query graph. *max_Iteration_nonterminal* is used to direct the generator to choose a non-terminal production only at the early recursion phase, which will avoid the graph rewriting being terminated by completing the replacement of all non-terminal node labels.

The set of parameters applies to both of GNRR and GERR graph grammars. The embedding rules of the two graph grammars are different during the recursion. The following subsections will introduce the generator algorithms for the two graph grammars.

The generators for GNRR and GERR are implemented using Java 8 with 1663 lines of code in total.

4.1.1 Generator based on GNRR Graph Grammar

Figure 4.1 shows the representation of GNRR graph grammar in Figure 3.1, which is represented in a input file. This graph grammar contains three productions. We

assign a probability to each of the productions. Our generator will choose the production based on the probabilities. Each production starts with a non-terminal symbol followed by a probability and the graph definition. The symbol serves as the left side of the production, while the graph definition serves as the right side of the production. We will always use a single capital letter to represent the left side, which is also called a non-terminal node label when it appears on the right side. The sum of the probabilities for a non-terminal symbol should be 1. In the graph definition, a node definition begins with v. For example, "v 1 a" defines a node with ID 1 and label "a"; and an edge definition begins with e, "e 1 2 x" defines an edge labeled x from node 1 to node 2. If the label of a node is non-terminal, it will be followed by a number, which indicates the node that will replace it. For example, "v 2 N 1" says there is a node 2 with label N and a connection instruction (2, 1) in the production.

```
S         N
1         1
v 1 a     v 1 a
v 2 N 1   v 2 b
v 3 d     v 3 c
v 4 N 2   e 1 2 x
e 1 2 x   e 1 3 y
e 1 3 z
e 1 4 y
```

Figure 4.1: A graph grammar representation in an input file

In an RDF graph, each node should have a unique label, but the node labels in generated graphs may be duplicated. We generate the node label by concatenating the original node label with the node ID. Algorithm 1 shows the generating process for GNRR graph grammar.

4.1.2 Generator based on GERR Graph Grammar

Figure 4.2 shows the representation of GERR graph grammar in Figure 3.2, which is represented in an input file. This graph grammar contains two productions. A

Algorithm 1 Graph and Query Generator Algorithm (for GNRR)

```
procedure GENERATOR(grammar, max_Iterations, query_Iteration_percent,
max_Iteration_nonterminal)
    read productions from grammar
    select a production whose left side is S according to the probability of each
    production. The right side of the selected production is starting_graph
    GNRR_EXPAND_GRAPH(grammar, starting_graph, 0)
end procedure
procedure GNRR_EXPAND_GRAPH(grammar, graph, level)
    if level == max_Iterations then
        EXPORT_GRAPH(graph) return
    else if level/max_Iterations <= query_Iteration_percent then
        EXPORT_QUERIES(graph)
    end if
    add all nodes and edges in graph to new_graph
    for all nodes v ∈ graph do
        if the label of v is non-terminal then
            if level <= max_Iteration_nonterminal then
                select a production whose left side is the label of v and right side
            contains non-terminal labels according to the probability of the
            productions
            else if level + 1 == max_Iterations then
                select a production whose left side is the label of v and right side
            contains only terminal labels according to the probability of the
            productions
            else
                select a production whose left side is the label of v according to the
            probability of the productions
            end if
            the right side of the selected production is replacing_graph
            update the label of v in new_graph with the label of replacing node v_i which is
            indicated in grammar input file
            add all the nodes and edges of replacing_graph to new_graph
            make every node incident to v_i in replacing_graph incident to v in new_graph
            delete node v_i in new_graph
        end if
    end for
    EXPAND_GRAPH(grammar, new_graph, level++)
end procedure
procedure EXPORT_GRAPH(graph)
    generate graph in N-Triple format
end procedure
procedure EXPORT_QUERIES(graph, level)
    Map level to a type of SPARQL query with specific features
    Generate SPARQL query with basic graph pattern graph
end procedure
```

probability is assigned to each of the productions. The generator will choose the production based on the probabilities. Each production starts with a symbol followed by a probability and a graph definition. As the representation of the GNRR graph grammar, the symbol serves as the left side of the production, while the graph definition serves as the right side of the production. We will always use a single capital letter to represent the left side, which is also called a non-terminal node label when it appears on the right side. The sum of the probabilities for a non-terminal symbol should be 1. In the graph definition, the node definition begins with v. For example, "v 3 c" defines a node with ID 3 and label "c" and an edge definition beginning with e, "e 3 2 x" defines an edge labeled x from node 3 to node 2. If the label of an edge is non-terminal, it will be followed by a pair of numbers, which indicates an edge that will replace it. For example, "e 3 1 E 3 2" says there is an edge (3,1) with label E and a connection instruction (3-1, 3-2) in the production.

We reserve the keyword "virtual" for the label of a virtual edge. The virtual edges will be removed from the graph before generating an RDF graph or query.

```
S            E
1            1
v 1 a        v 1 a
v 2 b        v 2 b
v 3 c        v 3 c
e 2 1 x      v 4 d
e 3 1 E 3 2  e 2 1 x
e 3 2 x      e 3 4 x
             e 3 2 x
             e 4 1 x
```

Figure 4.2: A graph grammar representation in a input file

Algorithm 2 shows the generating process based on GERR graph grammar.

Algorithm 2 Graph and Query Generator Algorithm (for GERR)

1: **procedure** GENERATOR(*grammar*, *max_Iterations*, *query_Iteration_percent*, *max_Iteration_nonterminal*)
2: read productions from *grammar*
3: select a production whose left side is S according to the probability of each production. The right side of the selected production is *starting_graph*
4: GERR_EXPAND_GRAPH(*grammar*, *starting_graph*, 0)
5: **end procedure**
6: **procedure** GERR_EXPAND_GRAPH(*grammar*, *graph*, *level*)
7: **if** *level* == *max_Iterations* **then**
8: EXPORT_GRAPH(*graph*) **return**
9: **else if** *level*/*max_Iterations* <= *query_Iteration_percent* **then**
10: EXPORT_QUERIES(*graph*)
11: **end if**
12: add all nodes and edges in *graph* to *new_graph*
13: **for all** edges $e \in graph$ **do**
14: **if** the label of e is non-terminal **then**
15: **if** *level* <= *max_Iteration_nonterminal* **then**
16: select a production whose left side is the label of e and right side contains non-terminal labels according to the probability of the productions
17: **else if** *level* + 1 == *max_Iterations* **then**
18: select a production whose left side is the label of e and right side contains only terminal labels according to the probability of the productions
19: **else**
20: select a production whose left side is the label of e according to the probability of the productions
21: **end if**
22: the right side of the selected production is the *replacing_graph* in *new_graph*,
23: update the labels of nodes v_i, v_j for edge e with the labels of replacing nodes v'_i, v'_j which are indicated in grammar input file
24: add all nodes and edges in *replacing_graph* to *new_graph*
25: in *new_graph*, make every node incident to v'_i incident to v_i
26: in *new_graph*, make every node incident to v'_j incident to v_j
27: delete node v'_i and v'_j in *new_graph*
28: **for all** edges $e \in new_graph$ **do**
29: **if** the label of e is "virtual" **then**
30: delete e in *new_graph*
31: **end if**
32: **end for**
33: **end if**
34: **end for**
35: GERR_EXPAND_GRAPH(*grammar*, *new_graph*, *level* + +)
36: **end procedure**
37: **procedure** EXPORT_GRAPH(*graph*)
38: generate graph in N-Triple format
39: **end procedure**
40: **procedure** EXPORT_QUERIES(*graph*, *level*)
41: Map *level* to a type of SPARQL query with specific features
42: Generate SPARQL query with basic graph pattern *graph*
43: **end procedure**

4.2 Query Performance Comparison

We generate RDF graph in N-triples format as defined in Chapter 2. We build query generators that can generate 4 different types of queries: order by, group by, optional pattern, and property path.

The RDF graph and queries are used as the RDF benchmark dataset. We provide the function for comparing the performance of different RDF data stores. We compare two different RDF data store products in our system: Sesame 3.2 and Jena 2.9.0. The native store is used on both of the products. The timeout will be set to 1800 seconds for each of the queries. We will show the load time of the RDF database and query execution time for each query. The details of the benchmark experiments and results will be described in Chapter 5.

Chapter 5

Experiments

In this chapter, we will show the experiments in which we compare the performance of two RDF data stores. We perform two different experiments: the first one is using the GNRR graph grammar built for Twitter; the second is using the GERR graph grammar for the chemistry structure of G-tetrad. The choice of the graph grammar depends on the graph structure of the represented domain.

5.1 System Environment

The experiments will be running on a machine with 16 GB memory,Inter(R) Core(TM) i7-6700 CPU, 3.40GHz processor and the following softwares.

- Operating System: Window 10 Pro
- Java Version: 1.8.0_131
- Jena: 3.2
- Sesame: 2.9.0

5.2 GNRR Graph Grammar based Benchmark for Twitter

The graph database is a natural choice for a social network application like Twitter. In this experiment, we will build the GNRR graph grammar based on the use cases

of Twitter and use the inferred graph grammar rules as the input of the dataset generator.

5.2.1 Graph Grammar

Figure 5.1 is built on Twitter's basic use cases, which also shows the relationships between these entities, i.e., user, tweet, hashtag, link.

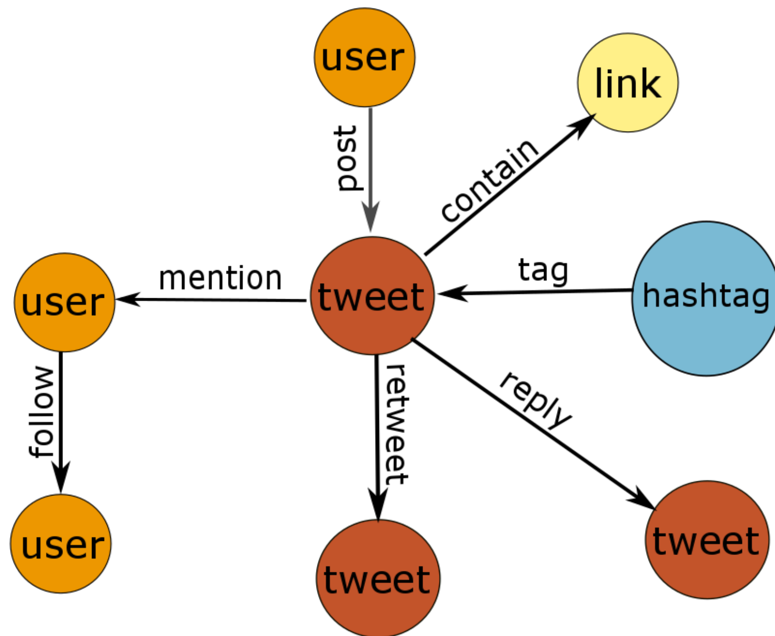

Figure 5.1: The graph model of Twitter

Considering each of the nodes to be a unique resource, the graph can be expanded by applying the use cases in this graph repetitively. For example, we have a single triple "user1-post-tweet1", when the user posts another tweet and follows another user, we get another 2 triple: "user1-post-tweet2", "use1-follow-user2". The 3 triples form a graph about user1. To generate the graph automatically in a graph grammar, the replacement on a node will happen while applying the graph grammar rules. Intuitively, we decompose the graph model into the basic graph patterns as Figure 5.2 shows. We add literal nodes in rectangles to describe the text value of each entity, i.e., *content* for tweet, *tagText* for hashtag and *userAccount* for user.

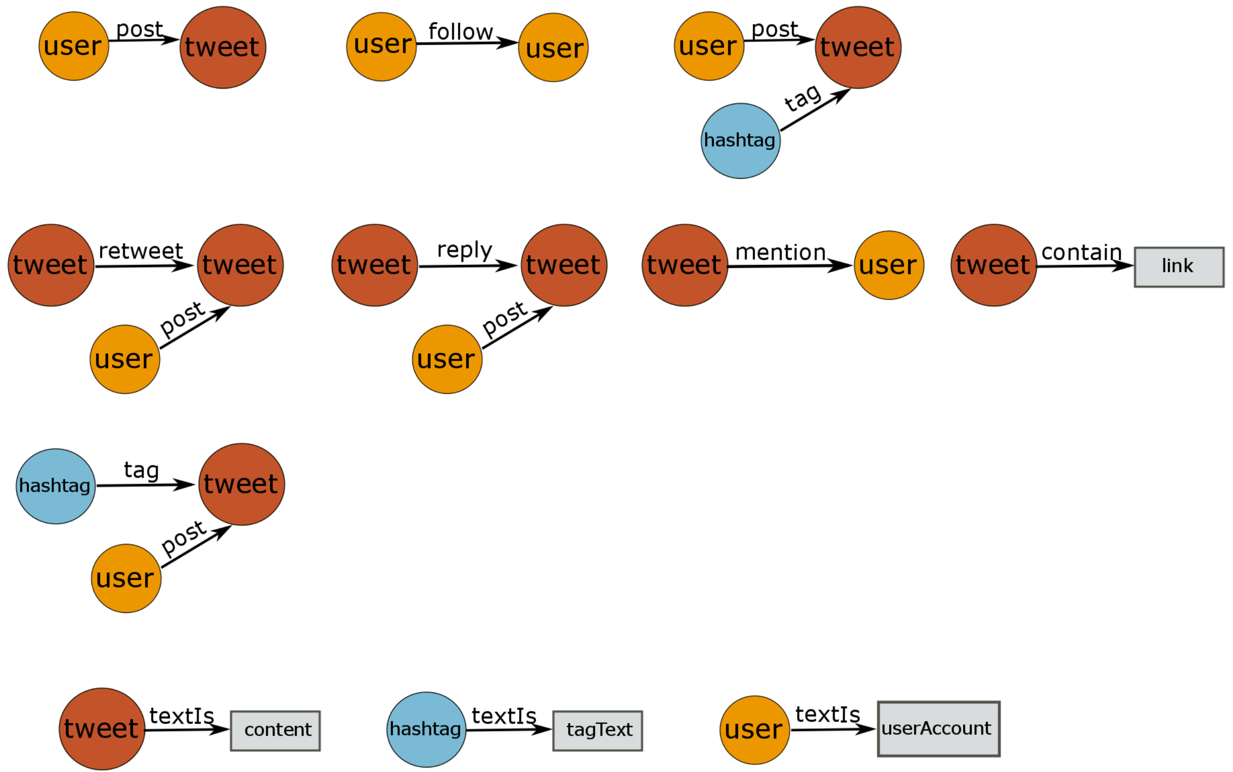

Figure 5.2: The decomposed graph model of Twitter

In a GNRR graph grammar, the replaced node should be a non-terminal node. We use the original node label as a terminal node label, namely, *user*, *tweet*, *hashtag*; use a single capital letter as a non-terminal node label, i.e. U is non-terminal node label of *user*, T is non-terminal node label of *tweet* and H is non-terminal node label of *hashtag*. The node replacement happens on each entity node during the graph rewriting. Thus, we bring the non-terminal node to the left side to construct a production. The triple "user-post-tweet" is used as a starting graph, which has the symbol S on the left side of the production. To make it easy to manage, each replacing node is numbered as 1 in the production graph. After specifying the connection instruction for each of the non-terminal graphs, the final GNRR graph grammar for Twitter is shown in Figure 5.3. The text format input file to the generator is shown in Figure 5.4. The probability should be assigned to each of the production, based on the estimated portion of the final generated graph covered by the production graph.

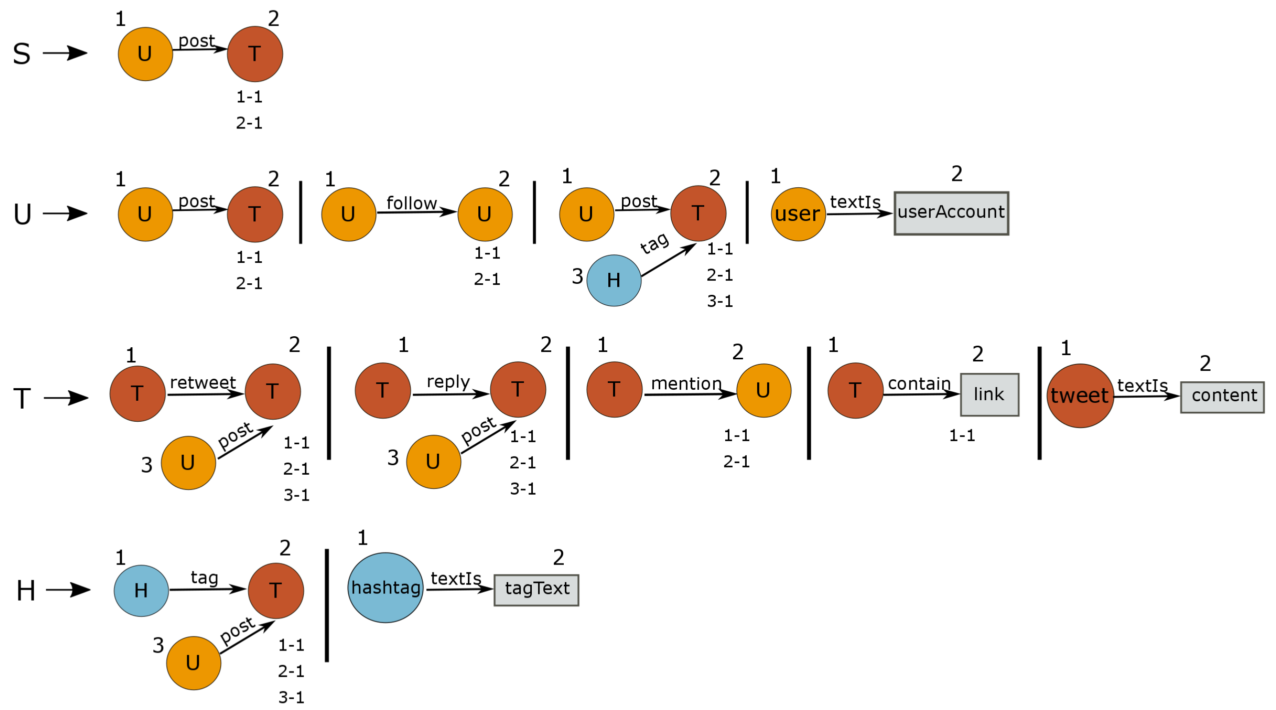

Figure 5.3: The graphical representation of Twitter's GNRR Graph Grammar

```
S
1
v 1 U 1
v 2 T 1
e 1 2 post
```

```
U
0.3
v 1 U 1
v 2 T 1
e 1 2 post
```

```
U
0.3
v 1 U 1
v 2 U 1
e 1 2 follow
```

```
U
0.1
v 1 U 1
v 2 T 1
v 3 H 1
e 1 2 post
e 3 2 tag
```

```
U
0.3
v 1 user
v 2 userAccount
e 1 2 textis
```

```
T
0.2
v 1 T 1
v 2 T 1
v 3 U 1
e 1 2 retweet
e 3 2 post
```

```
T
0.1
v 1 T 1
v 2 T 1
v 3 U 1
e 1 2 reply
e 3 2 post
```

```
T
0.1
v 1 T 1
v 2 U 1
e 1 2 mention
```

```
T
0.3
v 1 T 1
v 2 link
e 1 2 contain
```

```
T
0.3
v 1 tweet
v 2 content
e 1 2 textis
```

```
H
0.3
v 1 H 1
v 2 T 1
v 3 U 1
e 1 2 tag
e 3 2 post
```

```
H
0.7
v 1 hashtag
v 2 tagtext
e 1 2 textis
```

Figure 5.4: The text Representation of Twitter's GNRR Graph Grammar

5.2.2 Generated RDF Graphs and Queries

We ran the benchmark data generator on the graph grammar defined in Figure 5.4. Apart from the graph grammar, the other 3 parameters $max_Iterations$, $query_Iteration_percent$, $max_Iteration_nonterminal$(defined in Algorithm 1) are also passed to the generator. We generated 5 RDF databases and 11-14 queries for each of the databases. Details about the generated benchmark data set are summarized in Table 5.1. The tables shows our generator can generate 1071155-triple RDF database and 11 queries on the database in 12726 seconds. The Figure 5.5 shows a sample of the generated queries.

Table 5.1: Benchmark Datasets and Queries from Twitter's GNRR Graph Grammars

max_Iterations	*query_Iteration_ percent*	*max_Iteration_ nonterminal*	Num of Triples	Num of Queries	Time Cost(s)
10	0.5	8	1603	12	0.28
20	0.3	8	12527	14	2.18
30	0.2	12	488619	12	2631.51
40	0.3	10	377361	12	1546.14
50	0.25	10	1071155	11	12726.10

```
select ?U1 ?T5 where {?U1 <e:post> ?T2.OPTIONAL { ?U1 <e:follow> ?U3}
?T2 <e:reply> ?T4.?U1 <e:post> ?T5.OPTIONAL { ?T2 <e:retweet> ?T6}
?U3 <e:post> ?T7.?T4 <e:reply> ?T8.}

select ?U1 (str(count (*)) as ?count) where {?U1 <e:post> ?T2.?U1 <e:follow> ?U3.
?T2 <e:reply> ?T4.?U1 <e:post> ?T5.?T2 <e:retweet> ?T6.?U3 <e:post> ?T7.
?T4 <e:reply> ?T8.} group by ?U1

select ?U1 ?T2 ?T6where {?U1 <e:post> ?T2.?U1 <e:follow>* <v:U3>.
?T2 <e:reply> ?T4.?U1 <e:post> ?T5.?T2 <e:retweet>* <v:T6>.?U3 <e:post> ?T7.
?T4 <e:reply> ?T8.?U1 <e:post> ?T9.?T2 <e:retweet> ?T10.?U3 <e:post> ?T11.
?T4 <e:mention> ?U12.?T5 <e:reply> ?T13.?T6 <e:reply> ?T14.?T7 <e:reply> ?T15.
?T8 <e:retweet> ?T16.}

select ?U1 ?T4 ?T7 where {?U1 <e:post> ?T2.?U1 <e:follow> ?U3.?T2 <e:reply> ?T4.
?U1 <e:post> ?T5.?T2 <e:retweet> ?T6.?U3 <e:post> ?T7.?T4 <e:reply> ?T8.
?U1 <e:post> ?T9.?T2 <e:retweet> ?T10.?U3 <e:post> ?T11.?T4 <e:mention> ?U12.
?T5 <e:reply> ?T13.?T6 <e:reply> ?T14.?T7 <e:reply> ?T15.?T8 <e:retweet> ?T16.
?U1 <e:follow> ?U17.?T2 <e:mention> ?U18.?U3 <e:follow> ?U19.?T4 <e:reply> ?T20.
?T5 <e:retweet> ?T21.?T6 <e:mention> ?U22.?T7 <e:reply> ?T23.?T8 <e:mention> ?U24.
?T9 <e:retweet> ?T25.?T10 <e:retweet> ?T26.?T11 <e:retweet> ?T27.?U12 <e:post> ?T28.
?T13 <e:reply> ?T29.?T14 <e:retweet> ?T30.?T15 <e:retweet> ?T31.?T16 <e:retweet> ?T32.}
order by ?U1 limit 100
```

Figure 5.5: A set of queries generated from Twitter's GNRR graph grammars

5.2.3 Benchmark Results

We ran the benchmark data against two RDF data stores, Jena 3.2 and Sesame 2.9.0. Table 5.2 shows the load time of each data set to Jena and Sesame. The benchmark

results are shown in Table 5.3-5.7. The number of the query also indicate the number of iteration when the query is generated. We set the timeout for query execution time to 1800 seconds for both Jena and Sesame.

	Data set 1	Data set 2	Data set 3	Data set 4	Data set 5
#Triples	1603	12527	488619	377361	1071155
Jena(s)	0.44	1.11	12.17	9.55	23.28
Sesame(s)	0.36	2.43	55.80	40.98	126.60

Table 5.2: The load times to Jena and Sesame - Twitter's Datasets

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	1603	0.22	0.13
2	group by	1603	0.05	0.03
3	group by	1603	0.02	0.03
4	group by	1603	0.01	0.02
5	optional pattern	1603	0.08	0.15
6	group by	1603	0.04	0.03
7	property path	1603	0.04	0.02
8	group by	1603	9.19	13.21
9-12		1603	timeout	timeout

Table 5.3: Benchmark results for Twitter Dataset 1

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	12527	0.17	0.13
2	group by	12527	0.07	0.07
3	group by	12527	0.04	0.10
4	group by	12527	0.03	0.07
5	optional pattern	12527	0.36	90.59
6	group by	12527	0.04	5.36
7	property path	12527	0.07	0.01
8	group by	12527	8.39	1005.51
9-14		12527	timeout	timeout

Table 5.4: Benchmark results for Twitter Dataset 2

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	488619	0.33	1.01
2	group by	488619	0.64	0.96
3	group by	488619	0.82	timeout
4	group by	488619	0.62	1.21
5	optional pattern	488619	1.41	timeout
6	group by	488619	1.08	timeout
7	property path	488619	1.45	0.89
9-12		488619	timeout	timeout

Table 5.5: Benchmark results for Twitter Dataset 3

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	377361	0.3	0.91
2	group by	377361	0.42	0.83
3	group by	377361	0.55	1.87
4	group by	377361	0.45	1.64
5	optional pattern	377361	1.18	timeout
6	group by	377361	0.91	2.26
7	property path	377361	1.01	1.98
8	group by	377361	66.54	92.28
9-12		377361	timeout	timeout

Table 5.6: Benchmark results for Twitter Dataset 4

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	1071155	0.79	2.24
2	group by	1071155	0.74	2.11
3	group by	1071155	0.87	4.78
4	group by	1071155	0.77	4.27
5	optional pattern	1071155	timeout	timeout
6	group by	1071155	1.90	5.60
7	property path	1071155	4.79	0.01
8	group by	1071155	139.77	250.20
9-11		1071155	timeout	timeout

Table 5.7: Benchmark results for Twitter Dataset 5

5.3 GERR Graph Grammar based Benchmark for G-tetrad

In chemistry domain, the graphs of the chemistry structures often have the repetitive graph pattern overlapped on two nodes, for example, the PAHs and DNA. The graph structure and properties of the chemistry structure can be saved in an RDF data store. If the GERR graph grammars can be inferred from the graph structure, our generator can produce the dataset of the chemistry structure for benchmark purpose.

5.3.1 Graph Grammar

In this experiment, we use the graph grammars of G-tetrad inferred in [18]. The graphical graph grammars of G-tetrad are shown in Figure 5.6.

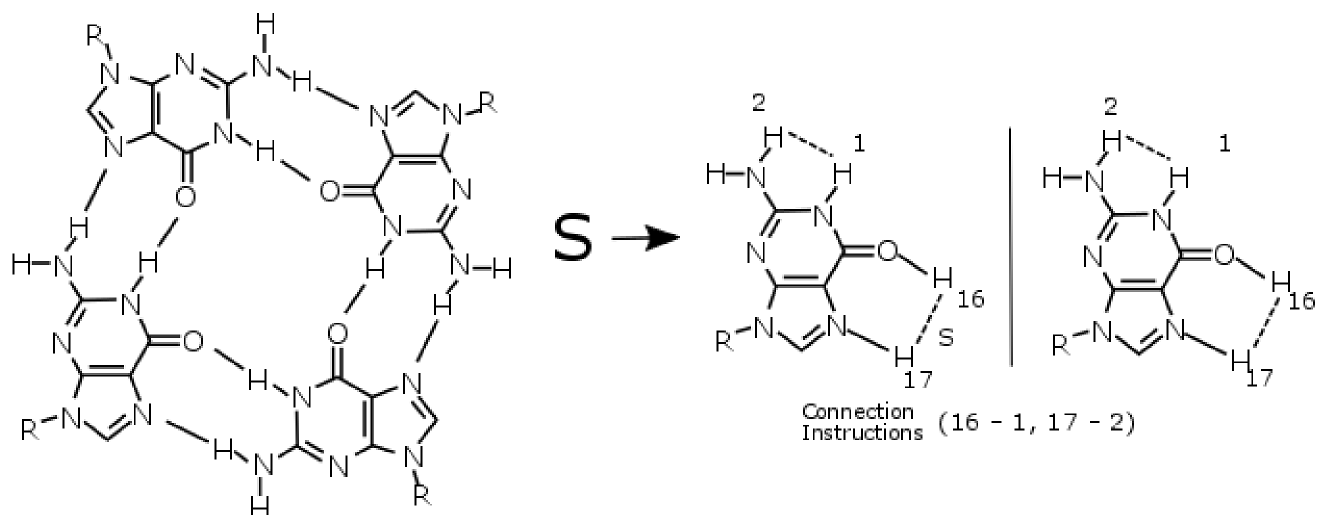

Figure 5.6: The graphical representation of G-tetrad's GERR Graph Grammars

Figure 5.7 is the text format of the graph grammars. We reserve the edge label "virtual" to represent a virtual edge, which will be removed from the graph when it is used to generate the database and query.

```
S
1
v 1 H
v 2 H
v 3 H
v 4 N
v 5 C
v 6 N
v 7 C
v 8 R
v 9 N
v 10 C
v 11 N
v 12 C
v 13 C
v 14 O
v 15 N
v 16 H
v 17 H
e 2 4 sb
e 4 3 sb
e 4 5 sb
e 5 6 db
e 6 7 sb
e 7 12 db
e 12 13 sb
e 13 15 sb
e 15 1 sb
e 15 5 sb
e 7 9 sb
e 9 8 sb
e 9 10 sb
e 10 11 db
e 11 12 sb
e 13 14 db
e 11 17 sb
e 14 16 sb
e 16 17 E 1 2
```

```
E
0.9
v 1 H
v 2 H
v 3 H
v 4 N
v 5 C
v 6 N
v 7 C
v 8 R
v 9 N
v 10 C
v 11 N
v 12 C
v 13 C
v 14 O
v 15 N
v 16 H
v 17 H
e 1 2 virtual
e 2 4 sb
e 4 3 sb
e 4 5 sb
e 5 6 db
e 6 7 sb
e 7 12 db
e 12 13 sb
e 13 15 sb
e 15 1 sb
e 15 5 sb
e 7 9 sb
e 9 8 sb
e 9 10 sb
e 10 11 db
e 11 12 sb
e 13 14 db
e 11 17 sb
e 14 16 sb
e 16 17 E 1 2
```

```
E
0.1
v 1 H
v 2 H
v 3 H
v 4 N
v 5 C
v 6 N
v 7 C
v 8 R
v 9 N
v 10 C
v 11 N
v 12 C
v 13 C
v 14 O
v 15 N
v 16 H
v 17 H
e 1 2 virtual
e 2 4 sb
e 4 3 sb
e 4 5 sb
e 5 6 db
e 6 7 sb
e 7 12 db
e 12 13 sb
e 13 15 sb
e 15 1 sb
e 15 5 sb
e 7 9 sb
e 9 8 sb
e 9 10 sb
e 10 11 db
e 11 12 sb
e 13 14 db
e 11 17 sb
e 14 16 sb
e 16 17 virtual
```

Figure 5.7: The graphical representation of G-tetrad's GERR Graph Grammars

5.3.2 Generated RDF Graphs and Queries

We generated 5 RDF databases and 15-16 queries for each of the databases. The graph grammar defined in Figure 5.7 determines even the smallest query will have 19 triples. When both of the query graph and database graph are very large, it will

consume overwhelming resources in computing the query results. Due to the limit of our experimental resource, we produce 63018-triple RDF database at most in this experiment. Our benchmark results in the experiment will show the queries running on the chemistry RDF database in this experiment will be more challenging to the RDF data stores than the queries in the Twitter experiment. The details of the generated benchmark data is shown in Table 5.8.

Table 5.8: Benchmark Datasets and Queries from G-tetrad's GERR Graph Grammars

max_Iterations	*query_Iteration_percent*	*max_Iteration_nonterminal*	Num of Triples	Num of Queries	Time Cost(s)
100	0.075	100	1818	16	0.25
1000	0.0075	1000	18018	16	0.75
2000	0.00375	2000	36018	16	2.32
3000	0.00025	3000	54018	16	4.85
3500	0.002	3500	63018	15	6.30

5.3.3 Benchmark Results

Table 5.9 shows the load time of each data set to Jena and Sesame. The benchmark results are shown in Table 5.10-5.14. The same as the previous experiment, the number of the query also indicate the number of iteration when the query is generated. we set the timeout for each query to 1800 seconds for both Jena and Sesame.

	Data set 1	Data set 2	Data set 3	Data set 4	Data set 5
#Triples	1818	18018	36018	54018	63018
Jena(s)	0.37	1.08	1.13	1.88	1.47
Sesame(s)	0.36	5.63	8.50	11.38	13.61

Table 5.9: The load times to Jena and Sesame - G-tetrad's Datasets

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	1818	0.11	0.11
2	group by	1818	1.25	32.16
3	group by	1818	233.20	1214.84
4	group by	1818	256.57	852.16
5	optional pattern	1818	1167.12	timeout
6	group by	1818	1357.65	timeout
7	property path	1818	1245.87	timeout
8	group by	1818	1589.45	timeout
9-16		1818	timeout	timeout

Table 5.10: Benchmark results for G-tetrad Dataset 1

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	18018	0.24	1.65
2	group by	18018	3.57	5.57
3	group by	18018	654.16	1604.20
4	group by	18018	1164.54	timeout
5	optional pattern	1654.23	timeout	timeout
6	group by	18018	1357.10	timeout
7-16		18018	timeout	timeout

Table 5.11: Benchmark results for G-tetrad Dataset 2

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	36018	0.89	0.37
2	group by	36018	12.68	12.38
3	group by	36018	578.21	1644.05
4	group by	36018	1365.45	timeout
5	optional pattern	36018	timeout	timeout
6	group by	36018	1524.57	timeout
7-16		36018	timeout	timeout

Table 5.12: Benchmark results for G-tetrad Dataset 3

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	54018	1.35	1.68
2	group by	54018	11.54	62.11
3	group by	54018	235.27	895.36
4	group by	54018	1674.024	timeout
5	optional pattern	54018	timeout	timeout
6	group by	54018	1268.24	timeout
7-16		54018	timeout	timeout

Table 5.13: Benchmark results for G-tetrad Dataset 4

Query	Feature	# Triples	Time Cost-Jena(s)	Time Cost-Sesame(s)
1	order by	63018	3.27	8.57
2	group by	63018	36.24	687.05
3	group by	63018	1166.01	timeout
4	group by	63018	1578.61	timeout
5-15		63018	timeout	timeout

Table 5.14: Benchmark results for G-tetrad Dataset 5

5.4 Experiment Conclusion

The number of the query indicates the number of iteration when the query is generated, so generally the larger the number is, the larger the query's basic graph pattern is. We set the timeout of each query execution time to be 1800 seconds in both experiments, and still notice quit a few query timeouts while the query graph and RDF graph are getting bigger.

From the results, we observe Jena TDB 3.2 is better than Sesame 2.9.0 in overall performance. Some of the query timeouts on Sesame are caused by a known issue[1], by which the iteration of the results will be halted when Sesame deals with some complex query.

Chapter 6

Conclusions and Future Work

We propose an RDF benchmark system, which uses two different types of Graph Grammars, GNRR (Generalized Node Replacement Recursive) and GERR (Generalized Edge Replacement Recursive) Graph Grammars to generate the graph patterns for the represented domain. The graph patterns are then used to generate RDF graph and queries. Comparing to the RDF benchmark approach depending on converting a relational database to an RDF database, our approach can generate RDF database representing the real graph structure of a domain. GNRR and GERR graph grammar can be constructed from either a single triple (e.g. GNRR of twitter) or an existing subgraph pattern (e.g. GERR of G-tetrad), which impose less restriction to the domain structure.

However, the current GNRR and GERR graph grammars only overlap the nodes in daughter graph and host graph on the nodes in connection instructions (two nodes overlap on one node in GNRR; two set of nodes overlap on two nodes respectively in GERR). The graph grammars still have limitations to represent the relationship between a specific node in the daughter graph and a specific node in the host graph. We can try to add extra types of connection instructions, by which the connection instructions of GNRR and GERR are used to expand graphs while the other types of embedding mechanisms are used to build extra relationships between nodes.

Our generator produces graphs and queries with simple node and edge labels, which help to focus on measuring the performance of RDF data stores in terms of subgraph pattern matching. To simulate the RDF database in real life, the RDF graphs and queries should contain namespaces or URI-like strings. To achieve this, the ontology should be considered before constructing the graph grammar and a set of vocabularies need to be defined for the domain.

For future work a first direction to explore is to extend our testing of RDF databases to other domains of graph-structured data, such as spatial, biological, and medical ([3, 7, 13, 17, 22, 25]). Another direction to explore is grammars for generating queries containing property paths, which are given by regular path queries (RPQs) (c.f. [6, 8, 10, 11]). They are very useful to express recursion in SPARQL. Also, we would like to explore the connections between graph grammars and tree languages (c.f. [19, 23, 26]) when restricting our framework to tree-structured data which are very common in many settings. Furthermore, we would like to consider probabilistic graphs ([5, 15, 20]) which give a probability distribution over the existence of each edge in the graph database. As such, we can use these distributions to better guide the application of our grammar rules when generating RDF databases.

Bibliography

[1] Max execution time does not work correctly for expensive queries [sesame 2.8.9]. https://github.com/eclipse/rdf4j/issues/636.

[2] Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query.

[3] Sören Auer, Jens Lehmann, and Sebastian Hellmann. Linkedgeodata: Adding a spatial dimension to the web of data. *The Semantic Web-ISWC 2009*, pages 731–746, 2009.

[4] Christian Bizer and Andreas Schultz. The berlin sparql benchmark, 2009.

[5] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1316–1325. ACM, 2014.

[6] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92, 2003.

[7] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David J Wild. Chem2bio2rdf: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC bioinformatics*, 11(1):255, 2010.

[8] Mariano P Consens, Valeria Fionda, Shahan Khatchadourian, and Giuseppe Pirro. S+ epps: construct and explore bisimulation summaries, plus optimize navigational queries; all on existing sparql systems. *Proceedings of the VLDB Endowment*, 8(12):2028–2031, 2015.

[9] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[10] Gösta Grahne and Alex Thomo. Algebraic rewritings for optimizing regular path queries. *Lecture notes in computer science*, pages 301–315, 2001.

[11] Gösta Grahne and Alex Thomo. Query answering and containment for regular path queries under distortions. In *FoIKS*, volume 4, pages 98–115. Springer, 2004.

[12] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[13] Tatiana Gutiérrez-Bunster, Ulrike Stege, Alex Thomo, and John Taylor. How do biological networks differ from social networks?(an experimental study). In *Advances in Social Networks Analysis and Mining (ASONAM), 2014 IEEE/ACM International Conference on*, pages 744–751. IEEE, 2014.

[14] K.V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[15] Nasrin Hassanlou, Maryam Shoaran, and Alex Thomo. Probabilistic graph summarization. In *International Conference on Web-Age Information Management*, pages 545–556. Springer, 2013.

[16] Daniel Malcolm Hoffman, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang. Grammar-based test generation with yougen. *Software: Practice and Experience*, 41(4):427–447, 2011.

[17] Simon Jupp, James Malone, Jerven Bolleman, Marco Brandizi, Mark Davies, Leyla Garcia, Anna Gaulton, Sebastien Gehant, Camille Laibe, Nicole Redaschi, et al. The ebi rdf platform: linked open data for the life sciences. *Bioinformatics*, 30(9):1338–1339, 2014.

[18] J. Kukluk. Inference of node and edge replacement graph grammars. 2007.

[19] Laks VS Lakshmanan and Alex Thomo. View-based tree-language rewritings for xml. In *FoIKS*, pages 270–289. Springer, 2014.

[20] Xiang Lian and Lei Chen. Efficient query answering in probabilistic rdf graphs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 157–168. ACM, 2011.

[21] P.M. Maurer. Generating test data with enhanced context-free grammars. *Software, IEEE*, 7(4):50–55, 1990.

[22] Zhuo Miao, Dan Stefanescu, and Alex Thomo. Grid-aware evaluation of regular path queries on spatial networks. In *Advanced Information Networking and Applications, 2007. AINA'07. 21st International Conference on*, pages 158–165. IEEE, 2007.

[23] Frank Neven. Automata theory for xml researchers. *ACM Sigmod Record*, 31(3):39–46, 2002.

[24] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1.

[25] Elham Sedghi, Jens H Weber, Alex Thomo, Maximilian Bibok, and Andrew MW Penn. Mining clinical text for stroke prediction. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 4(1):16, 2015.

[26] Maryam Shoaran and Alex Thomo. Evolving schemas for streaming xml. *Theoretical Computer Science*, 412(35):4545–4557, 2011.

[27] E.G. Sirer and B.N. Bershad. Using production grammars in software testing. In *ACM SIGPLAN Notices*, volume 35, pages 1–13. ACM, 1999.