ABSTRACT

Title of dissertation: SCALABLE ONTOLOGY SYSTEMS

Octavian Udrea, Doctor of Philosophy, 2008

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

Since the adoption of the Resource Description Framework (RDF) by the World Wide Web Consortium (W3C), ontologies have become commonplace as a way to represent both knowledge and data. RDF databases have flexible schemas, are easy to integrate and allow a semantically rich query language. Unfortunately, these advantages come at the expense of increased query and application complexity. Existing RDF systems have attempted to address this problem by representing RDF data in relational format and translating queries and answers to and from SQL. As we will show, typical access patterns in RDF are substantially different than those in relational databases, to the extent that the performance of relational-backed systems degrades significantly for large datasets or complex queries.

In this dissertation, we propose two solutions to the scalability issue in RDF databases. First, we introduce Annotated RDF, a representation language that extends the semantics of RDF by allowing triples to be annotated with partially ordered information such as temporal validity intervals, probabilities, provenance

and many others. In standard RDF, using such information creates a blowup in the size of the database and therefore greatly increases the data complexity of queries. We define a query language for Annotated RDF that extends the RDF query language SPARQL and provides query processing and view maintenance algorithms. Our experimental evaluation shows Annotated RDF can answer queries 1.5 to 3.5 times faster than widely used systems such as Jena2, Sesame2 or Oracle 11g.

Second, we introduce GRIN, to our knowledge the first index structure designed specifically for SPARQL queries. We describe query and update processing algorithms and a theoretical analysis of index optimization. GRIN is extended to Annotated RDF and evaluated thoroughly on real-world datasets of up to 26 million triples and benchmark synthetic datasets of up to 1 billion triples. Our results show that for SPARQL queries, GRIN outperforms all relational index structures at comparable resource expenditure. Moreover, we show GRIN can be integrated with Annotated RDF, but also with existing systems such as Jena2 or LucidDB.

SCALABLE ONTOLOGY SYSTEMS

by

Octavian Udrea

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor V.S. Subrahmanian, Chair/Advisor
Associate Professor Lise Getoor
Assistant Professor Jeffrey Foster
Professor Rama Chellappa
Professor Mario Dagenais

© Copyright by
Octavian Udrea
2008

Dedication

I dedicate this thesis to my wife Raluca, whose love and support made all this possible.

Acknowledgments

I want to thank my advisor, Prof. V.S. Subrahmanian, who has made my PhD experience a truly wonderful journey. Without his guidance, energy and involvement this dissertation would not be possible. I also like to thank Prof. Lise Getoor, Prof. Jeffrey Foster and Prof. Rama Chellappa for giving the amazing opportunity to learn from them and collaborate on exciting projects. I would like to thank Prof. Mario Dagenais and all the members of my advisory committee for agreeing to dedicate part of their invaluable time to helping improve this dissertation.

I had the fortune of working in a wonderful group who made my graduate experience really exciting. I would like to thank Dr. Yu Deng, with whom I have worked together for two years, Dr. Andrea Pugliese whose insight has been invaluable over the last year, Dr. Diego Reforgiato Recupero, Dr. Edward Hung, Dr. Massimiliano Albanese, Amy Sliva, Gerry and Vanina Martinez, Matthias Brocheler and Cristian Lumezanu.

I also have to thank Edna Walker for the many forms I did not have to fill. And to all those whom I forgot to thank, my apologies! You have made my four years at Maryland truly amazing.

Last, but not least, I would like to thank my mother, who encouraged me to follow my dreams no matter how far they take me and my father who guided my first baby steps into this field.

Table of Contents

List of Tables vi

List of Figures vi

1 Introduction 1
- 1.1 The need for scalable ontology systems 1
- 1.2 Representation and query processing 4
- 1.3 Indexing 7

2 Overview of RDF database systems 8
- 2.1 RDF syntax and semantics 8
  - 2.1.1 The SPARQL query language 12
- 2.2 Current RDF database systems 13
  - 2.2.1 Knowledge representation 15
  - 2.2.2 Querying 17
  - 2.2.3 Indexing 19

3 Annotated RDF 20
- 3.1 aRDF Syntax 23
- 3.2 aRDF Semantics 29
- 3.3 Annotated RDF with infinite partial orders 36
- 3.4 aRDF Query Language 37
  - 3.4.1 Simple queries 37
  - 3.4.2 Conjunctive queries 44
- 3.5 Summary 48

4 Querying Annotated RDF 49
- 4.1 aRDF Query Processing Algorithms 49
  - 4.1.1 Answering atomic queries 50
  - 4.1.2 Simple non-atomic queries 55
  - 4.1.3 Conjunctive queries 58
- 4.2 aRDF View Maintenance 64
  - 4.2.1 Path Annotation Function 66
  - 4.2.2 Incremental Consistency Checking 67
  - 4.2.3 Insertions 70
  - 4.2.4 Deletions 74
- 4.3 Experimental evaluation 78
- 4.4 Summary 87

5 Indexing RDF 88
- 5.1 SPARQL graph patterns 90
- 5.2 The GRIN index 94
- 5.3 Answering queries with GRIN 99

5.4 GRIN optimization 104
5.4.1 Coverage and overlap 104
5.5 Handling updates 110
5.6 Extending GRIN to aRDF 113
5.6.1 Distance metrics for aRDF 113
5.6.2 Queries with transitive properties 117
5.7 Experimental evaluation 118

6 Conclusions and future work 127
6.1 Future work 128

Bibliography 134

List of Tables

1.1 Scalability comparison for SPARQL and SQL 3

4.1 Summary of consistency checking and atomic query algorithms 80

4.2 Summary of conjunctive answer algorithms 82

4.3 Summary of view maintenance algorithms 84

List of Figures

2.1 Graph representation of an RDF database 10

3.1 Four aRDF graphs 25

3.2 Consistency checking algorithm for aRDF databases 32

3.3 Example aRDF conjunctive query graph 45

4.1 Answering atomic aRDF queries $(r_q, p_q : a_q, ?v)$ 51

4.2 Answering atomic aRDF queries $(r_q, ?p : a_q, v_q)$ 52

4.3 Answering atomic aRDF queries $(r_q, p_q :?a, v_q)$ 54

4.4 Answering simple aRDF query $(?r, p_q : a_q, ?v)$ 56

4.5 Answering conjunctive aRDF queries through inexact graph matching 59

4.6 Answering conjunctive aRDF by heuristic ordering of the component queries 60

4.7 Example H_Q graph 61

4.8 Computing the path annotation function 67

4.9 Incremental consistency verification for insertions 68

4.10 View maintenance for atomic queries for insertions 71

4.11 View maintenance for atomic queries for deletions 76

4.12 Consistency checking and atomic query answers 81

4.13 Conjunctive queries and view maintenance 83

4.14 Comparison between aRDF and competing systems 85

5.1 A RDF graph from the ChefMoz dataset 91

5.2 RDF query example 91

5.3 GRIN index example 95

5.4 An algorithm to build the GRIN index 99

5.5 An algorithm to answer queries over the GRIN index 100

5.6 An algorithm to build the optimized GRIN index 109

5.7 **GRIN** insert maintenance 111

5.8 **GRIN** delete maintenance 112

5.9 (a) Synthetic aRDF database with $\mathcal{A}_{time-int}$; (b) Example **GRIN** index; (c), (d) Example aRDF queries. 114

5.10 **GRIN** load time 119

5.11 **GRIN** query processing time 120

5.12 **GRIN** peak memory usage 121

5.13 Selectivity and number of constraints analysis 122

5.14 **tGRIN** memory requirements and query processing time 124

5.15 **tGRIN** performance for complex queries 125

Chapter 1

Introduction

1.1 The need for scalable ontology systems

Initially thought of as a language for describing metadata about web pages, the use of the Resource Description Framework (RDF) has expanded significantly since its adoption as a standard by the World Wide Web Committee (W3C). RDF databases are now used in a wide variety of domains ranging from Life Sciences to personal information management systems. The fast spread of RDF can be traced to its many attractive features. First, RDF databases have very flexible schemas that require little maintenance; practically any new item of information can be added without fear of inconsistencies. Second, the core of the data model is a very simple construct, the triple. A triple is of the form *(subject, property, value)*. For instance, the triple *(CollegePark, locatedIn, Maryland)* states that the real-world entity labeled College Park is located in Maryland, whereas the triple *(Maryland, hasPopulation, 5615727)* states that the population of the state of Maryland is approximately 5.6 million. Third, RDF databases have a natural graphical representation which makes the data model user-friendly. Each triple corresponds to an edge between the subject and the value of the triple that is labeled with the property. Note that the two triples we have shown are "linked" by the common entity, *Maryland*. Fourth, RDF supports a very rich query language in which queries are

provided by the user as graphs where variables can label the nodes and/or edges. The query graph is then matched against subgraphs of the database to located values for the query variables. The most popular RDF query language to date is called SPARQL (Simple Protocol and RDF Query Language).

The exciting features of RDF come at the cost of query and application complexity. The combined complexity[1] of answering SPARQL queries has been proved to be PSPACE-complete [45] and the best subgraph matching algorithms [9] used to answer such queries have a worst-time complexity of $\mathcal{O}(N!)$, where N is the size of the database. Fortunately, many RDF database systems [25, 49, 63] were developed to alleviate some of the complexity issues. More recently, RDF databases have gained commercial support in the Oracle 10g and 11g database servers. The vast majority of systems[2] were designed to take advantage of decades of advances in relational data storage, indexing and query optimization. The typical approach stores RDF triples in a relational database, using relational indexing to speed up queries and translating SPARQL queries to SQL and the resulting relational tuples back to RDF. This type of translation results in a large number of relational joins to the detriment of scalability. We performed a small comparative analysis of the scalability of SPARQL and SQL queries for various data sizes and queries of 15% selectivity[3]. We used datasets between 10 and 100 million triples for RDF generated with the Lehigh University Benchmark [20] and from 10 to 100 million

[1] A complexity measure for databases in which both the data and the query are considered part of the input.

[2] We are only aware of one exception, Mulgara `www.mulgara.org`

[3] The selectivity of a query is the percentage of data entities – tuples or triples – that are returned.

relational records generated according to the TPC-C benchmark [55]. We execute queries serially and measured the number of successfully executed queries in a 5 minute interval. In a first execution, we used Sesame2 [7] backed by a PostgreSQL representation for the RDF data and the PostgreSQL 8.0 DBMS for the relational data. In the second execution we used Oracle 11g for both RDF and relational data; no indexes were defined. The results are shown in Table 1.1. We notice two critical aspects:

1. The performance for RDF queries is much lower than that for relational queries, even at identical selectivities.

2. The performance for RDF queries degrades quickly for large datasets (from 82 queries in 5 minutes for a 10 million records dataset to 15 queries for a 100 million records dataset for Oracle 11g), while it is relatively flat for relational databases (from 356 queries in five minutes for a 10 million records dataset to 276 queries for a 100 million records dataset).

Table 1.1: Scalability comparison for SPARQL and SQL

Dataset size [millions]	10	30	50	70	90	100
Sesame2/PostgreSQL 8.0 [queries/5 mins]	49	40	31	22	16	8
PostgreSQL 8.0 [queries/5 mins]	134	121	117	106	96	93
Oracle 11g/RDF [queries/5 mins]	82	74	59	41	28	15
Oracle 11g/relational [queries/5 mins]	356	344	321	295	285	276

Many widely used RDF databases are sufficiently large so that we need to be concerned with scalability issues. For instance, the Universal Protein Resource

(uniProt) [61] is an RDF dataset with protein information consisting of approximately 5 billion triples. The website reports approximately 20,000 queries per day. GovTrack is a non-governmental organization monitoring the US Congress. The dataset is now at 26 million triples, up 6 million triples from 7 months ago. Approximately 10,000 queries are executed every day by users and an additional 15,000 for internal research. These are just two examples from a much larger list of databases currently in use. The catalog at `www.rdfdata.org` provides a good starting point for the study of these datasets. At the current rate of expansion of RDF databases, scalability will undoubtedly be a critical issue in the very near future.

In addition, RDF is the base representation format for richer ontology languages such as the Web Ontology Language (OWL). A large portion of answering OWL queries involves processing parts of the query over RDF. As the Semantic Web gains momentum, efficient RDF data management will play a major part in enabling the next generation of applications to share and query large amounts of information quickly.

1.2 Representation and query processing

In Section 1.1, we mentioned model simplicity and flexibility as some of the RDF data model's most important advantages. However, the simplicity of the triple-based data model also brings certain problems. We used the triple *(Maryland, hasPopulation, 5615727)* as an example. Such a triple clearly cannot be valid at all

timepoints[4]. To express the fact that this triple has been valid in 2006 in RDF, we can construct the following set of triples:

$(_, rdf : type, rdf : Statement)$

$(_, rdf : Subject, Maryland)$

$(_, rdf : Predicate, hasPopulation)$

$(_, rdf : Object, 5615727)$

$(_, validTime, 2006)$

The textual representation of these triples in RDF form is given below.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:type="Statement">
          <rdf:Subject rdf:about="Maryland"/>
          <rdf:Predicate rdf:about="hasPopulation"/>
          <rdf:Object rdf:about="5615727"/>
          <validTime>2006</validTime>
     </rdf:Description>
</rdf:RDF>
```

Note that we create a new anonymous (or blank) node _ that represents the original triple (also called a statement), identified the resources and values that are the subject, predicate and object of the triple and finally created a new property called *validTime* and linked the statement to 2006 through this property. This process is known in RDF as *reification*, which allows us to make statements about other statements in RDF. Although the concept of expressing metadata about metadata is very interesting, note that instead of a single triple (or at most two after introducing the *validTime*) we now have five triples in the dataset. Since query complexity depends on the size of the dataset, reification can bring a dramatic increase in the

[4]In a very strict practical interpretation, it is unlikely to be valid at more than one point in time, since population numbers are constantly changing. For the sake of the example, we will assume the triple is valid in the year the census data was collected.

processing time of queries. Furthermore, even though we introduced *validTime* as a property, it does not have any special semantics with respect to the rest of the dataset (i.e., it does not automatically imply that if we are not in 2006, the original triple will not be considered). We describe reification, as well as the semantics of RDF and some of the related work in Chapter 2.

A vast majority (over 99%) of the real-world datasets we studied only use one level of reification (i.e., there are no blank nodes linked to other blank nodes) and it was usually for the purpose of adding a fixed type of information to the triples – for instance, validity times or time intervals, confidence levels or provenance information. It is therefore much more effective to add the new data as part of the triple itself and give it semantics at the same time. For instance, the triple about the population of Maryland could be written *(Maryland, hasPopulation: 2006, 5615727)* and interpreted as valid only in the year 2006. To accomplish this, in Chapter 3 we introduce a new representation language called Annotated RDF that allows triples to be annotated with members of a partially ordered set. The new representation language can keep annotated datasets small and therefore process queries more efficiently than standard RDF database systems. It also introduces the concept of transitivity for user-defined properties – informally, if a property p is transitive, from (x, p, y) and (y, p, z) we can infer (x, p, z). We found that many datasets specified special semantics for transitive properties (for instance, *relatedTo* properties that link topics in the RDF representation of Wikipedia) separately from the dataset because RDF does not provide support for property transitivity.

With the exception of Gutierrez et al. [22], who provided a temporal exten-

sion to RDF and in-memory algorithms for answering queries, we are not aware of any query processing algorithms that operate directly on RDF data; existing RDF systems translate SPARQL queries to SQL. In Chapter 4, we provide several algorithm for answering SPARQL-like queries over Annotated RDF, together with a theoretical analysis of their complexity and an extensive empirical evaluation. Chapter 4 also describes the first – to our knowledge – view maintenance algorithms for SPARQL-like queries. Previous work by Hung et al. [29] describes view maintenance algorithms for RDF aggregate queries only.

1.3 Indexing

One of the principal problems in answering SPARQL queries efficiently was the lack of an index specialized for RDF. Existing systems typically rely on a combination of relational indices to speed up query processing. Such index structures are very well adapted to quickly locating values of attributes under a given set of constraints. However, in SPARQL queries, the structure of the query graph, hence the relationships between nodes is more important than their individual values. In Chapter 5 we introduce the first – to our knowledge – RDF index for SPARQL queries called GRIN. We extend GRIN for Annotated RDF, provide query processing and index construction algorithms and conduct a thorough experimental evaluation. The results show that GRIN can process queries several times faster than the best existing systems.

Chapter 2

Overview of RDF database systems

The Resource Description Framework is a W3C standard endorsed by over 500 companies. It is a framework for representing and processing metadata, with the stated goal of providing interoperability between applications that exchange data over the Web. Lassila et al. [34] introduced the model for representing RDF metadata as well as the syntax for encoding it; their work is refined and extended in the RDF specification [40]. The schema language RDFS (RDF Schema) was later introduced by Brickley et al. [6] and extended with a complete system of inference rules by Hayes [26]. The central element of RDF – the triple – is the basis for describing relationships between resources in terms of properties (attributes) and values. In contrast to an object-oriented model, RDF is property-centric. Its schema language, RDFS defines vocabulary to describe classes, properties and their relationships.

2.1 RDF syntax and semantics

The underlying model of RDF is a labeled directed graph where nodes are resources or literals. Each edge in the graph corresponds to a triple *(subject, predicate, object)*, where *subject* is a resource, *predicate* is the edge label and *object* is either a resource or a literal. The basic elements in the data model are:

- *Resources* are anything that can be identified by a Uniform Resource Identifier (URI). We will denote the set of resources by $\mathcal{R}$.

- *Literals* may be either plain or typed. Plain literals are strings; typed literals are strings combined with an URI denoting a basic data type. We will denote the set of literals by $\mathcal{L}$.

- *A property* is a resource that represents a specific characteristic or relation used to describe other resources. Note that properties are resources themselves. We will denote the set of properties by $\mathcal{P}$. Sometimes, it is useful to differentiate between properties and other resources for reasons of clarity.

- *Statements* (or triples) are ordered tuples that state a resource (the subject) is associated with a property and a value for that property (the object). Statements are a subset of $\mathcal{R} \times \mathcal{P} \times (\mathcal{R} \cup \mathcal{L})$.

A simple example RDF database is shown below and its graph representation is shown in Figure 2.1. In this dataset, 240, 208, etc. are literals and all other entities are resources.

```
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:lib="http://www.zvon.org/library">

     <rdf:Description about="Matilda">
          <lib:creator>#RoaldDahl</lib:creator>
          <lib:pages>240</lib:pages>
     </rdf:Description>

     <rdf:Description about="The BFG">
          <lib:creator>#RoaldDahl</lib:creator>
          <lib:pages>208</lib:pages>
```

```
    </rdf:Description>

    <rdf:Description about="Heart of Darkness">
        <lib:creator>#JosephConrad</lib:creator>
        <lib:pages>110</lib:pages>
    </rdf:Description>

    <rdf:Description about="Lord Jim">
        <lib:creator>#JosephConrad</lib:creator>
        <lib:pages>314</lib:pages>
    </rdf:Description>

    <rdf:Description about="The Secret Agent">
        <lib:creator>#JosephConrad</lib:creator>
        <lib:pages>249</lib:pages>
    </rdf:Description>
</rdf:RDF>
```

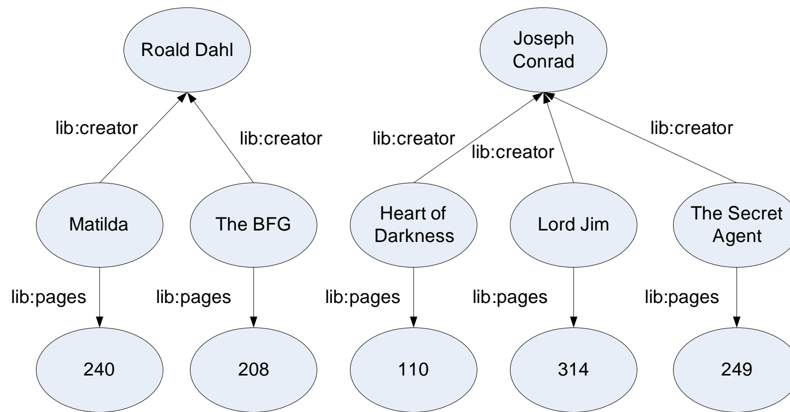


Figure 2.1: Graph representation of an RDF database

RDF supports two types of constructs with special semantics:

1. *Blank nodes* are not identified by an URI. They are essentially interpreted as existential variables. For instance, in the triple *(_, wrote, Beowful)*, the blank node _ signifies we know there existed someone that wrote Beowulf, but we do not know who. An RDF graph without blank nodes is called *ground*.

2. *Reification* is an alternate way of representing statements. For instance, the

triple *(Lord Jim, lib:creator, JosephConrad)* can be reified as *(_, rdf:Subject, Lord Jim)*, *(_, rdf:Predicate, lib:creator)*, *(_, rdf:Object, JosephConrad)*. By using reification, we can link statements with other resources or values.

The RDF semantics are defined through the means of an interpretation, which maps the resources, properties, literals and even statements to a set of "concrete" things called the universe of the interpretation. The full details of the RDF model theory semantics are given in [26]. For our purposes, it is enough to point out that with the exception of data type clashes, all RDF databases are consistent.

The RDF Schema introduces a class and property hierarchy in RDF:

- *rdfs:Resource*. Every resource is an instance of this class.
- *rdfs:Class*. Every class is an instance of this class, including rdfs:Class.
- *rdfs:subClassOf* is a property that states all instances of a class are instances of another. For example, *(Human, rdfs:subClassOf, Mammal)* means all humans are mammals.
- *rdfs:subPropertyOf* induces a hierarchy on the set of properties. If *(hasFather, rdfs:subPropertyOf, hasParent)* and *(Dan, hasFather, Michael)* then it also holds that *(Dan, hasParent, Michael)*. Note that this construct and rdfs:subClassOf introduce a basic form of inference in RDF.
- *rdf:type* is used to specify that a resource is an instance of a class. For instance, *(Michael, rdf:type, Human)*.

Despite the introduction of basic inference capabilities, these do not suffice for many real-world datasets we encountered. In Chapter 3, we will extend the RDF semantics to allow transitivity of user-defined properties (any property other than the ones outlined above).

2.1.1 The SPARQL query language

Many query languages have been developed for RDF, but the SPARQL language is now implemented by almost all RDF APIs. The SPARQL syntax looks a lot like SQL, with a few modifications. The central element of a SPARQL query is called a graph pattern.

Example 2.1. *Consider the database in Figure 2.1. The following is a slightly simplified SPARQL query over this database:*

$$SELECT\ ?w,\ ?b\ FROM$$

$$\{(Matilda\ lib:Creator\ ?w)\ .\ (?b\ lib:Creator\ ?w).$$

$$(?b\ lib:pages\ ?pn)\}\ FILTER\ (?pn\ >\ 200)$$

The query informally looks for a writer ?w and a book ?b such that ?w wrote Matilda, ?w wrote ?b, ?b has a certain number of pages ?pn that must be greater than 200. The answer to this query is $?w = RoaldDahl$ *and* $?b =$ *The BFG.*

Despite their apparent simplicity, SPARQL queries can be very complex to answer. Pérez et al. [45] have shown that the combined complexity of answering general SPARQL queries is PSPACE-complete. They also show that the data complexity of answering SPARQL queries is polynomial, but they were unable to provide a polynomial-time algorithm. The best subgraph matching algorithms [9]

can answer SPARQL-like queries with a worst-case complexity of $\mathcal{O}(N!)$, where N is the size of the dataset.

An interesting development for SPARQL has been the appearance of SPARQL endpoints, Web Services that allow users to ask SPARQL queries via HTTP. One successful example is the Nokia product catalog endpoint.

2.2 Current RDF database systems

In this section we provide a brief overview of some of the current RDF database systems and relevant research. We will focus primarily on those systems used for comparisons in our experimental evaluation.

Jena2 [63] is a very popular API and RDF database system developed at HP Research Labs. One of the first comprehensive toolkits for RDF, it includes components for RDF I/O, storage, querying and inference. It supports both in-memory and on-disk handling of RDF data. For the secondary storage, it uses a variation of a widely used scheme to represent RDF in relational databases called the *triple store*. In this approach, each RDF statement is stored as a single row in a "statements" table. To save space, Jena2 also normalizes some of the long literals and resources with long URIs. This means the resources and literals are stored in separate tables and then referenced from the statements table. Jena2 currently supports all relational databases for which there exists a JDBC (Java DataBase Connectivity) driver. Jena2 can use relational indexes on the statements table to speed up query processing.

Sesame2 [7] is an open-source RDF framework that supports RDF schema inference. The framework contains its own I/O library for fast access to RDF called Rio. It supports a wide variety of backend representations, including relational databases, main memory, filesystems, keyword indexers and its internal flat file representation. When deployed over a relational databases, Sesame2 creates 6 index structures on the statements table, one for each of the subject, predicate, object of the triple and three more for combinations of two of the above. Sesame2 supports the SPARQL and ReQL query languages.

RDFBroker [49] is an RDF storage system that uses a relational database as a backend. However, unlike Jena2 and Sesame2, it does not use the standard triple store approach. Instead, it creates a database schema based on *signatures* – sets of properties that are likely to be used together to answer queries. The main drawback of the approach is that the number of tables in the schema tends to be monotonic with the size of the RDF dataset. RDFBroker supports a subset of SPARQL queries.

3store [25] is an RDF triple store that has been ported from an older storage system WebKBC. It supports RDQL and SPARQL queries, but only over HTTP. 3store is backed by MySQL and BerkeleyDB.

Mulgara Semantic Store (`www.mulgara.org`) is a metadata storage systems that also supports RDF, but only through its proprietary query language called iTQL. It provides native RDF support, which means it does not rely on standard relational-to-RDF mapping.

Oracle has supported RDF since the 10g version of their database server. Now, as part of the 11g package, they also provide a lightweight platform dedicated

to RDF called Oracle Spatial 11g. It supports several query languages, including SPARQL and several proprietary index structures.

2.2.1 Knowledge representation

Most modern knowledge representation systems evolved from Description Logics (DLs) [3, 60], soon followed by corresponding reasoning algorithms [4]. Horrocks et al. [28] prove that RDF and OWL correspond to description logic from the $\mathcal{SHIQ}$ family. Su and Ilebrekke [53] provide a comprehensive comparison of ontology languages and tools, some of which we will present briefly here.

CycL [36] was one of the first ontology languages derived directly from first order predicate logic. It later evolved as part of the Cyc project as an ontology for commonsense reasoning. Ontolingua [13] evolved from an earlier languages called the Frame Ontology and provides reasoning support over terms such as class, subclass-of and instance-of. Because axioms cannot be expressed in this form, Ontolingua adds another layer on top of frame-based logic and represents ontologies in the Knowledge Interchange Format (KIF). Frame Logic [32] is a logic language integrated with an object-oriented programming paradigm. Concepts such as class, methods, types and inheritance have direct representations in the language. However, frame logic lacks some of the more powerful characteristics in Ontolingua (such as reification – the ability to use formulas as terms in meta-formulas). OCML (Operational Conceptual Modeling Language) was developed by the Knowledge Media Institute (KMI) as part of the VITAL [48] project. It provides mechanisms for defining relations,

functions, classes, instances, rules and procedures and supports internal theorem proving and function evaluation mechanisms; the primary goal of the language was to serve in rapid prototyping environments. LOOM [46] is the first knowledge representation language based on description logic. One of the primary tasks of the language is to provide support for computing subsumption relationships between descriptions and organizing them into taxonomies. Telos [44] is another language with an object-oriented focus. In addition to the previous languages, it can specify integrity constraints and it has extensions for temporal specification.

Starting in the 1990s, knowledge representation languages focused on modeling data from the World Wide Web. RDF was one of the first such languages, soon followed by special vocabularies for temporal [9], fuzzy, [10, 51] and provenance information [8]. OIL (Ontology Inference Layer) [14] is both a representation and an exchange language for ontologies. The language has primitives from frame logic and reasoning services and formal semantics based on description logic. In parallel with OIL, DARPA developed their DAML (DARPA Agent Modeling Language), with similar characteristics. The two finally were merged in DAML+OIL [43]. OWL was eventually developed from DAML+OIL and then branched into three levels of complexity: OWL Full (undecidable), OWL DL which covers most of description logic and OWL Lite which is the most tractable of the three.

There has also been a solid body of work on extending RDF with new features such as time intervals and uncertainty. Gutierrez et al. [22] have been the first to propose a model for RDF enhanced with valid-time intervals. They also provide a model theory semantics for Temporal RDF, as well as a query algorithm; unfortu-

nately, no empirical evaluation is presented. We have recently extended their model to handle uncertainty in the temporal annotations [47] – for instance, in cases when we know the triple holds at some point during the interval, but we do not know when. Dubois et al. and Straccia et al. [10, 51] have introduced a possibilistic and fuzzy extension for description logics (and by extension to RDF). Caroll et al. [8] describes a model for representing named RDF graphs, thus allowing statements about RDF graphs to be represented in RDF. Gergatsoulis and Lilis [17] define a model for representing multi-dimensional RDF, where information can be context dependent; for instance the title of a book may be represented in different languages.

2.2.2 Querying

An excellent survey of RDF query languages and their capabilities is given in [24]. We will briefly survey a few of the prominent languages.

RQL (The RDF Query Language) is a typed language following a functional approach. It supports generalized path expressions with variables both on nodes and edge labels. RQL relies on a formal graph model that captures the RDF modeling primitives and permits the interpretation of superimposed resource descriptions by means of one or more schemas. RQL follows an OQL-like syntax: *Select Pub from* $\{Pub\}$ *ns3:year* $\{y\}$ *where* $y =$ *"2004"*.

SeRQL stands for Sesame RDF Query Language and is a querying and transformation language loosely based on several existing languages, such as RQL, RDQL and N3. SeRQL syntax is similar to that of RQL though modifications have been

made to make the language easier to parse. Like RQL, SeRQL is based on a formal interpretation of the RDF graph, but SeRQL's formal interpretation is based directly on the RDF Model Theory.

The syntax of **RDQL** follows an SQL-like select pattern, where a from clause is omitted. For example, *select ?p where (?p, <rdfs:label>, "foo")* collects all resources with label "foo" in the free variable p. The select clause at the beginning of the query allows projecting the variables. Namespace abbreviations can be defined in a query via a separate *using* clause. RDF Schema information is not interpreted.

Notation3 (N3) provides a text-based syntax for RDF. Therefore the data model of N3 conforms to the RDF data model. Additionally, N3 allows to define rules, which are denoted using a special syntax, for example: *?y rdfs:label "foo"* ⇒ *?y a :QueryResult*. Such rules, while not a query language, can be used for the purpose of querying.

XsRQL (XQuery-style RDF Query Language) derives much of its syntax from the XQuery language for XML. It is a typed, functional language and provides a library of built-in functions that can be used in expressing queries.

Work on query and view maintenance algorithms in RDF is relatively minuscule. Volz et al. [62] were the first to introduce views into RDF. The required that the results of queries contain class instances and that the result itself has the pattern of an RDF statement. Magnaraki et al. [39] proposed RVL – a language for RDF views. However, they do not address the view maintenance problem. Hung et al. [29] present a mechanism to handle aggregate queries and update aggregate views over RDF databases. Very recently, Stocker et al. [50] presented a method

for optimizing basic graph patterns in SPARQL queries. Their method is based on gathering statistics about the RDF data beforehand, which allows the query optimizer to better estimate the selectivity of query components. The algorithms have been implemented as part of the Jena2 ARQ framework.

2.2.3 Indexing

Work in indexing RDF is sparse as well. Previous work was focused primarily on *path queries* [37], in which queries are path expressions (akin to regular expressions) or *reachability queries* [42], in which the purpose is finding out whether a vertex or set of vertices is reachable from a fixed start vertex. Path queries are expressible in SPARQL, but form a very small subset of the language. Heiner et al. [52] propose an architecture for querying distributed RDF repositories, based on the Sesame system. Graph indexing is also focused on a different type of queries, in which the goal is to find from a set of graphs the ones that are supergraphs to the query [2, 64, 56].

Recently, Abadi et al. [1] have proposed a method to speed up SPARQL queries by vertically partitioning the statements table. Their approach thus avoids many of the self-join operations that result from translating SPARQL into SQL. A similar technique is used in column stores, which store data on disk by column rather than row [11], an optimization that benefits query processing rather than handling updates.

Chapter 3

Annotated RDF

In Section 1.1 we presented empirical evidence that shows relational-backed RDF systems such as Jena2 and Oracle 11g exhibit poor performance for queries over reified triples. To determine how these queries can be answered more efficiently, we examined 35 real-world RDF datasets available at `www.rdfdata.org`, an online catalog of RDF databases. The datasets span multiple domains from congressional information to life sciences. They range from 12 thousand – W3C standards dataset – to over 91 million – Wikipedia^3 (note that this is not a footnote, but the actual name of the dataset), an RDF representation of Wikipedia information triples[1]. We looked at the type of data typically associated with reified statements and found the following:

- With a single exception (the `daml.org` publication metadata), each dataset annotates **all** its reified triples with the same type of information.
- 77% of the datasets attached temporal or fuzzy[2] values to their reified statements, 10% used both temporal and fuzzy values, while the remaining datasets used a discrete set of provenance sources.
- In 85% of the datasets, transitive properties were specified in the attached

[1] Some datasets were omitted due to their specialization. For instance, uniProt is a dataset of over 5 billion triples, but is hardly understandable outside life sciences.

[2] Confidence levels in $[0, 1]$.

documentation. The semantics of a transitive property p informally state that from (x, p, y) and (y, p, z) we can infer the triple (x, p, z). Typical examples of transitive properties include *relatedTo* that links topics in Wikipedia[3] or *friend-of-a-friend* (FOAF) relations between persons. Since RDF does not support transitivity for user-defined properties, the list of such properties is typically described in the documentation of each dataset and must be implemented in the application logic rather than the database.

The findings of this survey suggest two improvements over the RDF semantics. First, data used to annotate reified triples can be "moved" inside the triple itself; this approach eliminates the vast majority of blank nodes (if all triples are reified, it reduces the data size by 75%), hence reducing query complexity. Second, the introduction of transitivity for user-defined properties is useful to the large majority of application domains.

In this chapter, we propose an extension to the standard RDF semantics that incorporates these two observations. Our *Annotated RDF* [58] (or aRDF for short) attaches members from an arbitrary partial order to RDF triples and defines semantics for property transitivity. Annotated RDF builds on top of annotated logic [33, 35], which has been subsequently used, extended and improved [15] for a wide range of knowledge representation tasks. aRDF also incorporates probabilistic RDF [59], an extension we previously defined to represent uncertainty in RDF databases. In Chapter 4, we show that as anticipated, answering queries using aRDF is 1.5 to 3.5 times faster than systems such as Jena2, Sesame2 or Oracle 11g.

Other authors have previously recognized the need to extend RDF with new features. Gutierrez et al. [22] annotate triples with time intervals, by stating that a triple holds at all points in a given interval, but does not hold at any time point outside it. Dubois and Prade [10] and Straccia [51] annotate RDF triples with uncertainty (though these are one page position papers). Carroll et al. [8] describe a model for representing named RDF graphs, thus allowing statements about RDF graphs to be represented in RDF. Gergatsoulis and Lilis [17] define a model for representing multi-dimensional RDF, where information can be context dependent; for instance the title of a book may be represented in different languages. Our contributions are different than the above in that:

- aRDF is the first approach that handles many types of annotations — temporal intervals, fuzzy vales, provenance or combinations of these — under a unified semantics.

- aRDF is the first approach that handles transitivity of user-defined properties.

- To our knowledge, this is the first approach that proposes a query language similar to SPARQL and query processing and view maintenance algorithms for this language.

- None of the previous approaches provides an empirical evaluation on real-world datasets or on synthetic data of more than 5,000 triples. In Chapter 4, we provide an extensive evaluation of aRDF on real-world datasets of up to 26 million triples and synthetic datasets of up to 10 million triples.

3.1 aRDF Syntax

In this section we define the syntax of aRDF triples. We assume the existence of a partially ordered finite set $(\mathcal{A}, \preceq)$ where elements of $\mathcal{A}$ are called *annotations* and $\preceq$ is a partial ordering on $\mathcal{A}$. We further assume $\mathcal{A}$ has a bottom element. For example, several of the scenarios we found in practice are:

1. $\mathcal{A}_{fuzzy}$ is a finite subset of the real numbers in the closed interval $[0, 1]$ with the usual "less than or equals" ordering.

2. $\mathcal{A}_{time}$ is a finite set of non-negative integers (denoting time points) with the usual "less than or equals" ordering.

3. $\mathcal{A}_{time-int} \subseteq \{[x, y] \mid x, y \in \mathbf{N}\}$ is a finite set of time intervals. The interval $[x, y]$ as usual denotes the set of all $t \in \mathbf{N}$ such that $x \leq t \leq y$. The inclusion ordering $\subseteq$ is a partial ordering on this set.

4. $\mathcal{A}_{provenance}$ could be an enumerated set consisting of the names of information sources with a partial ordering on them. If $s_1, s_2 \in \mathcal{A}_{provenance}$, then we could think of $s_1 \preceq s_2$ to mean that s_2 is more reliable than s_1.

5. $\mathcal{A}_{fuztime}$ is a finite set of pairs (x, y) such that $x \in [0, 1]$ is a fuzzy value and y is a time point. The $\preceq$ ordering on $\mathcal{A}_{fuztime}$ can be defined as $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

These are just a few examples of partial orders. All the partial orders above except $\mathcal{A}_{provenance}$ are complete lattices. A partially ordered set $(X, \leq)$ is a complete

lattice iff (i) every subset of X has a unique greatest lower bound and (ii) every *directed* subset of X has a unique least upper bound. A set $Y \subseteq X$ is directed iff for all $y_1, y_2 \in Y$, there is an $x \in X$ such that $y_1 \leq x$ and $y_2 \leq x$. Note that one can construct arbitrary combinations of partial orders by taking the Cartesian Product of two known partial orders and taking the pointwise ordering on the Cartesian Product as shown in the definition of $\mathcal{A}_{fuztime}$.

Suppose now that $(\mathcal{A}, \preceq)$ is an arbitrary but fixed partially ordered set. As in the case of RDF, we also assume the existence of some arbitrary but fixed set $\mathcal{R}$ of resources (including blank nodes), a set $\mathcal{P}$ of property names, and a set $dom(p)$ of values associated with any property name p.

An *annotated RDF database* (aRDF-database for short) is a finite set of triples $(r, p : a, v)$ where r is a *resource* name, p is a *property* name, $a \in \mathcal{A}$ and v is a value in $dom(p)$ (v could also be a resource name).

This representation also supports RDF Schema triples such as[3]:

(i) *(A, rdfs:subClassOf, B)* indicates a subclass relationship between classes (which are also resources);

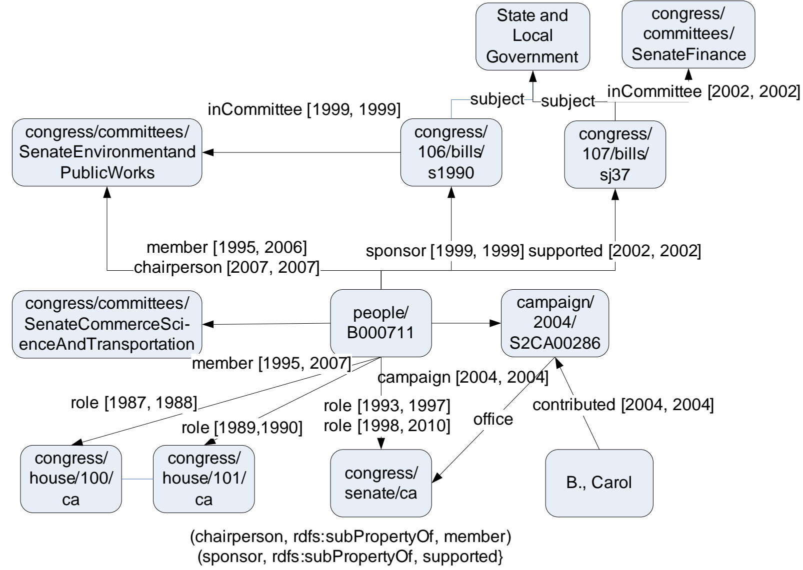
(ii) *(X, rdf:type, C)* indicates that a resource X is an instance of some class C;

(iii) *(p, rdfs:subPropertyOf, q)* denotes a sub-property relation between $p, q \in \mathcal{P}$[4]. We denote by $rdfs : subPropertyOf^*$ the transitive closure of $rdfs : subPropertyOf$.

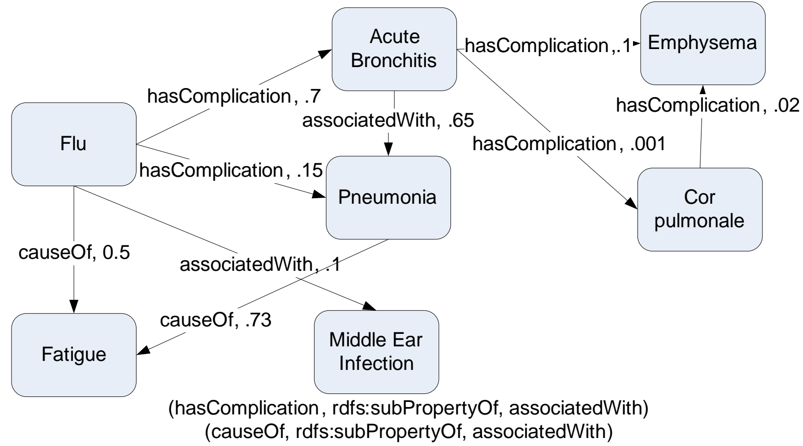
[3] $rdfs : range$ and $rdfs : domain$ are also possible, as well as any other RDFS construct. However, for the purpose of answering queries, $rdfs : subPropertyOf$ triples are the most important schema triples.

[4] Note we did not require that $\mathcal{P} \cap \mathcal{R} = \emptyset$.

Once $\mathcal{R}, \mathcal{P}$ and $dom(\cdot)$ are fixed, we use the notation $Univ$ to denote the set of all triples (r, p, v) where $s \in \mathcal{R}, p \in \mathcal{P}$ and $v \in dom(p)$. *Throughout this chapter, we will assume that* $\mathcal{R}, \mathcal{P}, \mathcal{A}, \preceq, dom(\cdot)$ *are all arbitrary, but fixed.*



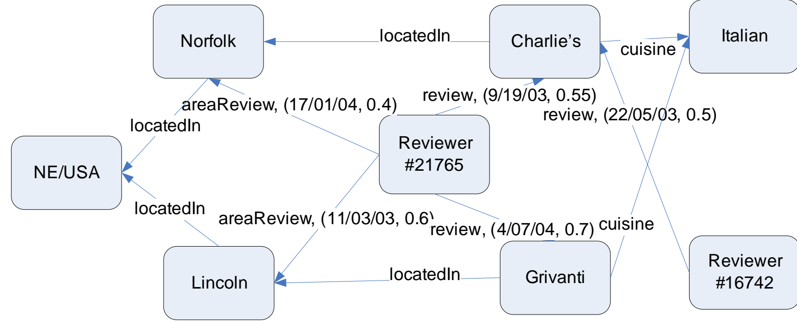
(a) Example aRDF graph annotated with $\mathcal{A}_{time-int}$. Extracted from the GovTrack dataset available at http://www.govtrack.us.



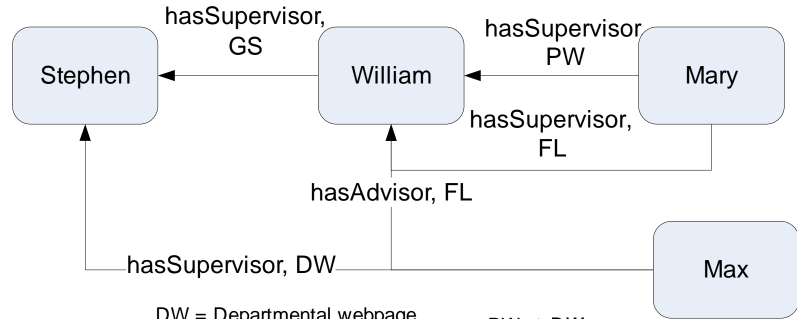
(b) Example aRDF graph annotated with $\mathcal{A}_{fuzzy}$. aRDF constructed based on information from www.wrongdiagnosis.com.

Figure 3.1: Four aRDF graphs

Definition 3.1. *(aRDF graph). Suppose $O \subseteq Univ$ is an aRDF-database. An* aRDF



(c) Example aRDF graph annotated with $\mathcal{A}_{fuztime}$. Extracted from the ChefMoz dataset available at http://chefmoz.org.



(d) Example aRDF graph annotated with $\mathcal{A}_{pedigree}$. Example is purposefully inconsistent to illustrate the aRDF consistency checking algorithm.

Fig. 3.1 (continued) Four aRDF graphs

graph *for* O *is a labeled graph* (V, E, λ) *where*

(1) $V = \mathcal{R} \cup \mathcal{L}$ *is the set of vertices.*

(2) $E = \{(r, r') \mid$ *there exists a property* p *such that* $(r, p : a, r') \in O\}$ *is the set of edges.*

(3) $\lambda(r, r') = \{p : a \mid (r, p : a, r') \in O\}$ *is the edge labeling function.*

It is easy to see that there is a one-to-one correspondence between aRDF databases and aRDF graphs. Hence, we will often talk interchangeably talk about

both aRDF databases and aRDF graphs.

Example 3.2. *Figure 3.1 shows four examples of* aRDF *graphs. Figure 3.1(a), annotated with elements of $\mathcal{A}_{time-int}$, is extracted from the GovTrack dataset. The dataset consists of approximately 12 million* aRDF *triples (1.5 GB) containing detailed information about the U.S. Congress and the election campaigns since the early 1980s until the present. The triple (people/B000711, role:[1987,1988], congress/house/100/ca) denotes the fact that the congressperson identified by people/B00711 was a representative of the state of California in the 100th Congress between 1987 and 1988.*

Figure 3.1(b) shows an example aRDF *graph constructed manually from information available at www.wrongdiagnosis.com, a website that presents medical information in an ontology-like fashion. The data is annotated with $\mathcal{A}_{fuzzy}$. The triple (Flu, causeOf:0.5, Fatigue) says that in 50% of cases of Flu, Fatigue is one of the symptoms. The size of the full dataset is 4547 triples.*

Figure 3.1(c) shows an example extracted from the ChefMoz dataset, which contains information on and reviews of restaurants throughout the world. The dataset consists of approximately 550,000 aRDF *triples (220 MB). We used the review information (time and score) from the dataset to annotate the triples. The triple (Reviewer #21765, review: (4/07/04, .7), Grivanti) denotes the fact that the reviewer with identifier 21765 wrote a review for the Grivanti restaurant on July 4th 2004, giving it a score of .7. In this example, the triples without annotations are assumed to be annotated with the current time and the value 1.*

Finally, 3.1(d) is an example annotated with pedigree information. The example will be used to illustrate the consistency checking algorithm for aRDF *in Section 3.2. In this dataset, there are four sources of information (described in the figure), along with a partial order based on the reliability of the sources. The triple (Max, hasSupervisor: DW, Stephen) denotes the fact that the department webpage (DW) lists that Stephen is Max' supervisor.*

As mentioned at the beginning of the chapter, aRDF differentiates between *transitive* and *non-transitive* properties. The lack of support for transitive properties p in standard RDF means that: (i) Inferences of the type $\frac{(x,p,y)\ (y,p,z)}{(x,p,z)}$ are all computed apriori for the entire database or (ii) inferences are computed as needed at query time, which places some of the query complexity burden on the application. In aRDF we assume that all properties in $\mathcal{P}$ are marked transitive or non-transitive. For instance, in Figure 3.1(d), we consider *hasSupervisor* to be a transitive property[5]. In this example, from *(Max, hasAdvisor:FL, William)* and *(hasAdvisor, rdfs:subPropertyOf, hasSupervisor)* we can infer that *(Max, hasSupervisor:FL, William)*. The example also states that *(William, hasSupervisor:GS, Stephen)*. According to the semantics of transitive properties, from these two triples we should be able to infer that *(Mas, hasSupervisor, Stephen)*, but we do not yet have a method for associating an annotation with this inferred triple. The concept of a *p-Path* will be later used to assign an annotation to the inferred triple.

[5]Although this may not always be the case in the real world, it is the case for synthetic datasets generated with the Lehigh University Benchmark.

Given a transitive property p, a p-path intuitively is a path in the aRDF graph that only consists of edges labeled with the property p. However, in some cases, an edge might be labeled with a property q which is a sub-property of p. In this case, the q edge is considered part of the p path because the triple (s, p, o) can be inferred from (s, q, o) when q is a subproperty of p. This is the intuition behind a p-path which is defined formally below.

Definition 3.3 (p-Path)**.** *Let O be an aRDF graph, p be a transitive property in O, and suppose $r, r' \in O$ are two vertices. There is a p-*path between *r* and *r' if there exist triples $t_1 = (r, p_1 : a_1, r_1), \ldots, t_i = (r_{i-1}, p_i : a_i, r_i), \ldots, t_k = (r_{k-1}, p_k : a_k, r') \in O$ such that for all $i \in [1, k]$ $(p_i, rdfs : subPropertyOf^*, p)$. We will denote a p-path Q by the set of triples $\{t_1, \ldots, t_k\}$ that form the path. We also denote by $A_Q = \{a_1, \ldots, a_k\}$ the set of annotations of the triples on the p-path Q.*

Example 3.4. *Consider the aRDF graph shown in Figure 3.1(d) and suppose the hasSupervisor property is transitive. The triples (Max, hasAdvisor:FL, William) and (William, hasSupervisor:GS, Stephen) form a hasSupervisor-path (remember that hasAdvisor is a subproperty of hasSupervisor). For this p-Path, $A_Q = \{FL, GS\}$.*

3.2 aRDF Semantics

In this section, we provide a declarative semantics for aRDF databases and study the consistency of such databases.

Definition 3.5. *An* aRDF-interpretation *I is a mapping from $Univ$ to $\mathcal{A}$.*

The definition of an aRDF-interpretation follows that in annotated logic [33]. However, there are two differences that we note here. First, annotated logic in [33] assumes that $\mathcal{A}$ is a complete lower semilattice, while we only require that it be a partial order. Second, our definition of satisfaction must take into account the difference between properties that are transitive and those that are not. This induces a more complex definition in our case than that in [33].

Definition 3.6. *An* aRDF-interpretation I *satisfies* $(r, p : a, v)$ *iff* $a \preceq I(r, p, v)$. I *satisfies an aRDF-database* O *iff:*

(S1) I *satisfies every* $(r, p : a, v) \in O$.

(S2) For all transitive properties $p \in \mathcal{P}$ *and for all p-paths* $Q = \{t_1, \ldots, t_k\}$ *in* O, *where* $t_i = (r_i, p_i : a_i, r_{i+1})$, *there exists* $a \in \mathcal{A}$ *such that* $a \preceq a_i$ *for all* $1 \leq i \leq k$ *and for all* $a \in \mathcal{A}$ *such that* $a \preceq a_i$ *for all* $1 \leq i \leq k$, *it is the case that* $a \preceq I(r_1, p, r_{k+1})$.

O *is* consistent *iff there is at least one aRDF-interpretation that satisfies it.* O entails $(r, p : a, v)$ *iff every aRDF-interpretation that satisfies* O *also satisfies* $(r, p : a, v)$.

The definition of satisfaction and the complex definition of case (S2) above are best illustrated with an example.

Example 3.7. *Let* O *be the aRDF graph in Figure 3.1(b), where* $\mathcal{A} = \mathcal{A}_{fuzzy}$. *Suppose the associatedWith property is transitive. Let* $I_0(t) = 1\ \forall t \in Univ$. I_0 *satisfies* O *and hence* O *is consistent. Furthermore,* $O \models$ *(Flu, causeOf: .4, Fatigue) because for any satisfying interpretation,* $0.4 \preceq 0.5 \preceq$ *I(Flu, causeOf, Fatigue).*

The intuition behind item (S2) of Definition 3.6 is related to the notion of entailment. For instance, in Figure 3.1(b) — with associatedWith transitive — from the triples (Flu, hasComplication: .7, AcuteBronchitis) and (AcuteBronchitis, associatedWith: .65, Pneumonia), we can infer that with a confidence level of at least .65, Flu is associatedWith Pneumonia since $\forall\ a \in \mathcal{A}_{fuzzy}$ *s.t.* $a \preceq .7$ *and* $a \preceq .65$ *(i.e.* $\forall\ a \preceq .65$*)),* $a \preceq$ *I(Flu, associatedWith, Pneumonia).*

It follows from Definition 3.6 that unlike RDF databases which are always consistent with the exception of data type clashes, aRDF databases can be inconsistent. Consider the aRDF graph in Figure 3.1(d) and assume the *hasSupervisor* property is transitive. We can identify the following sources of inconsistency:

1. The triples *(Mary, hasSupervisor:PW, William)* and *(Mary, hasSupervisor:FL, William)*[6] indicate that for any interpretation I, we cannot have that $PW \preceq$ *I(Mary, hasSupervisor, William)* and $FL \preceq$ *I(Mary, hasSupervisor, William)*, which contradicts item (S1) from Definition 3.6.

2. The presence of the different *hasSupervisor*-paths {*(Max, hasAdvisor:FL, William), (William, hasSupervisor:GS, Stephen)*} and {*(Max, hasSupervisor:DW, Stephen)*} means that for any interpretation I, we cannot have that $FL \preceq$ *I(Max, hasSupervisor, Stephen)* and $DW \preceq$ *I(Max, hasSupervisor, Stephen)*, thus contradicting item (S2) from Definition 3.6.

We now state a necessary and sufficient condition for checking consistency of

[6] The presence of such triples is reasonable since it indicates the same information was obtained from different sources for which we cannot compare the pedigree according to the partial order given.

an aRDF database. These conditions are needed because they are more amenable to constructing a consistency verification algorithm than Definition 3.6.

Algorithm aRDFconsistency(O,$\mathcal{A}$, $\preceq$)

Input: aRDF database O and annotation $(\mathcal{A}, \preceq)$.
Output: $True$ if O is consistent, $False$ otherwise.

1: **for** $(r, p, r') \in \{(r, p, r') | \exists\ a \in \mathcal{A}\ s.t.\ (r, p : a, r') \in O\}$ **do**
2: $\quad A \leftarrow \{a \in \mathcal{A} | (r, p : a, r') \in O\}$
3: $\quad$ **if** $|A| > 1$ **then**
4: $\quad\quad$ **if** $\nexists\ a \in \mathcal{A}$ s.t. $\forall a' \in A, a' \preceq a$ **then**
5: $\quad\quad\quad$ **return** $False$
6: $\quad\quad$ **end if**
7: $\quad$ **end if**
8: **end for**
9: **for** $p \in \mathcal{P}$ transitive **do**
10: $\quad O' \leftarrow O|_p$
11: $\quad P \leftarrow \{paths\ Q \subseteq O' |\ \nexists Q' \subseteq O' \wedge Q' \supset Q\}$
12: $\quad$ **for** $(r, r') \in N(O') \times N(O')$ **do**
13: $\quad\quad P' \leftarrow \{Q \in P | r, r'$ are the first and last vertex respectively in $Q\}$
14: $\quad\quad$ **if** $|P'| > 0$ **then**
15: $\quad\quad\quad A \leftarrow \{A_Q | Q \in P'\}$
16: $\quad\quad\quad B \leftarrow \{b \in \mathcal{A} | \exists A_Q \in A\ s.t.\ \forall\ a \in A_Q, b \preceq a\}$
17: $\quad\quad\quad$ **if** $\nexists\ a \in \mathcal{A}$ s.t. $\forall b \in B, b \preceq a$ **then**
18: $\quad\quad\quad\quad$ **return** $False$
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **end for**
23: **return** $True$

Figure 3.2: Consistency checking algorithm for aRDF databases

Theorem 3.8. *Let O be an aRDF database. O is consistent iff:*

(C1) $\forall p \in \mathcal{P}$ and $\forall\ r, r' \in \mathcal{R}$ such that there exist distinct $a_1, \ldots a_k \in \mathcal{A}$ and for all $i \in [1, k]$ $\exists (r, p : a_i, r') \in O$, then $\exists\ a \in \mathcal{A}$ s.t. $\forall i \in [1, k]$ $a_i \preceq a$ AND

(C2) $\forall p \in \mathcal{P}$ transitive, $\forall r, r' \in \mathcal{R}$, let $\{Q^1, \ldots, Q^k\}$ be the set of different p-paths between r and r' and let $\{A_{Q^1}, \ldots, A_{Q^k}\}$ be the annotations for these p-paths.

Let $B_{Q^i} = \{a \in \mathcal{A} | a \preceq a' \ \forall a' \in A_{Q^i}\}$. *Then* $\exists\ a \in \mathcal{A}$ *s.t.* $\forall b \in \bigcup_{i \in [1,k]} B_{Q^i}, b \preceq a$[7].

Proof. Let O be an aRDF database that meets conditions (C1) and (C2) above. Then we can build a satisfying interpretation as follows:

- For any set of triples that match (C1), assign $I(r, p, r') = a$. For any other triple $(r, p : a, v) \in O$, assign $I(r, p, v) = a$.
- For any transitive property p and pair of resources r, r' that match (C2), assign $I(r, p, r') = a$.

It is straightforward to show that the above interpretation satisfies Definition 3.6.□

Conversely, let O be a consistent aRDF database. Let I be a satisfying interpretation. We need to show that conditions (C1) and (C2) hold.

To see why condition (C1) holds, suppose $p, r, r', a_1, \ldots, a_k$ are as in (C1) and suppose $(r, p : a_i, r') \in O$. Then by condition (S1) in the definition of satisfaction, $a_i \leq I(r, p, v)$ for all $1 \leq i \leq k$. In this case, we can take a to be $I(r, p, v)$.

To see why condition (C2) holds, suppose p is transitive, $\{Q^1, \ldots, Q^k\}$ is the set of different p-paths between resources r and r', and $\{A_{Q^1}, \ldots, A_{Q^k}\}$ are the annotations for these p-paths. Then, by condition (S2) in the definition of satisfaction, for each annotation a^i_j in path Q^i, $a^i_j \preceq I(r, p, v)$. Therefore, $I(r, p, v)$ is an upper bound for the sets B_{Q^i} and hence can serve as the annotation a in (C2). □

The following result states that if we require $\mathcal{A}$ to be a partial order with a top element[8], then we are guaranteed consistency.

[7]Note that (C2) implies (C1) when p is transitive, since paths of length 1 are possible.
[8]An element $\top \in \mathcal{A}$ is a "top" element if $x \preceq \top$ for all $x \in \mathcal{A}$.

Corollary 3.9. *Let $\mathcal{A}$ be a partial order with a top element. Then any aRDF database O annotated w.r.t. $\mathcal{A}$ is consistent.*

The justification is immediate, since the interpretation that maps every triple in $Univ$ to the top element satisfies any aRDF database.

Theorem 3.8 provides an immediate algorithm for checking the consistency of aRDF databases. We present this algorithm in Figure 3.2. For a property p, we define $SP(p) = \{q \in \mathcal{P} | (q, rdfs : subPropertyOf^*, p)\}$. We denote by $O|_p$ the restriction of the aRDF graph O to triples containing properties from $SP(p)$. $N(O)$ denotes the set of vertices in the aRDF graph O. Algorithm $aRDFConsistency$ (Figure 3.2) starts by verifying that every set of annotations on triples with identical subject, property and value have a greatest lower bound (lines 1–8). In lines 9–22, the algorithm iterates through all p-Paths in the graph and for each p-Path it ensures that the set of annotations for the triples on that path has a greatest lower bound (lines 15–19).

Example 3.10. *Let O be the aRDF graph in Figure 3.1(d). When we run our consistency check algorithm and execution reaches line 4 with (r, p, r')=(Mary, hasSupervisor, William), $A = \{PW, FL\}$ from line 2. Since $\nexists\ a \in \mathcal{A}$ s.t. $PW, FL \preceq a$, the algorithm will determine that the database is inconsistent.*

Now consider the same aRDF database without the triple (Mary, hasSupervisor:PW, William). In this case, the algorithm will proceed to the loop starting on line 9. However, for the iteration for which $p = hasSupervisor$ on line 9 and $(r, r') = (Max, Stephen)$ on line 12, the set P' will contain the two possible

hasSupervisor-paths from Max to Stephen from Example 3.7. Consequently, on line 15, $A = \{\{DW\}, \{FL, GS\}\}$ *and on line 16* $B = \{DW, FL\}$. *Since* $\nexists\, a \in \mathcal{A}$ *s.t.* $DW, FL \preceq a$, *the algorithm will return* $False$ *on line 18.*

The following result states the correctness of our consistency check algorithm.

Proposition 3.11 (Consistency check correctness)**.** *The* $aRDFconsistency$ *on input* $(O, \mathcal{A}, \preceq)$ *returns* $True$ *iff* O *is consistent.*

Proof. The loop on lines 1–8 corresponds to condition (C1) in Theorem 3.8; lines 9–22 correspond to condition (C2) of the same theorem. The algorithm uses the fact that if property (C2) in Theorem 3.8 holds for maximal p-paths, then it will also hold for shorter p-paths. This result follows directly from Definition 3.6 and the definition of a partial order. □

The consistency check algorithm runs in polynomial time as shown below.

Proposition 3.12 (Consistency check complexity)**.** *Let* O *be an* aRDF *graph and let* $n = |N(O)|$, *let* $e = |O|$ *and let* $p = |\mathcal{P}|$. *Let* $(\mathcal{A}, \preceq)$ *be a partial order and let* $a = |\mathcal{A}|$[9]. *Then* $aRDFconsistency(O, \mathcal{A}, \preceq)$ *is* $\mathcal{O}(p \cdot (n^3 \cdot e + n \cdot a^2))$.

The result follows from the loop on lines 9—22. For any transitive property, we first compute the set of all maximal paths in $O|_p$ (line 11). Since we have to keep the paths in memory (and not only their cost), this operation can be performed in at most $n^3 \cdot e$ steps in a modified version of Floyd's algorithm [16] that records the paths explored. The loop on line 12 iterates through all the maximal paths found

[9]We assume without loss of generality that $a < e$, since we can use at most one annotation for each edge.

— there can be at most $2n$ of them. For each such path, we compute the set A (line 12), which takes at most e steps, since any maximal path is of length less than or equal to e. The size of each set A is bounded by a and the number of maximal paths for the entire graph is at most $O(n)$, meaning line 15 will be run at most $\mathcal{O}(n \cdot a^2)$ times. Line 16 is run at most $\mathcal{O}(n \cdot a^2)$ times as well, since $|B|$ is bounded by a.

3.3 Annotated RDF with infinite partial orders

So far, we have defined aRDF to use a finite partial order $(\mathcal{A}, \preceq)$. In this section, we show a straightforward extension to infinite partial orders. Let us assume that $\mathcal{A}$ is infinite and let O be a aRDF database. Our goal is to find a finite set of annotations $\mathcal{A}^* \subseteq \mathcal{A}$ such that O is consistent under $(\mathcal{A}^*, \preceq)$[10] if and only if O is consistent under $(\mathcal{A}, \preceq)$.

Proposition 3.13. *Let O be an inconsistent aRDF database annotated with $(\mathcal{A}, \preceq)$. Then O is inconsistent for any partial order $(X, \preceq)$ with $X \subseteq \mathcal{A}$.*

Proof. Assume that there exists an $X \subseteq \mathcal{A}$ such that O is consistent under the annotation $(X, \preceq)$. Then any satisfying interpretation I of O is clearly a valid interpretation for the superset $\mathcal{A}$ as well, hence O would be consistent for $(\mathcal{A}, \preceq)$.□

Let us assume that O is consistent under $(\mathcal{A}, \preceq)$. We build the set $\mathcal{A}^*$ in the following way. Let I be an arbitrary satisfying interpretation of O. We define $\mathcal{A}^* = \{a | \exists\ u \in\ Univ \text{ s.t. } I(u) = a\}$. Then it follows directly that I is a satisfying interpretation for O under the $(\mathcal{A}^*, \preceq)$ annotation. $\mathcal{A}^* \subseteq \mathcal{A}$ is clearly finite since

[10] Here, we assume that $\preceq$ is restricted to the elements of $\mathcal{A}^*$.

$|\mathcal{A}^*| \leq |Univ|$ and $Univ$ is finite.

Even though we can always reduce an infinite partial order to a finite subset as shown above, the consistency check algorithm in Figure 3.2 must still be able to handle computations on lines 4 and 16. This requires that for all infinite partial orders used, we must be able to compute a finite representation of the sets $\mathcal{A}_{\preceq a} = \{a' \in \mathcal{A} | a' \preceq a\}$ and $\mathcal{A}_{a\preceq} = \{a' \in \mathcal{A} | a \preceq a'\}$. This is true of all annotations we found in practice. For instance, for fuzzy values, let $x \in [0, 1]$ be an arbitrary annotation. Then the set $\mathcal{A}_{\leq x} = [0, x]$ and the set $\mathcal{A}_{x\leq} = [x, 1]$; the case of timepoints or time intervals is analogous. For such cases in which $\mathcal{A}_{\preceq a}$ and $\mathcal{A}_{a\preceq}$ can be computed in constant time, the complexity of the consistency check algorithm becomes $\mathcal{O}(n^3 \cdot e)$.

3.4 aRDF Query Language

In this section, we define the aRDF Query Language. We start by discussing simple queries – annotated triples in which any of the subject, property, value or annotation can be either constant or variable. We then extend these to general conjunctive queries and discuss their relationship to SPARQL graph patterns for RDF. Finally, we define the formal semantics of a correct answer to a query and provide a simple query processing algorithm.

3.4.1 Simple queries

We assume the existence of sets of variables ranging over resources, properties, values and $\mathcal{A}$. A term over one of these sets is either a member of that set or a

variable ranging over that set. An *aRDF query* is a triple $(R, P : A, V)$ where R, P, A, V are all terms over resources, properties, annotations and values respectively. An aRDF query of the above form is atomic if at most one term in it is a variable.

Example 3.14. *Consider the aRDF graphs in Figures 3.1(a)–(c). The following are atomic aRDF queries:*

- *What committees was people/B000711 a member of between 1997 and 2001? This is expressed as: (people/B000711,member:[1997, 2001],?v).*
- *What conditions is Flu associatedWith in at least 10% of cases (assuming hasComplication, associatedWith are transitive)? This can be expressed as: (Flu, associatedWith: .1, ?v).*
- *What reviewers gave the restaurant Grivanti scores of .5 or higher after 01/01/2004? This can be expressed as: (?s, review: (01/01/2004,.5), Grivanti).*

Definition 3.15 (Semi-unifiable aRDF triples)**.** *Suppose θ is a substitution. Two aRDF triples $(r, p : a, v), (r', p' : a', v')$ are θ semi-unifiable iff $r\theta = r'\theta \wedge p\theta = p'\theta \wedge v\theta = v'\theta$.*

We call triples following the conditions of Definition 3.15 semi-unifiable since we are do not require the existence of a substitution from a to a'. Note that this is particular, tractable case of the general semi-unifiability theory [30]. As usual, $r\theta$ denotes the application of the substitution θ to r. The definition of semi-unifiable aRDF queries also applies to aRDF triples as they are also simple aRDF queries.

A query consisting of a constant triple will return that triple if it exists in the

aRDF database or the empty set otherwise. A query that has a least one variable term will return the set of triples that are semi-unifiable with the query, are entailed by the aRDF database and have an annotation that is "greater than or equal to" the annotation in the query under the $\preceq$ order.

Definition 3.16 (Query answer)**.** *Let O be a consistent aRDF database and let $q = (r_q, p_q : a_q, v_q)$ be a atomic aRDF query on O. Let $A_O(q)$ be the set of triples $(r, p : a, v)$ such that for any $(r, p : a, v) \in A_O(q)$ the following hold:*

1. *$(r, p : a, v)$ contains no variables.*
2. *$(r, p : a, v)$ is semi-unifiable with q*
3. *$O \models (r, p : a, v)$*
4. *$((a_q$ is a variable$)$ or $(a_q \preceq a))$*

The answer *to q is defined as $Ans_O(q) = \{(r, p : a, v) \in A_O(q) |\ \nexists\ S \subseteq Ans_O(q) - \{(r, p : a, v)\}$ s.t. $S \models (r, p : a, v)\}$. We also define the set of answer substitutions as $\Theta_O(q) = \{\theta$ substitution $|\exists (r, p : a, v) \in A_O(q)$ such that $(r, p : a, v) = (r_q, p_a : a_q, v_q)\theta\}$.*

$A_O(q)$ consists of all ground (i.e. variable-free) instances of q that are entailed by O. However, $A_O(q)$ may contain redundant triples – for example, using our $time - int$ partial ordering, if $(r, p : [1, 100], v)$ is in $A_O(q)$, then there is no point including redundant triples such as $(r, p : [1, 10], v)$ in it. $Ans_O(q)$ eliminates all such redundant triples from $A_O(q)$.

Example 3.17. *Consider the queries in Example 3.14. The answers are:*

- $Ans_O(q) = \{$*(people/B000711, member: [1995, 2006], congress/committees/Senate-EnvironmentAndPublicWorks), (people/B000711, member: [1995, 2007], congress/committees/SenateCommerceScienceAndTransportation)*$\}$*. Note that the answer does not include for instance (people/B000711, member: [1997, 2001], congress/ committees/Senate-EnvironmentAndPublicWorks) since it is already entailed by a triple in the answer.*
- $Ans_O(q) = \{$*(Flu, associatedWith: .65, Pneumonia), (Flu, hasComplication: .1, Emphysema), (Flu, hasComplication: .7, AcuteBronchitis*$\}$.
- $Ans_O(q) = \{$*(Reviewer #21765, review: (4/07/04, .7), Grivanti)*$\}$.

Since we are looking for the answer to a query among all triples entailed by a database, we would like to find a set of conditions of entailment that can be checked by an algorithm. The interpretation-related conditions in Definition 3.6 do not directly support a tractable computational approach to entailment. The following result specifies a set of conditions that must hold when O entails a ground aRDF triple.

Theorem 3.18. *Let O be a consistent aRDF database and let $(r, p : a, v)$ be an aRDF triple. $O \models (r, p : a, v)$ iff one of the following conditions holds:*

(E1) $\exists$ $(r, p : a_1, v), \ldots, (r, p : a_k, v) \in O$ *and let A be the set of values a' such that $a_i \preceq a'$ $\forall i \in [1, k]$ ($|A| \geq 1$ since O is consistent). Then* $\forall$ $a' \in A$, $a \preceq a'$.

(E2) $\exists$ *p-paths $Q^1, \ldots, Q^k$ between r and v. Let $B_{Q^i} = \{b \in \mathcal{A} | b \preceq a' \ \forall a' \in A_{Q^i}\}$. Let A be the set of values a' such that* $\forall$ $b \in \bigcup_{i \in [1,k]} B_{Q^i}, b \preceq a'$ *($|A| \geq 1$ since*

O is consistent). Then $\forall\ a' \in A,\ a \preceq a'$.

Proof. Assume that none of (E1, E2) holds. Then we are in one of the following cases:

- There is no edge labeled with p or a subproperty of p or a p-path between r, v. Then for any satisfying interpretation that has $I(r,p,v) \neq \bot$, the interpretation I' s.t. $I'(t) = I(t) \forall\ t \in Univ - \{(r,p,v)\}$ and $I'(r,p,v) = \bot$ would also be a satisfying interpretation that implies $O \not\models (r, p : a, v)$.
- $\exists\ (r, p : a_1, v), \ldots, (r, p : a_k, v) \in O$ and $\exists a' \in A, a' \preceq a$. Then for any satisfying interpretation I, we can construct I' that differs from I in that $I'(r,p,v) = a'$; I' is also a satisfying interpretation, that does not satisfy $(r, p : a, v)$, which implies $O \not\models (r, p : a, v)$.
- The case where there exist p-paths is similar to the case in which there exist edges between r and v.

□

Given an database O, we can infer new triples from O using the following two operators, f_1, f_2 (we use two operators for the purpose of readability):

1. $f_1(O) = \{(r, p : a, v) | \exists\ (r, p : a_1, v), (r, p' : a_2, v) \in O$ s.t. $(p', rdfs : subPropertyOf^*,$ $p)\ \wedge a$ is a minimal upper bound[11] of $a_1, a_2\}$.

2. $f_2'(O) = \{(r, p : a, v) | \exists (r, p' : a_1, r'), (r', p'' : a_2, v) \in O$ s.t. $(p', rdfs :$ $subPropertyOf^*,\ p)\ \wedge (p'', rdfs : subPropertyOf^*, p) \wedge (\forall\ a' \in \mathcal{A}, (a' \preceq a_1 \wedge$

[11] a is an minimal upper bound of a_1, a_2 iff $a_1 \preceq a$ and $a_2 \preceq a$ and there is no other a' such that $a' \preceq a$ and $a_1 \preceq a'$ and $a_2 \preceq a'$.

$a' \preceq a_2) \Rightarrow (a' \preceq a))\}$. We then define $f_2(O) = \{(r, p : a, v) \in f_2'(O) |\ \nexists\ (r, p : a', v)\ s.t.\ a \preceq a'\}$.

Let $\mu(O) = f_1(O) \cup f_2(O)$.

Proposition 3.19 (Closure of O)**.** *μ is a monotonic operator, i.e. if $O_1 \subseteq O_2$ then $\mu(O_1) \subseteq \mu(O_2)$. Hence, by the Tarski-Knaster theorem, it has a least fixpoint denoted by lfp(O) called the* closure *of O.*

Example 3.20. *Let O be the aRDF database in Figure 3.1(b). Besides the triples in O, lfp(O) also contains $(Flu, associatedWith : .65, Pneumonia)$, $(Flu, hasComplication : .1, Emphysema)$, $(Flu, hasComplication : .001, CorPulmonale)$.*

The following result is a necessary and sufficient condition for entailment by an aRDF database.

Proposition 3.21. *Let O be an aRDF database. $O \models (r, p : a, v)$ if and only if at least one of the following conditions holds:*

1. *$(r, p : a, v) \in$ lfp(O) OR*
2. *$\exists (r', p' : a', v') \in$ lfp(O) s.t. $\{(r', p' : a', v')\} \models (r, p : a, v)$. The second condition avoids redundancy by requiring that from all annotated triples with the same subject, property and value, only the one with the maximal (with respect to $\preceq$) annotations are present in the closure.*

Proof. Follows from Definition 3.19, which corresponds to the conditions in Theorem 3.18. Here, $\mu(O)$ has been defined to be the set of triples that can be

inferred by conditions (E1),(E2) of Theorem 3.18 in exactly one step (i.e., considering only pairs of triples). $\mu(O)$ is then augmented at every step until any triple entailed by O is either contained in $\mathsf{lfp(O)}$ or trivially entailed by a triple in the fixpoint. $\square$

The above proposition provides an immediate algorithm to answer ground atomic queries to an aRDF database. The next proposition will provide us with a mechanism to answer atomic queries.

Proposition 3.22. *Let O be a consistent aRDF database and q a query on O. Then the following hold:*

1. *(Soundness) $Ans_q(O) \subseteq$ lfp(O).*
2. *(Completeness) For all substitutions θ such that $q\theta$ is ground and $O \models q\theta$, $Ans_q(O) \models q\theta$.*

Proof. From Proposition 3.21, we know that all possible answers to a query must either be in $\mathsf{lfp(O)}$ or entailed by a triple in $\mathsf{lfp(O)}$. Therefore, $\forall (r, p : a, v) \in Ans_q(O) - \mathsf{lfp(O)}$, exists $(r, p' : a', v) \in \mathsf{lfp(O)}$ s.t. $\{(r, p' : a', v)\} \models (r, p : a, v)$. On one hand, $(r, p' : a', v) \notin Ans_q(O)$ since $Ans_q(O)$ is maximal w.r.t. entailment. On the other hand, by Definition 3.16, $(r, p : a, v) \in Ans_q(O) \Rightarrow (r, p' : a', v) \in Ans_q(O)$ should also be in the answer. We have a contradiction, therefore, $Ans_q(O) \subseteq \mathsf{lfp(O)}$. Intuitively, the closure $\mathsf{lfp(O)}$ is minimal set of triples entailed by O w.r.t. entailment within the set μO.

The completeness of $Ans_q(O)$ results directly from the definition of a query answer. Since $O \models q\theta$, then the ground triple $q\theta$ must be in $A_q(O)$ (Definition 3.16). From the definition of $Ans_q(O)$, $q\theta$ is either in $Ans_q(O)$ or entailed by it. $\square$

The above proposition gives us a very simple algorithm for answering simple queries (which we will call *naiveSimpleAnswer*). Recall that simple queries are atomic queries with at most one variable in it.

1. Consider query $q = (r, p : a, v)$ on aRDF database O. Compute lfp(O).

2. $A \leftarrow \{(r', p' : a', v') \in$ lfp(O)$|(r', p' : a', v')$ semi-unifiable with $q \wedge((a$ is a variable $) \vee (a \preceq a'))\}$.

3. Eliminate from A triples $(r, p : a, v)$ entailed by subsets of $A - \{(r, p : a, v)\}$.

However, we can do much better by avoiding the costly computation of lfp(O), as we will show in Chapter 4.

3.4.2 Conjunctive queries

Conjunctive queries are simply sets of simple aRDF queries that have common variables.

Definition 3.23 (Conjunctive query)**.** *A conjunctive query Q is a set of simple queries such that for any simple query $q \in Q$, there exists a variable v in q that also appears in another simple query $q' \in Q, q' \neq q$.*

Note that a conjunctive query is basically a partially instantiated aRDF graph. The condition imposed on the query is that the corresponding query graph is *connected*. This requirement is best illustrated with an example.

Example 3.24. *Consider the aRDF database in Figure 3.1(c). The following is a conjunctive query over this database: which Italian restaurants located in the NE*

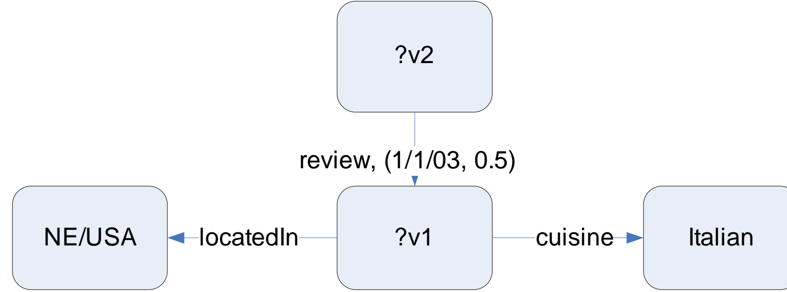


Figure 3.3: Example aRDF conjunctive query graph

USA had a review with an annotation value a such that $(1/1/2003, .5) \preceq a$*? Since we are using less-than for the partial order of both dates and review scores, this means we are looking for Italian restaurants in the NE USA with reviews newer than January 1 2003 and review score higher than .5. The query can be expressed as* $Q = \{$*(?v1, cuisine, Italian), (?v1, locatedIn, NE/USA), (?v2, review: (1/1/03, .5), ?v1)*$\}$*. A graphical representation of the query is given in Figure 3.3. We remind the reader that triples not annotated in this example are assumed to be annotated with* $(now, 1)$ *and that locatedIn is a transitive property. Note that the natural language version of the query also asks for a projection operation – we only want the restaurants, hence the possible substitutions for ?v1. Since projection can be easily performed once the set of possible substitutions for ?v1 and ?v2 is computed, we do not explicitly define this operation in this dissertation.*

Note that this conjunctive query does meet the restriction in Definition 3.23, since ?v1 appears in all the simple queries. However, the query $\{$*(?v1, cuisine, Italian), (?v2, cuisine, Italian*$\}$ is not a valid conjunctive query since the two simple queries are not linked by any variable. This query can be simply decomposed in two atomic queries that can be answered independently.

We also point out that the aRDF conjunctive queries are very similar to SPARQL query patterns, albeit without the syntactic sugar. The query Q is in fact a SPARQL graph pattern and can be expressed as (annotations are omitted as they cannot be expressed in standard SPARQL):

SELECT ?v1, ?v2 FROM ChefMoz WHERE

{(?v1 locatedIn NE/USA) . (?v1 cuisine Italian) .

(?v1 review ?v2)}

The SPARQL query language is based on the concept of *graph patterns*, which correspond directly to aRDF conjunctive queries. The following features of SPARQL are not included in our discussion of aRDF queries.

1. *OPTIONAL graph patterns* in SPARQL specify parts of the graph pattern that can provide substitutions for variables in the query, but which need not be matched for the query to return an answer.

2. *UNION* queries define a set of patterns that can be answered independently. The resulting substitutions can then be unioned to give the query answer. UNION and OPTIONAL patterns appeared in less than 3% of the queries that were publicly available at `www.rdfdata.org`.

3. *FILTER* constructs allow conditions on the range of values the variables take.

Union and optional patterns are easy to treat by decomposing them into conjunctive queries that can be answered independently. Filter conditions can be applied after a conjunctive query has been processed.

Definition 3.25 (Conjunctive query answer)**.** *Let* $Q = \{q_1, \ldots, q_n\}$ *be a conjunctive* aRDF *query. Let* $\Theta_O(Q) = \{(\theta_1, \ldots, \theta_n) \in \Theta_O(q_1) \times \ldots \times \Theta_O(q_n) | \theta_1 \cup \ldots \cup \theta_n$ *is consistent*[12]$\}$*. We define* $A_O(Q) = \{\{e_1, \ldots, e_n\} | \exists\ (\theta_1, \ldots, \theta_n) \in \Theta_O(Q)$ *s.t.* $\forall\ i \in [1, n], e_i = q_i\theta_i\}$*. The answer to* Q *is* $Ans_O(Q) = \{\{e_1, \ldots, e_n\} \in A_O(Q) | (A_O(Q) - \{\{e_1, \ldots, e_n\}\}) \not\models \{e_1, \ldots, e_n\}\}$*.*

For a conjunctive query Q, one element of the answer is a set of aRDF triples and each element of this latter set is an answer to one of the simple queries in Q. $Ans_O(Q)$ has the same purpose as for simple queries, namely to eliminate redundant elements from the answer.

Example 3.26. *Consider the query* Q *in Example 3.24. The* aRDF *database in Figure 3.1(c) contains two answers to this query (remember that locatedIn is a transitive property):*

- $\{$*(Grivanti, locatedIn, NE/USA), (Grivanti, cuisine, Italian), (Reviewer #21765, review: (4/07/04, .7), Grivanti)*$\}$
- $\{$*(Charlie's, locatedIn, NE/USA), (Charlie's, cuisine, Italian), (Reviewer #16742, review: (22/05/03, .5), Charlie's)*$\}$

Definition 3.25 also provides a naive method of answering a conjunctive query Q (which we will call *naiveConjunctiveAnswer*):

1. Compute the substitutions (answers) $\Theta_O(q)$ for each of the simple queries $q \in Q$.

[12] We assume, as is frequently done in the unification literature [41], that a substitution can be viewed as a system of equations and that a set of substitutions is compatible iff the union of the set of equations corresponding to each substitution is a solvable system of equations.

2. Compute the cartesian product $\Theta' = \Pi_{q \in Q} \Theta_Q(q)$.

3. Select only those elements in $(\theta_1, \ldots, \theta_n)$ for which $\theta_1 \cup \ldots \cup \theta_n$ is consistent.

4. Compute the set of answers $A_O(Q)$ by applying each remaining substitution in Θ' to Q.

5. Eliminate the redundant answers from $A_O(Q)$ to obtain $Ans_O(Q)$.

3.5 Summary

In this chapter, we have defined the formal syntax and semantics of Annotated RDF, which allows RDF triples to be annotated with values drawn from a finite or infinite partial order. We have define the notions of an aRDF database, aRDF interpretation and consistency. We have also identified a set of necessary and sufficient conditions for an aRDF database to be consistent and provided an algorithm that checks these conditions. We have shown that aRDF is capable of supporting diverse forms of reasoning as well as combinations (e.g., via `fuztime`) and has a rich declarative semantics. Finally, we have defined a language and semantics of queries over aRDF and showed their relationship to SPARQL graph patterns. In Chapter 4, we introduce a set of very efficient query algorithms for aRDF.

Chapter 4

Querying Annotated RDF

In Chapter 3, we have defined the formal syntax and semantics of Annotated RDF. In this chapter, we will introduce algorithms to query aRDF databases. We provide several algorithms for simple one-variable queries and two methods for SPARQL-like conjunctive queries. We also address the problem of view maintenance, in which we have to recompute the answer to a query when a triple is inserted or deleted. We conclude with an extensive evaluation of our query algorithms over two real-world datasets (ChefMoz and GovTrack) and a set of synthetically generated data. Our claim that aRDF can answer queries more efficiently than standard RDF is supported by comparisons with leading systems such as Jena2, Sesame2 and Oracle 11g. aRDF answers queries 1.5 to 3.5 times faster than these systems and scales very well for large selectivity queries and large query sizes.

4.1 aRDF Query Processing Algorithms

We develop the aRDF query processing algorithms incrementally, starting with the problem of atomic queries and generalizing to simple queries with more than one variable. We then give two distinct methods of answering conjunctive queries. The first is based on a subgraph matching algorithm, whereas the second answers each element in the conjunction separately and then heuristically determines a good

order for the necessary joins. In this chapter, we will continue to assume $\mathcal{A}$ is finite for clarity, but the extension to infinite partial orders in Section 3.3 still applies.

4.1.1 Answering atomic queries

Although the closure of an aRDF database gives a simple method of computing the answer to queries (as shown in Chapter 3), the computation of $\mathsf{lfp}(\mathsf{O})$ is potentially very expensive. In fact, we show that we can do much better by building only those parts of the closure that are of interest to the given query. We start by focusing on atomic queries – i.e., simple queries with only one variable. The algorithm for queries of type $q = (r, p : a, ?v)$ is given in Figure 4.1; computing the answers to atomic queries of type $q = (?r, p : a, v)$ is almost identical (with the proper notation change) and therefore is omitted.

Algorithm *atomicAnswerV* starts by analyzing all triples that are semi-unifiable with the query (lines 3–9) if the property p_q in the atomic query is non-transitive. For all triples with the same subject, property and object (but different annotations), *atomicAnswerV* determines the greatest lower bound of their annotations (lines 5–7). If p_q is transitive, then *atomicAnswerV* iterates through all the p_q-paths starting at the subject r_q of the query. For each such p_q-path, the algorithm determines the greatest lower bound of the set of annotations on that path and forms the corresponding answer (lines 12–15).

Example 4.1. *Consider the aRDF graph in Figure 3.1(b) and the query (Flu, associatedWith: 0.1, ?v). Since associatedWith is transitive, the algorithm will go*

Algorithm atomicAnswerV(O,$\mathcal{A}$, $\preceq$, q)

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $q = (r_q, p_q : a_q, ?v)$.

Output: $Ans_O(q)$.

1: $O \leftarrow O|_{p_q}$
2: $Ans \leftarrow \emptyset$
3: **if** p_q is non-transitive **then**
4: **for** $(r_q, p', v') \in \{(r_q, p' : a', v') \in O\}$ **do**
5: $A \leftarrow \{a' \in \mathcal{A} | (r_q, p' : a', v') \in O\}$
6: $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
7: $C \leftarrow \{c \in B | \nexists c' \in B, c' \neq c \; s.t. \; c' \preceq c\}$
8: $Ans \leftarrow Ans \cup \{(r_q, p' : c, v') | c \in C \wedge a_q \preceq c\}$
9: **end for**
10: **else if** p_q transitive **then**
11: **for** all v' s.t. $\exists \; Q^1, \ldots, Q^k$ p-paths from r_q to v' **do**
12: $B \leftarrow \{b \in \mathcal{A} | \exists i \in [1, k] \; s.t. \; \forall \; a' \in \; A_{Q^i}, b \preceq a'\}$
13: $C \leftarrow \{c \in \mathcal{A} | \forall b \in B, b \preceq c\}$
14: $D \leftarrow \{d \in C | \; \nexists \; d' \in C, d' \neq d \; s.t. \; d' \preceq d\}$
15: $Ans \leftarrow Ans \cup \{(r_q, p_q : d, v') | d \in D \wedge a_q \preceq d\}$
16: **end for**
17: **end if**
18: **return** Ans

Figure 4.1: Answering atomic aRDF queries $(r_q, p_q : a_q, ?v)$

on the second branch, starting at line 10. The loop on line 11 iterates through all the values reachable through associatedWith-paths from Flu, which are exactly $\{$*AcuteBronchitis, Pneumonia, Emphysema, CorPulmonale*$\}$. *Let us consider the second iteration, where* $v' = Pneumonia$. *There are two associatedWith paths between Flu and Pneumonia. For the first path, going through AcuteBronchitis,* $A_(Q^1) = \{.7, .65\}$. *For the second, direct path,* $A_{Q^2} = \{.15\}$. *Therefore B on line 12 is exactly the interval* $(0, .65]$ *and* $C = [.65, 1]$. *As a result, on line 14,* $D = \{.65\}$ *and the triple (Flu, associatedWith: .65, Pneumonia) is added to the answer.*

The following theorem states that *atomicAnswerV* is correct and runs in polynomial time.

Proposition 4.2. *Let O be an* aRDF *graph and let n be the number of vertices in the graph O, let $e = |O|$ and let $p = |\mathcal{P}|$. Let $(\mathcal{A}, \preceq)$ be a partial order and let $a = |\mathcal{A}|$. Then the following hold:*

1. *atomicAnswerV$(O, \mathcal{A}, \preceq, q)$ returns $Ans_O(q)$.*
2. *atomicAnswerV$(O, \mathcal{A}, \preceq, q)$ is $\mathcal{O}(n^2 \cdot e + n \cdot e \cdot a^2)$.*

Proof. Algorithm correctness follows directly from Theorem 3.18. Lines 3—9 correspond to condition (E1), whereas lines 10—17 correspond to condition (E2).

The complexity result is given by the loop on lines 10—17. We start by determining all values reachable by p_q-paths from r_q and the paths themselves. This process takes at most $\mathcal{O}(n^2 \cdot e)$ steps. Since there are at most $\mathcal{O}(n)$ p_q-paths originating from r_q, each with at most $O(e)$ edges and the annotation for each path is bounded by a, line 12 will be run at most $\mathcal{O}(n \cdot e \cdot a^2)$ times. Since the sizes of B, C, D are all bounded by a, the same result holds for lines 13—15. □

Algorithm atomicAnswerP$(O, \mathcal{A}, \preceq, q)$

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $q = (r_q, ?p : a_q, v_q)$.

Output: $Ans_O(q)$.

1: $Ans \leftarrow \{(r_q, p : a, v_q) | a_q \preceq a\}$
2: **for** all p' such that $\exists\ Q^1, \ldots, Q^k$ p'-paths from r_q to v_q **do**
3: $\quad B \leftarrow \{b \in \mathcal{A} | \exists i \in [1, k]\ s.t.\ \forall\ a' \in\ A_{Q^i}, b \preceq a'\}$
4: $\quad C \leftarrow \{c \in \mathcal{A} | \forall b \in B, b \preceq c\}$
5: $\quad D \leftarrow \{d \in C |\ \nexists\ d' \in C, d' \neq d\ s.t.\ d' \preceq d\}$
6: $\quad Ans \leftarrow Ans \cup \{(r_q, p' : d, v_q) | d \in D \wedge a_q \preceq d\}$
7: **end for**
8: **return** $\{(r', p' : a', v') \in Ans |\ \nexists\ S \subseteq Ans - \{(r', p' : a', v')\}\ s.t.\ S \models (r', p' : a', v')\}$

Figure 4.2: Answering atomic aRDF queries $(r_q, ?p : a_q, v_q)$

An even tighter complexity bound holds when the annotation is a complete

lattice. In this case, after computing the set A on line 11, we can simply compute the least upper bound of the elements in A and thus obtain set C (on line 13). For complete lattices such as $\mathcal{A}_{time-int}$, this can be done in at most a linear number of steps in $|A|$. Thus, the overall complexity of the algorithm becomes $\mathcal{O}(n^2 \cdot e + n \cdot e \cdot a)$.

Algorithm *atomicAnswerP* given in Figure 4.2 computes the answer to atomic queries with an unknown property. The algorithm iterates through all p-paths between the subject r_q and the object v_q of the query. For each such p-path, it computes the set of annotations on the path and determines its greatest lower bound (lines 3–5). It then uses the property on the p-path, the greatest lower bound computed and the subject r_q and value v_q of the query to form an answer (line 6). Finally, *atomicAnswerP* eliminates any redundant triples on line 8.

The main difference between *atomicAnswerP* and *atomicAnswerV* is that the graph we need to explore is the one containing all paths between r and v, instead of the one containing all p-paths starting at r. Depending on the shape of the aRDF database (e.g., breadth vs. depth), either search space may be larger, but the worst case complexity is identical.

Proposition 4.3. *Let O be an aRDF graph and let n be the number of vertices in the graph O, let $e = |O|$ and let $p = |\mathcal{P}|$. Let $(\mathcal{A}, \preceq)$ be a partial order and let $a = |\mathcal{A}|$. Then the following hold:*

1. *$atomicAnswerP(O, \mathcal{A}, \preceq, q)$ returns $Ans_O(q)$.*

2. *$atomicAnswerP(O, \mathcal{A}, \preceq, q)$ is $\mathcal{O}(n^2 \cdot e + n \cdot e \cdot a^2)$.*

Proof. The correctness of the algorithm again follows from Theorem 3.18.

Algorithm atomicAnswerA(O,$\mathcal{A}$, $\preceq$, q)

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $q = (r_q, p_q : ?a, v_q)$.

Output: $Ans_O(q)$.

1: $O \leftarrow O|_{p_q}$
2: $Ans \leftarrow \emptyset$
3: **if** p_q is non-transitive **then**
4:     **for** $(r_q, p', v_q) \in \{(r_q, p' : a', v_q) \in O | p' \in SP(p_q)\}$ **do**
5:         $A \leftarrow \{a' \in \mathcal{A} | (r_q, p' : a', v_q) \in O\}$
6:         $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
7:         $C \leftarrow \{c \in B | \nexists c' \in B, c' \neq c \ s.t. \ c' \preceq c\}$
8:         $Ans \leftarrow Ans \cup \{(r_q, p' : c, v_q) | c \in C\}$
9:     **end for**
10: **else if** p_q transitive **then**
11:     $\{Q^1, \ldots, Q^k\} \leftarrow \{p\text{-paths from } r_q \text{ to } v_q\}$
12:     $B \leftarrow \{b \in \mathcal{A} | \exists i \in [1, k] \ s.t. \ \forall \ a' \in \ A_{Q^i}, b \preceq a'\}$
13:     $C \leftarrow \{c \in \mathcal{A} | \forall b \in B, b \preceq c\}$
14:     $D \leftarrow \{d \in C | \nexists \ d' \in C, d' \neq d \ s.t. \ d' \preceq d\}$
15:     $Ans \leftarrow Ans \cup \{(r_q, p_q : d, v_q) | d \in D\}$
16: **end if**
17: **return** Ans

Figure 4.3: Answering atomic aRDF queries $(r_q, p_q : ?a, v_q)$

The case of non-transitive properties in handled directly in the initialization of Ans on line 1. Similarly to algorithm *atomicAnswerV*, lines 2–7 handle case (E2) of Theorem 3.18.

The complexity follows from the loop in lines 2–7 of the algorithm. Computing all paths between r_q and v_q takes $\mathcal{O}(n^2 \cdot e)$ iterations, and there are at most $\mathcal{O}(n)$ paths (in the worst case, a path that passes through every vertex in the graph different from r_q and v_q). Each path has less than $\mathcal{O}(e)$ edges, hence we obtain the same complexity result as for *atomicAnswerV*. □

Algorithm *atomicAnswerA* given in Figure 4.3 computes the answer to atomic queries with unknown annotation. Since the subject, property and object of the

query are all known, $atomicAnswerA$ only needs to determine the greatest lower bound for the set of annotations on triples with the given subject, property and object. It does so for non-transitive properties in lines 3–9 and for transitive properties on lines 10–16. Since the subject, property and object are all known, the algorithm has lower complexity than its two counterparts.

Proposition 4.4. *Let O be an aRDF graph and let n be the number of vertices in the graph O, let $e = |O|$ and let $p = |\mathcal{P}|$. Let $(\mathcal{A}, \preceq)$ be a partial order and let $a = |\mathcal{A}|$. Then the following hold:*

1. *$atomicAnswerA(O, \mathcal{A}, \preceq, q)$ returns $Ans_O(q)$.*
2. *$atomicAnswerA(O, \mathcal{A}, \preceq, q)$ is $\mathcal{O}(n \cdot e \cdot a^2)$.*

Proof. The correctness of the algorithm also follows from Theorem 3.18 - the two branches of the conditional on line 3 correspond to cases (E1) and (E2) respectively. Since we now know the resource, property and value in the query, the step in which we compute all paths (line 11) can be performed in at most $\mathcal{O}(n \cdot e)$ steps. This means that, similarly to the previous two atomic answer algorithms, the complexity of $atomicAnswerA$ is $\mathcal{O}(n \cdot e + n \cdot e \cdot a^2) = \mathcal{O}(n \cdot e \cdot a^2)$. □

4.1.2 Simple non-atomic queries

In the previous section, we have defined algorithms that compute the answer to atomic queries in polynomial time, by avoiding the expensive computation of lfp(O). The common trait of all $atomicAnswerX$ (where X is one of V, P, A) is that

they compute a part of $\mathsf{lfp(O)}$ localized to the subset of the database that contains the answer to the query. We extend this approach to simple non-atomic queries.

Algorithm answerSV(O,$\mathcal{A}$, $\preceq$, q)

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $q = (?r, p_q : a_q, ?v)$.

Output: $Ans_O(q)$.

1: $O \leftarrow O|_{p_q}$
2: $Ans \leftarrow \emptyset$
3: **if** p_q is non-transitive **then**
4:     **for** $(r', p', v') \in \{(r', p' : a, v') \in O\}$ **do**
5:         $A \leftarrow \{a \in \mathcal{A} | (r', p' : a, v') \in O\}$
6:         $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
7:         $C \leftarrow \{c \in B | \not\exists c' \in B, c' \neq c \ s.t. \ c' \preceq c\}$
8:         $Ans \leftarrow Ans \cup \{(r', p' : c, v') | c \in C \wedge a_q \preceq c\}$
9:     **end for**
10: **else if** p_q transitive **then**
11:     **for** all r', v' s.t. $\exists\ Q^1, \ldots, Q^k$ p_q-paths from r' to v' **do**
12:         $B \leftarrow \{b \in \mathcal{A} | \exists i \in [1, k] \ s.t. \ \forall\ a' \in\ A_{Q^i}, b \preceq a'\}$
13:         $C \leftarrow \{c \in \mathcal{A} | \forall b \in B, b \preceq c\}$
14:         $D \leftarrow \{d \in C | \not\exists\ d' \in C, d' \neq d \ s.t. \ d' \preceq d\}$
15:         $Ans \leftarrow Ans \cup \{(r', p_q : d, v') | d \in D \wedge a_q \preceq d\}$
16:     **end for**
17: **end if**
18: **return** Ans

Figure 4.4: Answering simple aRDF query $(?r, p_q : a_q, ?v)$

As an example, we show an algorithm for a simple non-atomic query in which the subject and value are both variables, i.e., a simple query of the form $(?r, p : a, ?v)$. The algorithm in Figure 4.4 is an adaptation of *atomicAnswerS* and *atomicAnswerV* to answer such queries. There are two main differences between *answerSV* and *atomicAnswerV*:

1. In *answerSV*, line 4 now iterates through all the (r, p, v) combinations in $O|_{p_q}$, whereas in *atomicAnswerV* it would iterate through all the (r_q, p, v) combinations since r_q was a constant.

2. In $answerSV$, line 11 iterates through all the pairs r', v' that have a p_q-path between them, whereas in $atomicAnswerV$ it only iterates through the vertices v' that r_q has a p_q-path to.

Algorithm $answerSV$ iterates through all subject-object pairs that are connected through the query property p_q. If p_q is transitive, then these pairs are obtained by iterating through all p_q-paths in the database (the loop on line 11). If p_q is non-transitive, these pairs are obtained by looking at all triples containing p_q as the property (loop on line 4). In either case, we consider the set of annotations on the triples connecting the subject and object values found and determine the greatest lower bound of this set (lines 5–8 for non-transitive p_q and lines 12–15 for transitive p_q).

Proposition 4.5. *Let O be an aRDF graph and let n be the number of vertices in the graph O, let $e = |O|$ and let $p = |\mathcal{P}|$. Let $(\mathcal{A}, \preceq)$ be a partial order and let $a = |\mathcal{A}|$. Then the following hold:*

1. $answerSV(O, \mathcal{A}, \preceq, q)$ *returns* $Ans_O(q)$.

2. $answerSV(O, \mathcal{A}, \preceq, q)$ *is* $\mathcal{O}(n^3 \cdot e + n^2 \cdot e \cdot a^2)$.

Proof The correctness of the algorithm follows from Theorem 3.18. Similarly to the proof of Proposition 4.2, in lines 4–9 we include in the answer all aRDF triples on the property p_q that match condition (E1) of Theorem 3.18 and on lines 10–16 we analyze all aRDF p_q paths in the aRDF database that match condition (E2) of Theorem 3.18. As seen before, all triples entailed by the aRDF database fall into

one of the above-mentioned categories, therefore the algorithm returns all possible answers.

In terms of complexity, note that we need $(O)(n^3 \cdot e)$ steps to compute the paths on line 11. There are now at most n^2 p_q-paths considered on line 11, which means line 12 will be run at most $\mathcal{O}(n^2 \cdot e \cdot a^2)$. The overall complexity of the algorithm will therefore be $\mathcal{O}(n^3 \cdot e + n^2 \cdot e \cdot a^2)$. Note that a complexity of this form is intuitively what we expect, since the difference between *atomicAnswerV* and *answerSV* is that the subject of the query becomes variable. Therefore, we can obtain *answerSV* from *atomicAnswerV* by adding an enclosing loop that iterates through all possible values for $?r$. Indeed, the complexity for *answerSV* can by computed by multiplying the complexity value of *atomicAnswerV* by the number of resources n. The same process can be applied to answer any simple non-atomic query. Since the algorithms are very similar to their *atomicAnswer* counterparts, we omit their formal descriptions. □

4.1.3 Conjunctive queries

For simple queries, we clearly want to avoid the expensive computation of $\mathsf{lfp(O)}$ in *naiveSimpleAnswer*. For conjunctive queries, it is not immediately clear which of computing $\mathsf{lfp(O)}$ or computing the cartesian product $\Pi_{q \in Q} \Theta_Q(q)$ in step (2) of *naiveConjunctiveAnswer* is more computationally intensive. The comparison depends on both the aRDF database O and on the size of the conjunctive query Q (the number of variables). Therefore, we propose two distinct methods of answering

conjunctive queries:

Algorithm conjunctAnswer_GraphMatching$(O, \mathcal{A}, \preceq, Q)$

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $Q = \{q_i = (r_i, p_i : a_i, v_i) | i \in [1, m]\}$. G_Q is the graph representation of the query Q.

Output: $Ans_O(q)$.

1: $O \leftarrow$ lfp(O)
2: $Ans \leftarrow \emptyset$
3: **execute** graph matching algorithm on G_Q and O
4: **for** all matchings between G_Q and O **do**
5: $\quad ok \leftarrow true$
6: $\quad$ **for** $i \in [1, m]$ **do**
7: $\quad\quad (r, p : a, v) \leftarrow$ the triple in O matched to q_i
8: $\quad\quad$ **if** $\neg(a_i\ variable) \wedge \neg(a_i \preceq a)$ **then**
9: $\quad\quad\quad ok \leftarrow false$
10: $\quad\quad\quad$ **break**
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad$ **if** ok **then**
14: $\quad\quad Ans \leftarrow Ans \cup \{$ set of triples matched to $G_Q\}$
15: $\quad$ **end if**
16: **end for**
17: **return** Ans

Figure 4.5: Answering conjunctive aRDF queries through inexact graph matching

1. *conjunctAnswer_GraphMatching* is based on the observation that conjunctive queries are partially instantiated aRDF graphs. Therefore, inexact graph matching algorithms [27, 9] can be used to match the query graph against the aRDF graph. To obtain a correct answer, graph matching must be performed on the closure of the original database — therefore we must compute lfp(O).

2. *conjunctAnswer_Ordering* uses the efficient simple query answering algorithms to derive answers for the elements of the conjunctive query, thus avoiding the fixpoint computation. The algorithm uses a heuristic ordering of the elements in the conjunctive query to compute the smallest possible part of the

Algorithm onjunctAnswer_Ordering$(O, \mathcal{A}, \preceq, Q)$ ———

Input: Consistent aRDF database O, annotation $(\mathcal{A}, \preceq)$ and query $Q = \{q_i = (r_i, p_i : a_i, v_i) | i \in [1, m]\}$. For a simple query q, $card(q)$ represents an estimate of the cardinality of the answer for q.

Output: $Ans_O(q)$.

1: **construct** graph H_Q {Graph H_Q has a vertex for each component of the conjunctive query. The nodes for two components containing the same variable are linked through an edge labeled with that variable. Figure 4.7 contains an example graph.}
2: **while** there exists a cycle in H_Q **do**
3: **choose** q_i with the lowest $card(q_i)$ in the cycle
4: $q_j \leftarrow$ value q that maximizes $card(q)$ over the set $\{q | \exists (q, q_i) \in H_Q\}$
5: $H_Q \leftarrow H_Q - \{(q_i, q_j)\}$
6: **end while**
7: $L \leftarrow$ depth-first traversal of H_Q starting with the component q with the smallest $card(q)$ {L is a FIFO queue}
8: $\Theta \leftarrow \emptyset$
9: **while** $L \neq \emptyset$ **do**
10: $q \leftarrow dequeue(L)$
11: **compute** $\Theta_O(q)$
12: $\Theta \leftarrow \{(\theta, \theta') | \theta' \in \Theta \text{ and } \theta' \in \Theta_O(q)\}$
13: $\Theta \leftarrow \Theta - \{(\theta_1, \ldots, \theta_k) | \theta_1 \cup \ldots \cup \theta_k \text{ inconsistent }\}$
14: **end while**
15: **compute** $A_O(Q)$ and $Ans_O(Q)$ based on Θ
16: **return** Ans

Figure 4.6: Answering conjunctive aRDF by heuristic ordering of the component queries

cartesian product.

Algorithm *conjunctAnswer_GraphMatching* (Figure 4.5) starts by computing the closure lfp(O) on line 1. After lfp(O) is computed, inexact graph matchings [27, 9] are used to determine potential answers to the conjunctive query (line 3). Since graph matching algorithms cannot take the semantics of the aRDF annotations into account, we have to check the potential answer triple by triple against the query annotation (if constant) on lines 7–11. If all triples have "better" annotations (in terms of the $\preceq$ order) than the corresponding query triples, the answer

is stored (line 14). The complexity of *conjunctAnswer_GraphMatching* is $\mathcal{O}(n!)$ in the worst case, since graph matching algorithms are factorial in the size of the graph [9]. However, we have determined experimentally that the average complexity is close to polynomial in the size of the database and in the size of the query.

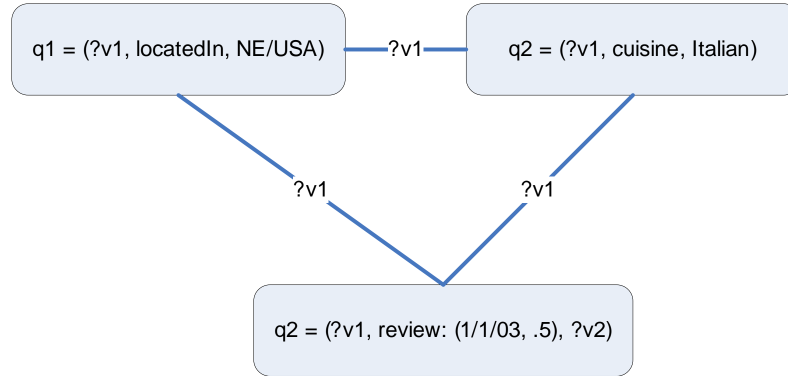


Figure 4.7: Example H_Q graph

Algorithm *conjunctAnswer_Ordering* (Figure 4.6) starts by creating a partial order of the component queries in the conjunctive query Q (lines 1—6). The partial order indicates which parts of the cartesian product should be computed first in order to minimize the number of operations. The process is similar to that of determining the ordering of joins in a relational databases. To create a partial order for the component queries, we create an undirected labeled graph H_Q (line 1) as follows:

- Each $q_i \in Q$ is a vertex in H_Q. There are no other vertices in H_Q.
- There exists an edge between q_i and q_j labeled with $?v$ iff there exists a variable $?v$ that appears in both q_i and q_j.

The resulting H_Q graph may contain cycles for certain queries. For instance, the conjunctive query $Q = \{q_1 = (?v1, associatedWith : .65, ?v2), q_2 = (?v2, associatedWith : .4, ?v1)\}$ results in a graph with two edges, both between q_1 and q_2, labeled with $?v1$ and $?v2$ respectively. In such cases, it is not clear which of q_1, q_2 should be executed first. To break cycles, we use an estimate of the cardinality of each component query based on a very recent method [38] that uses a pattern-based summarization framework to estimate the cardinality of RDF graph patterns. The method uses minimal overhead, especially since we only estimate cardinalities for simple queries. Based on the cardinality estimation, we break cycles as follows:

1. For each cycle, we choose the node $q_i \in H_Q$ with the lowest cardinality.
2. Let q_j be the neighbor of q_i with the highest estimated cardinality.
3. We remove the edges between q_i and q_j.

Intuitively, we remove those edges (lines 2—6) that would not allow a query with a low estimated cardinality to be executed before other queries with higher cardinality estimates. The cycle-free graph thus obtained is a partial order of the component queries that minimizes the number of cartesian product operations.

Finally, the algorithm takes the depth-first traversal of H_Q (line 8) and incrementally computes the cartesian product of the sets of substitutions for each component query (lines 12—14). We use a depth-first traversal rather than a breadth-first traversal to minimize the number of operations. A breadth-first traversal will compute cartesian products between the $\Theta_O(q_i)$ and $\Theta_O(q_j)$, even if q_i and q_j have no variables in common. This will produce a result of size $|\Theta_O(q_i)| \cdot |\Theta_O(q_j)|$ (since

there are no common variables). However, if we go depth-first, we will favor queries q_i, q_j with common variables and thus with a smaller size for the cartesian product $\Theta_O(q_i) \times \Theta_O(q_j)$.

We will illustrate the *conjunctAnswer_Ordering* through an example.

Example 4.6. *Consider the query in Figure 3.3 on the* aRDF *database in Figure 3.1(c). The* H_Q *graph for this conjunctive query is shown in Figure 4.7. Since there is a cycle, the algorithm will enter the loop on line 2. In this case, the cardinality estimation method will most likely give lower cardinalities to* q_1 *and* q_2*, which contain a single variable. Let us assume that* q_2 *has the lowest cardinality and* q_3 *the highest. On line 5, the edge between* q_2 *and* q_3 *will be deleted and the depth-first traversal on line 8 will yield* $L = \{q_2, q_1, q_3\}$*. On line 11, we will remove* q_2 *from the queue and compute (line 12)* $\Theta_O(q_2) = \{?v1 \leftarrow Charlie's, ?v1 \leftarrow Grivanti\}$*.* $\Theta = \Theta_O(q_2)$*. At the next step, we will remove* q_1 *from the queue and obtain the same set of substitutions, which means* Θ *will not change. Finally, on the third iteration,* $\Theta_O(q_3) = \{$*(?v1* $\leftarrow$ *Charlie's, ?v2* $\leftarrow$ *Reviewer #21765), (?v1* $\leftarrow$ *Grivanti, ?v2* $\leftarrow$ *Reviewer #21765), (?v1* $\leftarrow$ *Charlie's, ?v2* $\leftarrow$ *Reviewer #16742)*$\}$*. After computing the cartesian product and removing inconsistent substitutions,* $\Theta = \Theta_O(q_3)$*.*

Proposition 4.7 (Correctness)**.** *The algorithm conjunctAnswer_Ordering*$(O, \mathcal{A}, \preceq, Q)$ *is correct, i.e., it terminates and returns* $Ans_O(Q)$*.*

Proof. The loop on lines 2—6 will always terminate since we are removing at least one edge from H_Q at each iteration. This implies that eventually H_Q will be cycle free. The loop on line 10–15 is on the finite set L. Note that lines 10 – 16

correspond to the process described in Definition 3.25. Since we are computing a cartesian product, the particular order that we choose in lines 1—6 does not affect the correctness of the result, only the computation time. □

Proposition 4.8 (Complexity)**.** *The worst time complexity of conjunctAnswer_Ordering is* $\mathcal{O}((n^2 \cdot p)^{|Q|})$*, where* $n = |\mathcal{R}|$*,* $p = |\mathcal{P}|$ *and* $|Q|$ *is the number of simple queries in* Q*.*

Proof. For each component query, the worst case cardinality of the answer is $\mathcal{O}(n^2 \cdot p)$. Since we have p such simple queries, the worst-case cartesian product is $\mathcal{O}((n^2 \cdot p)^{|Q|})$. However, our experimental results show that in practice the cardinality of the answer of each component query is linear in the size of the database (with a very low factor) even for queries with high selectivity. □

4.2 aRDF View Maintenance

In this section, we explore solutions to the aRDF *view maintenance* problem. Suppose a query q is often posed by users. It then becomes efficient to store the results of q and if possible avoid the expensive re-computation of $Ans_O(q)$ by incrementally updating the result when the underlying aRDF database changes. Views are omnipresent in databases, and there is a large literature on them summarized by Gupta and Mumick [21]. The queries q defining a view can be used to express conditions that users want to track.

In order to maintain aRDF views, we require that an additional data structure called the *path annotation function* be stored with the aRDF database. This new

data structure exploits the fact that, as seen in the atomic answer algorithms, we are interested mainly in the annotations on a p-path rather than the actual vertices on the path. In the following sections, we will explore in detail how the *path annotation function* can be computed, and how we can incrementally check consistency on updates. For the rest of the section, let $q = (r_q, p_q : a_q, v_q)$ be a simple query and let $R = Ans_O(q)$ be the stored answer to q. We will present our view maintenance algorithms for simple queries. Conjunctive queries can be easily maintained in the following way:

1. Whenever a conjunctive query view is created, create and store an answer set for all of the simple queries that are part of the conjunction.

2. In case of insertions and deletions, execute the view maintenance algorithms in this section on each conjunction component.

3. Use *conjunctAnswer_Ordering* to obtain an answer to the conjunctive view based on the answers of the conjunction components.

There are two main challenges in incrementally updating views:

1. Check the aRDF database consistency incrementally when the database changes through insertions and/or deletions.

2. Re-compute the answers to the queries incrementally.

4.2.1 Path Annotation Function

In order to recompute path annotations quickly when the aRDF database changes, we need to maintain an additional data structure called the *path annotation function*. We point out that in all the *atomicAnswer* and *answer* algorithms, we are only interested in the sets of annotations on each path and not the actual resources on the path. Therefore, we only need store subsets of $\mathcal{A}$ that annotate p-paths in the aRDF database to quickly re-compute query answers when the database changes.

Definition 4.9 (Path annotation function)**.** *Let q be a simple query. The path annotation function δ for q is a function $\delta : Ans_O(q) \rightarrow 2^{2^{\mathcal{A}}}$ such that $\forall (r, p : a, v) \in R$, P is a p-path between r and v iff $A_P \in \delta(r, p : a, v)$, where A_P is the set of annotations for the triples on the path P.*

In short, a path annotation function maps elements of an answer set to a set containing sets of annotations. Each element $(r, p : a, v)$ is mapped to a set X in which every element $A \in X$ is the set of annotations for a p-path between r and v.

Example 4.10. *Consider the aRDF database in Figure 3.1(b) and the query q=(Flu, associatedWith: .5, ?v). The answer to this query is $Ans_O(q)$ =$\{$(Flu, associatedWith : .65, Pneumonia), (Flu, hasComplication: .7, AcuteBronchitis)$\}$ and δ(Flu, associatedWith : .65, Pneumonia)=$\{\{$(.7, .65),(.15)$\}\}$.*

We should note a few important properties of the path annotation function:

1. $\delta(r, p : a, v)$ does not depend on the annotation a; more precisely, $\delta(r, p :$

....

10. **else if** p transitive **then**

11. **for** all v' s.t. $\exists\ Q^1, \ldots, Q^k$ p-paths from r to v' **do**

12'. $\delta(r, p, v') \leftarrow \{A_{Q^1}, \ldots, A_{Q^k}\}$;

12. $B \leftarrow \{b \in \mathcal{A} | \exists i \in [1, k]\ s.t.\ \forall\ a' \in\ A_{Q^i}, b \preceq a'\}$;

....

Figure 4.8: Computing the path annotation function

$a, v) = \delta(r, p : a', v)$, $\forall\ (r, p : a, v), (r, p : a', v) \in Ans_O(q)$. We will simply write $\delta(r, p, v)$ to denote the value of the path annotation function.

2. δ is a shared data structure. In other words, δ is not dependent on a particular query q. The path annotation function can be computed and stored either at system startup or incrementally as queries are being answered.

As an example of how to compute δ incrementally, Figure 4.8 shows how to do this during the *atomicAnswerV* algorithm. In the newly inserted line (12'), the value of δ for a triple that could be in the answer can be simply stored from what the algorithm has already computed.

4.2.2 Incremental Consistency Checking

In this section, we look at the problem of incremental consistency verification and answer re-computation when a new triple is inserted into the aRDF database. Let $(r_i, p_i : a_i, v_i)$ be the newly inserted triple. Of course, our goal is to avoid a full re-computation if possible.

Although the *aRDFconsistency* algorithm is quite efficient in practice, we

wonder whether we can do better by analyzing only a part of the aRDF database which is "close" to the newly inserted triple. Algorithm *aRDFconsistencyInsert* (Figure 4.9) accomplishes this in the following way. For a newly inserted triple $(r_i, p_i : a_i, v_i)$, the algorithm recomputes the set of annotations on triples with subject r_i, property p_i and object v_i (line 1). The algorithm then checks that this set of annotations has a least upper bound (line 1). Furthermore, for all new p-paths that are newly created after inserting the triple, *aRDFconsistencyInsert* computes the set of annotations for each p-path and checks that the set has a greatest lower bound (lines 9 –16).

Algorithm aRDFconsistencyInsert$(O, (r_i, p_i : a_i, v_i))$

Input: Consistent aRDF database O, newly inserted triple $(r_i, p_i : a_i, v_i)$.
Output: $True$ iff $O \cup \{(r_i, p_i : a_i, v_i)\}$ is consistent.

1: $A \leftarrow \{a \in \mathcal{A} | \exists (r_i, p_i : a, v_i) \in O\} \cup \{a_i\}$
2: **if** $\nexists\, a \in \mathcal{A}$ s.t. $\forall a' \in A, a' \preceq a$ **then**
3:     **return** $False$
4: **end if**
5: **for** $p \in \mathcal{P}$ transitive **do**
6:     $O' \leftarrow O|p$
7:     $O'' \leftarrow (O \cup \{(r_i, p_i : a_i, v_i)\})|p$
8:     $P \leftarrow \{paths\ Q \subseteq O'' |\ \nexists Q' \subseteq O'', Q' \supset Q\}$
9:     **for** r, r' connected by additional paths in O'' than in O' **do**
10:         $P' \leftarrow \{Q \in P | Q\ is\ a\ path\ between\ r, r'\}$
11:         $A \leftarrow \{A_Q | Q \in P'\}$
12:         $B \leftarrow \{b \in \mathcal{A} | \exists A_Q\ s.t.\ \forall a \in A_Q, b \preceq a\}$
13:         **if** $\nexists a \in \mathcal{A}$ s.t. $\forall b \in B, b \preceq a$ **then**
14:             **return** $False$
15:         **end if**
16:     **end for**
17: **end for**
18: **return** $True$

Figure 4.9: Incremental consistency verification for insertions

Example 4.11. *Consider the example* aRDF *database in Figure 3.1(b) and let the*

triple to be inserted be (Pneumonia, associatedWith: .25, CorPulmonale). The algorithm will determine on line 6 that Flu and CorPulmonale, as well as Pneumonia and CorPulmonale are linked together by new paths. Let us consider the step in which r = Flu, r' = CorPulmonale. The path annotations are recomputed in A to be $\{\{.7, .001\}, \{.15, .25\}\}$. *B will be computed to be the interval* $[0, .15]$ *and the condition on line 10 is clearly false. After verifying the remaining pair of newly connected resources, the algorithm will return True.*

Theorem 4.12. *Let O be a consistent aRDF database and let $(r_i, p_i : a_i, v_i)$ be an aRDF triple. Then $aRDFconsistencyInsert(O, (r_i, p_i : a_i, v_i))$ returns true iff $O \cup \{(r_i, p_i : a_i, v_i)\}$ is a consistent aRDF database.*

Proof *p_i is a non-transitive property.* According to condition (C1) of Theorem 3.8, $O \cup \{(r_i, p_i : a_i, v_i)\}$ is consistent if and only if the set of annotations on triples (r_i, p_i, v_i) has an upper bound. In lines 1–4, $aRDFconsistencyInsert$ will return $False$ if and only if the set does not have such a bound, hence the algorithm returns the correct answer in this case.

p_i is a transitive property. Since O is consistent, the only paths that could cause $O \cup \{(r_i, p_i : a_i, v_i)\}$ to be inconsistent according to condition (C2) of Theorem 3.8 are paths that contain the newly inserted $(r_i, p_i : a_i, v_i)$. In the loop on line 9, $aRDFconsistencyInsert$ iterates over all resources that have a new path between them created by the insertion of $(r_i, p_i : a_i, v_i)$. For all such paths, $aRDFconsistencyInsert$ performs the same checks in lines 10–15 as the $aRDFconsistency$ algorithm does on lines 12–14. $aRDFconsistencyInsert$ therefore returns $False$ if

and only if such paths do not verify condition (C2) of Theorem 3.8. □

In the worst case, the complexity of the $aRDFconsistencyInsert$ is the same as $aRDFconsistency$; however such a scenario requires that every edge in the theory including $(r_i, p_i : a_i, v_i)$ is on a p-path between the same two vertices. In the general case, $aRDFconsistencyInsert$ looks only at the strongly connected component of $O|_{p_i}$ that contains the newly inserted triple.

4.2.3 Insertions

In this section we address the problem of incrementally computing $R' = Ans_{O \cup \{(r_i, p_i : a_i, v_i)\}}(q)$ from R and δ. The algorithm $viewMaintenanceInsert$ show in Figure 4.10 performs this incremental computation. To keep the formal description as simple as possible, we assume that p_q is a constant; the cases in which p_q is variable are a straightforward extension. We will also assume that δ is updated accordingly after a successful insertion. This can be done while performing the view maintenance on insertion, in the same way as in Figure 4.8.

The algorithm starts by analyzing the cases in which the property of the triple to be inserted is non-transitive (lines 5—11). If this is the case, we have to recompute the sets of annotation on direct edges between r_i and v_i on the property p_i. If p_i is transitive, then we analyze three different cases for each element in the previous query answer R:

(1) If r_i has become connected through p_q paths to vertices $v \neq v_i$ (lines 16–19)

(2) If v_i has become connected through p_q paths to vertices $r \neq r_i$ (lines 22–25)

method addResult$(Res, A,\ r, p, v)$

1: $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
2: $C \leftarrow \{b \in B |\ \nexists b' \in B, b' \neq b\ s.t.\ b' \preceq b\}$
3: $Res \leftarrow Res - \{(r, p : a, v) \in R\}$
4: $Res \leftarrow Res \cup \{(r, p : c, v) | c \in C\}$

Algorithm viewMaintenanceInsert$(O,\ q,\ R,\ \delta, (r_i, p_i : a_i, v_i))$

Input: Consistent aRDF database O, query $q = (r_q, p_q : a_q, v_q)$, answer R, precomputed path annotation function δ and newly inserted triple $(r_i, p_i : a_i, v_i)$.
Output: $R' = Ans_{O \cup \{(r_i, p_i : a_i, v_i)\}}(q)$.

1: $R' \leftarrow R$
2: **if** p_q is transitive and $p_i \notin SP(p_q)$ **then**
3:     **return** R
4: **end if**
5: **if** p_i is not transitive **then**
6:     **if** $\exists a \in \mathcal{A}$ s.t. $(r_i, p_i : a, v_i) \in R$ **then**
7:         $A \leftarrow \{a' \in \mathcal{A} | (r, p_i : a', v_i) \in O\} \cup \{a_i\}$
8:         $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
9:         $C \leftarrow \{c \in B |\ \nexists c' \in B, c' \neq c\ s.t.\ c' \preceq c\}$
10:         $R' \leftarrow R - \{(r_i, p_i : a, v_i) \in R\} \cup \{(r_i, p_i : c, v_i) | c \in C \wedge a \preceq c\}$;
11:     **end if**
12: **else** $\{p_i$ is transitive$\}$
13:     **for** $(r, p : a, v) \in R$ **do**
14:         **for** $p' \in SP(p_i) \cap SP(p)$ **do**
15:             **if** (r_i, p_q, v) semi-unifiable with q **then**
16:                 **for** all p'-paths P from v_i to r **do**
17:                     $A \leftarrow \{b \preceq a_i | \exists S \in \delta(r, p, v)\ s.t.\ \forall a' \in S \cup A_P, b \preceq a'\}$
18:                     $addResult(R', A, r_i, p', v)$
19:                 **end for**
20:             **end if**
21:             **if** (r, p_q, v_i) semi-unifiable with q **then**
22:                 **for** all p'-paths P from v to r_i **do**
23:                     $A \leftarrow \{b \preceq a_i | \exists S \in \delta(r, p, v)\ s.t.\ \forall a' \in S \cup A_P, b \preceq a'\}$
24:                     $addResult(R', A, r, p', v_i)$
25:                 **end for**
26:                 **for** all p'-paths P_1 from r to r_i and P_2 from v_i to v **do**
27:                     $A \leftarrow \{b \preceq a_i | \exists S \in \delta(r, p, v)\ s.t.\ \forall a' \in S \cup A_{P_1} \cup A_{P_2}, b \preceq a'\}$
28:                     $addResult(R', A, r, p', v)$
29:                 **end for**
30:             **end if**
31:         **end for**
32:     **end for**
33:     **for** $r, v \in N(O) \times N(O)$ such that (r, p_q, v) semi-unifiable with q **do**
34:         **for** $p' \in SP(p_i) \cap SP(p)$ **do**
35:             **for** all p'-paths P_1 from r to r_i and P_2 from v_i to v **do**
36:                 $A \leftarrow \{a \preceq a_i | \forall a' \in A_{P_1} \cup A_{P_2}, a \preceq a'\}$
37:                 $addResult(R', A, r, p', v)$
38:             **end for**
39:         **end for**
40:     **end for**
41: **end if**
42: **return** R'

Figure 4.10: View maintenance for atomic queries for insertions

(3) If the new edge between r_i and v_i creates a new p_q path from r to v through r_i and v_i – in which case r and v will become p_q-connected (lines 26–29)

After this step, we recompute the annotations for the affected (or new) paths and update the result accordingly. So far, we have only analyzed updates to elements in the answer. In lines 33–40 we also analyze whether any new triples should be added to the answer (similar to case (3) above, but for resources r and v that do not belong to a triple $(r, p, v) \in R$).

Example 4.13. *Consider the example* aRDF *database in Figure 3.1(b) and let the triple to be inserted be (Pneumonia, associatedWith: .25, CorPulmonale). The query to be maintained is q=(Flu, associatedWith: .15, ?v) and the answer* ***before insertion*** *is* $Ans_O(q)$*={(Flu, associatedWith : .65, Pneumonia), (Flu, hasComplication: .7, AcuteBronchitis)}. Since associatedWith is transitive, the algorithm will follow the branch starting on line 12. R will not change until we reach line 26, where we find a new path linking Flu and CorPulomonale through Pneumonia. On line 27,* $A = [0, .15]$ *and we will add a new triple to the result: (Flu, associatedWith: .15, CorPulmonale).*

Theorem 4.14. *Let* O *be a consistent* aRDF *database and let* $(r_i, p_i : a_i, v_i)$ *be an* aRDF *triple such that* $O \cup \{(r_i, p_i : a_i, v_i)\}$ *is consistent. If* q *is a simple query and* δ *is the path annotation function, then* $viewMaintenanceInsert(O, q, Ans_O(q), \delta, (r_i, p_i : a_i, v_i))$ *returns* $Ans_{O\cup\{(r_i,p_i:a_i,v_i)\}}(q)$*.*

Proof. Let us assume that *viewMaintenanceInsert* does not return the correct answer. Then one of the following is true: (i) either the algorithm returns

a triple $(r, p : a, v)$ that is not an answer to the query or (ii) there exists an answer to the query that is not in the answer returned by the algorithm. We will examine each case in turn.

Let us assume that there exists a triple $(r, p : a, v)$ returned by the algorithm that is not an answer to q. $(r, p : a, v)$ may not be an answer to q for two reasons.

1. $(r, p : a, v)$ is not semi-unifiable with q. For non-transitive properties p, note that we are only returning triples $(r_i, p_i : c, v_i)$ on line 10 such that $(r_i, p_i : a, v_i)$ was already in the previous answer R (line 6), hence it was semi-unifiable with the query. For transitive properties, we only analyze triples that are semi-unifiable with q (lines 15, 21, 26, 33).

2. For queries with a constant annotation a_q, we have that $a_q \npreceq a$. That cannot be the case due to the conditions imposed on any triples added to the result on line 10 and through *addResult* on lines 17–18, 23–24, 27–28 and 36-37.

We have established that we cannot have a triple returned by the algorithm that is not an answer to the query. Let us assume that there exists an answer to the query q that will not be returned. Since we do not change parts of the answer that are not affected by the inserted triple, the answer we are missing must be related to $(r_i, p_i : a_i, v_i)$. There are several cases in which the inserted triple can affect the answer:

1. For non-transitive properties, it may either add a new element to the answer identical to the inserted triple or it may alter the annotation for an existing

answer with the resource, property and value (r_i, p_i, v_i). Both cases are handled in lines 5–11.

2. For transitive properties, $(r_i, p_i : a_i, v_i)$ can alter the existing paths in the answer R that are semi-unifiable with the query by pre-pending existing paths (handled in lines 16–19), appending to existing paths (lines 22–25) or simply connecting two existing portions of a path that were previously not connected (handled in lines 26–29). Finally, the newly inserted triple can create new paths that were not represented by any result in R (handled on lines 33–40).

□

We point out that the worst-case complexity of the view maintenance algorithms is the same as that of the corresponding query algorithms since in the worst case, all triples in the answer may be changed. If this happens, for all practical purposes view maintenance will rerun the query algorithm to recompute all answers. However, we show experimentally that in most cases, performing view maintenance is much faster than recomputing the entire query answer from scratch.

4.2.4 Deletions

Suppose now that we intend to delete the triple $(r_d, p_d : a_d, v_d)$ from O. We would first like to show that deletions do not affect the consistency of an aRDF database.

Theorem 4.15. *Let O be a consistent aRDF database and let $(r, p : a, v) \in O$ be an arbitrary triple. Then $O - \{(r, p : a, v)\}$ is aRDF consistent.*

Proof. Let I be a satisfying interpretation for O. We can easily prove that I satisfies O':

(i) I satisfies every triple in O implies that I satisfies any triple in $O' = O - \{(r, p : a, v)\}$.

(ii) For all transitive properties $p \in \mathcal{P}$ let P be the set of p-paths $Q = \{t_1, \ldots, t_k\}$ in O. The set of p-paths in O' is clearly a subset of the set of paths in O. We know for all $a \in \mathcal{A}$ such that $a \preceq a_i$ for all $1 \leq i \leq k$, it is the case that $a \preceq I(r_1, p, r_{k+1})$. That will clearly hold for a subset of the paths considered for O, hence it will hold for O'.

O' has a satisfying interpretation and is thus consistent.□

We present an algorithm for computing $R' = Ans_{O-\{(r_d, p_d : a_d, v_d)\}}(q)$ in Figure 4.11. We again assume that δ is updated accordingly after the deletion. The algorithm starts with a similar procedure as $viewMaintenanceInsert$ for non-transitive properties (lines 3 –9). For transitive properties, we simply look for p_q-paths in the answer that have been interrupted by the deletion (lines 10–13). We compute the new path annotations by removing from delta the annotations for all the interrupted paths (line 13) and recompute the values for the remaining path annotations (lines 14 – 16).

Example 4.16. *Consider the example aRDF database in Figure 3.1(b) and let the triple to be deleted be (AcuteBronchitis, associatedWith: .65, Pneumonia). The query to be maintained is q=(Flu, associatedWith: .25, ?v) and the answer* ***before insertion*** *is $Ans_O(q)$={(Flu, associatedWith : .65, Pneumonia), (Flu, hasCom-*

Algorithm viewMaintenanceDelete$(O, q, R, \delta, (r_d, p_d : a_d, v_d))$

Input: Consistent aRDF database O, query $q = (r_q, p_q : a_q, v_q)$, answer R, precomputed function δ and deleted triple $(r_d, p_d : a_d, v_d)$.

Output: $R' = Ans_{O \cup \{(r_d, p_d : a_d, v_d)\}}(q)$.

1: $R' \leftarrow R$
2: **if** p_d is not transitive **then**
3:   **if** $\exists a \in \mathcal{A}$ s.t. $(r_d, p_d : a, v_d) \in R$ **then**
4:     $A \leftarrow \{a' \in \mathcal{A} | (r, p' : a', v') \in O\} \cup \{a_i\}$
5:     $B \leftarrow \{b \in \mathcal{A} | \forall a \in A, a \preceq b\}$
6:     $C \leftarrow \{c \in B | \not\exists c' \in B, c' \neq c \ s.t. \ c' \preceq c\}$
7:     $R' \leftarrow R' - \{(r_d, p_d : a, v_d) \in R\} \cup \{(r, p' : c, v') | c \in C \wedge a \preceq c\}$
8:   **end if**
9: **else** $\{p_d$ is transitive$\}$
10:   **for** $(r, p : a, v) \in R$ **do**
11:     **if** $\exists S \in \delta(r, p, v)$ s.t. $p_d \in S$ **then**
12:       **for** P_1, P_2 p-paths between r, r_d and v_d, v respectively **do**
13:         $T \leftarrow \delta(r, p : a, v) - \{A_{P_1} \cup A_{P_2} \cup \{p_d\}\}$
14:         $A \leftarrow \{a' \in \mathcal{A} | \exists S \in T \ s.t. \ \forall a'' \in S, a' \preceq a''\}$
15:         $B \leftarrow \{b \in \mathcal{A} | \forall a' \in A, a' \preceq b\}$
16:         $C \leftarrow \{c \in B | (\not\exists c' \in B, c' \neq c \ s.t. \ c' \preceq c) \wedge ((a_q \preceq c) \vee (a_q \text{ variable}))\}$
17:         $R' \leftarrow R' - \{(r, p : a, v)\} \cup \{(r, p : c, v) | c \in C\}$
18:       **end for**
19:     **end if**
20:   **end for**
21: **end if**
22: **return** R'

Figure 4.11: View maintenance for atomic queries for deletions

plication: .7, AcuteBronchitis)$\}$. *Since associatedWith is transitive, the algorithm will follow the branch starting on line 9. We find that one of the paths between Flu and Pneumonia was interrupted, hence* $T = \{\{.15\}\}$ *on line 13. We recompute* $A = [0, .15]$, $B = [.15, 1]$ *and* $C = \emptyset$. *As a result, the triple (Flu, associatedWith : .65, Pneumonia) will be removed from the answer.*

Theorem 4.17. *Let* O *be a consistent aRDF database and let* $(r_d, p_d : a_d, v_d)$ *be an aRDF triple. If* q *is a simple query and* δ *is the path annotation function, then* $viewMaintenanceDelete(O, q, Ans_O(q), \delta, (r_d, p_d : a_d, v_d))$ *returns* $Ans_{O - \{(r_i, p_i : a_i, v_i)\}}(q)$.

Proof. Let us assume that *viewMaintenanceDelete* does not return the correct answer. This means that either (i) the algorithm returns a triple that is not an answer to q or (ii) there exists a triple that is an answer to q that is not returned.

Let $(r, p : a, v)$ be a triple returned by the algorithm that is not an answer to q. Then we are in one of the following cases:

1. $(r, p : a, v)$ is not semi-unifiable with q. Note that due to the conditions on lines 3 and 12, the triples we are adding to the answer will have a resource, property and value that are already in a triple in R. Therefore, $(r, p : a, v)$ must be semi-unifiable with q.

2. If a_q is constant, $a_q \not\preceq a$. This cannot be the case due to the annotation recomputation in lines 4–7 and 13–17.

Let us assume that there exists a triple that is an answer to q that is not returned by the algorithm. Clearly, all answers that are unaffected by $(r_d, p_d : a_d, v_d)$ will still be returned. In lines 2–8 we will re-compute the annotation (or remove any answers) if p_d is non-transitive. If p_d is transitive, we re-compute the annotations and remove all answers corresponding to paths "disconnected" by the deleted triple in lines 13–17.□

As was the case with insertion view maintenance, the complexity of *viewMaintenanceDelete* is also the same as that of the algorithm to compute $Ans_O(q)$ since in the worst case we may have to update the entire answer.

4.3 Experimental evaluation

The aRDF query, consistency check and view maintenance algorithms were implemented in 5,300 lines of Java code. The experiments were performed on an Intel Core2 Duo 3.0 GHz machine with 3GB of RAM, running openSuse 10.2. The aRDF datasets were stored in flat binary files on disk; running time for all algorithms includes disk I/O. We experimented on three distinct datasets. The *GovTrack* (`http://www.govtrack.us`) dataset consists of approximately 26 million RDF triples (1.5 GB), annotated with temporal intervals that specify the period of time during which a triple is considered "valid." Converting reified triples resulted in 12,340,576 aRDF triples. The ChefMoz (`http://chefmoz.org`) dataset consists of 802,371 RDF triples (approximately 220 MB) describing restaurant information, including review scores dates. We used the review information to annotate the dataset with $\mathcal{A}_{fuztime}$. This resulted in 549,781 aRDF triples.

Finally, to study the dependence of the query processing time on various features of the aRDF database, we also generated a synthetic dataset ranging from 10,000 to 10,000,000 aRDF triples. The number of corresponding RDF triples after reification is on average 1.65 times that of aRDF triples[1]. For each database size, we generated 15 independent random datasets using uniform distributions for the random generator. To make the dataset as close to real-world datasets as possible – based on our study of the previous two and other RDF datasets – we maintained the following characteristics constant during the generation process:

[1]We remind the reader that two additional vertices (a blank node for the statement and a vertex for the annotation) are added to the RDF graph after reification.

1. The number of properties $|\mathcal{P}|$ follows a Gaussian distribution around 0.5% of the size of the dataset, with a standard deviation of no more than 0.01% of the size of the dataset.

2. The number of transitive properties was held constant at 5% of the total number of properties.

3. The number of $rdfs : subPropertyOf$ relations was uniformly distributed between 10 and 20% of the number of properties.

Approximately 15 of the 35 datasets available at `www.rdfdata.org`, including GovTrack, provide access to logs of their most frequent queries. We chose 50 GovTrack queries with selectivity factors[2] between 3 and 25% uniformly at random. 91% of the frequent queries for GovTrack are within this interval. Unless otherwise specified, the running times reported are an average over all queries. We have also investigated all available query logs to determine typical query sizes and variables/constants ratios. The average query size was 25.6 query components, with a standard deviation of 7.4. Variable/constants ratio was typically between 15% and 25%. Based on this information, we generated random queries for the ChefMoz and synthetic datasets (50 atomic and 50 conjunctive queries each) using the following criteria: (i) varied selectivity (uniformly distributed) between 3 and 25%; (ii) for conjunctive queries, the number of components in the conjunction was varied between 5 and 50 elements; (iii) the number of variables in the query was varied between 10 and 35% of the total number of subject, property, object and annotation

[2]The selectivity factor of a query is the percentage of triples it returns as an answer.

elements in each query.

In our experimental evaluation, we were interested in studying the following:

1. The consistency checking time on the GovTrack and ChefMoz datasets and its variation with the size of the synthetic dataset.

2. A comparison of the query processing time (including the naive algorithm) for all types of atomic queries and its variation with the size of the aRDF database.

3. The comparison between the running times of *conjunctAnswer_GraphMatching*, *conjunctAnswer_Ordering* and the naive algorithm and their variations with the size of the query and that of the database.

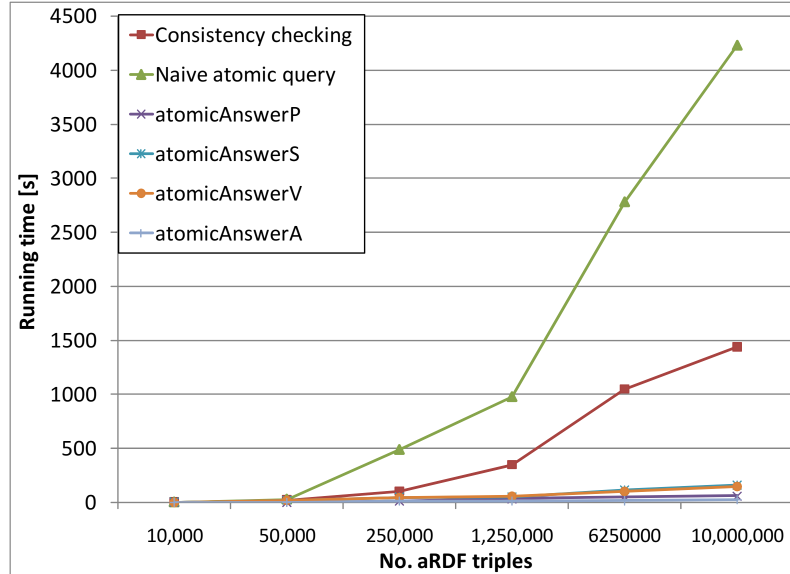
4. A comparative evaluation of view maintenance time versus re-running the entire query.

5. A comparison between aRDF, Jena2, Sesame2 and Oracle 11g in terms of query performance at various data sizes and query selectivity factors.

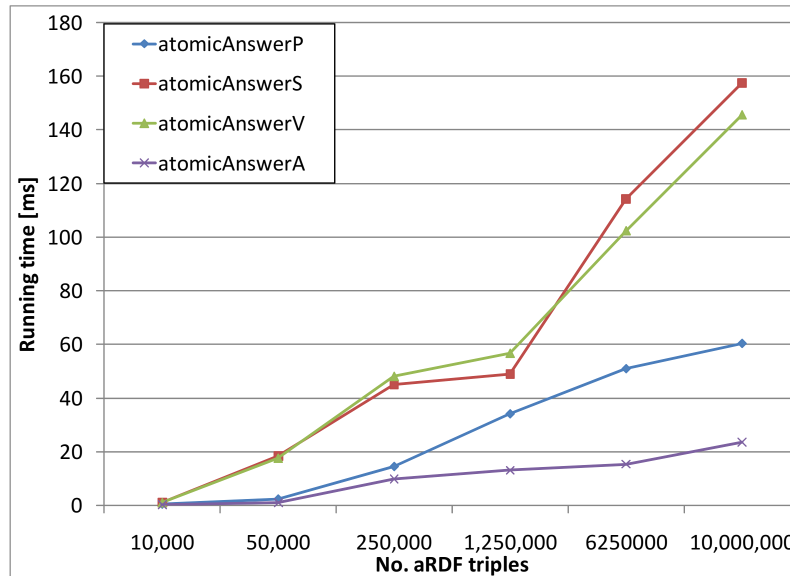
Table 4.1: Summary of consistency checking and atomic query algorithms

Dataset	Synthetic [ms]						ChefMoz [ms]	GovTrack [ms]
No. aRDF triples x 1,000	*10*	*50*	*250*	*1,250*	*6,250*	*10,000*	*549*	*12,340*
Consistency checking	2.1	23.5	103.4	346.1	1047.1	1438.1	187.1	1754.1
Naive atomic query	1.2	27.6	487.2	976.1	2781.3	4231.5	1076.1	4891.3
atomicAnswerP	0.5	2.4	14.5	34.1	50.9	60.3	19.9	68.1
atomicAnswerS	1.1	18.5	45.1	49.1	114.3	157.6	47.5	190.8
atomicAnswerV	1	17.6	48.1	56.7	102.4	145.6	51	176.1
atomicAnswerA	0.2	1	9.8	13.1	15.4	23.6	10.4	34.1

As a first step, we measured the running time of the consistency checking, naive atomic answer and the four atomic answer algorithms on all three datasets.



(a) Running time of *consistencyCheck*, *naiveSimpleAnswer*, *answerA*, *answerP*, *answerV* and *answerS*



(b) Running time of *answerA*, *answerP*, *answerV* and *answerS*

Figure 4.12: Consistency checking and atomic query answers

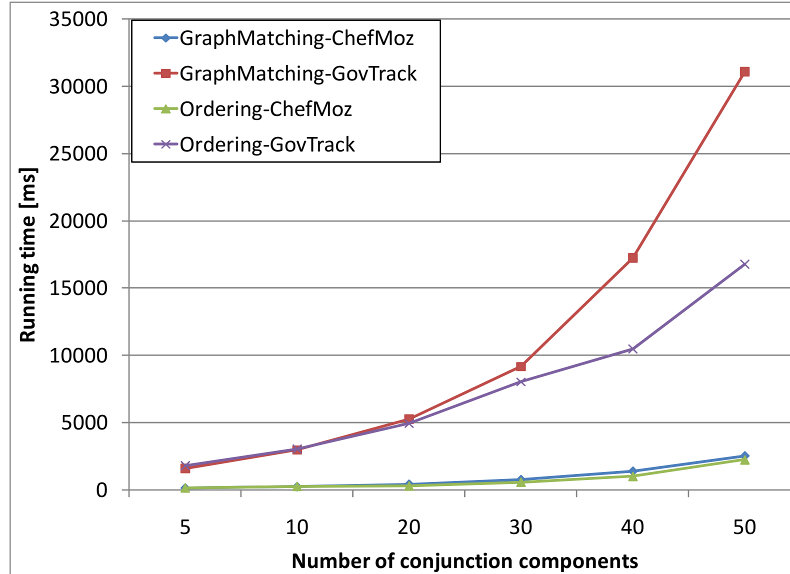
For the query answer algorithms, the results are averaged over 50 separate queries over each dataset. A summary of the results is shown in Table 4.1 and a graphical depiction of the variation of the running time with the size of the synthetic dataset

is shown in Figure 4.12(a) and (b) (in quasi-logscale). We point out that the most time consuming operation is by far the fixpoint computation, as seen from the data for the naive query answer algorithm. Given the logscale plot, the data suggests a nearly double exponential rise for the naive query algorithms. We can see that the consistency checking algorithm is much faster than the naive atomic query algorithm, primarily due to the fact that it avoids the fixpoint computation. Finally, from the atomic query answer algorithms, *atomicAnswerS* and *atomicAnswerV* take the longest. The motivation follows from the formal description of the algorithms: *atomicAnswerS* and *atomicAnswerV* search for $p_q - paths$ originating at a known vertex r or ending in a known vertex v. On the other hand, for *atomicAnswerP* and *atomicAnswerA*, both r and v are known, which narrows the search space considerably.

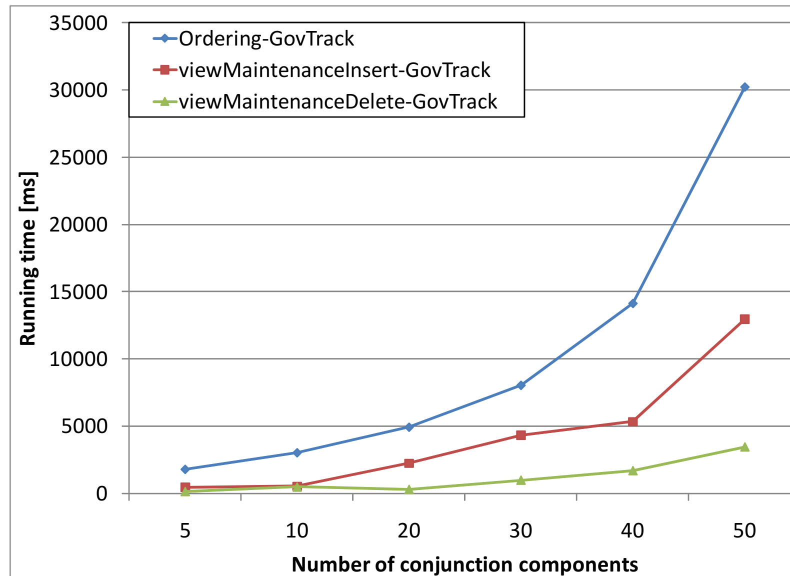
Table 4.2: Summary of conjunctive answer algorithms

Dataset	**Query size**	**Synthetic [ms]**						**ChefMoz [ms]**	**GovTrack [ms]**
No. aRDF triples x 1000		*10*	*50*	*250*	*1,250*	*6,250*	*10,000*	*549*	*12,340*
Graph Matching	**5**	5	21	88	288	892	1146	124	1567
	10	8	38	150	543	1684	1955	220	2945
	20	14	71	274	1006	3121	3540	403	5254
	30	25	130	491	1877	5435	6289	745	9155
	40	47	231	846	3525	9316	10715	1366	17221
	50	87	438	1601	6213	16936	19535	2501	31100
Ordering	**5**	5	25	105	313	896	1341	134	1771
	10	10	41	176	544	1941	1962	255	3006
	20	10	54	267	913	2746	2788	283	4905
	30	24	113	335	1644	4132	5988	567	8017
	40	41	158	783	2458	8374	7023	989	10456
	50	84	389	1290	5925	13596	19130	2227	16758

In the next step, we analyzed the two conjunctive query algorithms, with *naiveConjunctAnswer* as a baseline. We varied the size of the conjunctive queries



(a) Running time of *conjunctAnswer_GraphMatching* and *conjunctAnswer_Ordering* on the GovTrack and ChefMoz datasets



(b) Comparison between *conjunctAnswer_Ordering*, *viewMaintenaceInsert* and *viewMaintenanceDelete*

Figure 4.13: Conjunctive queries and view maintenance

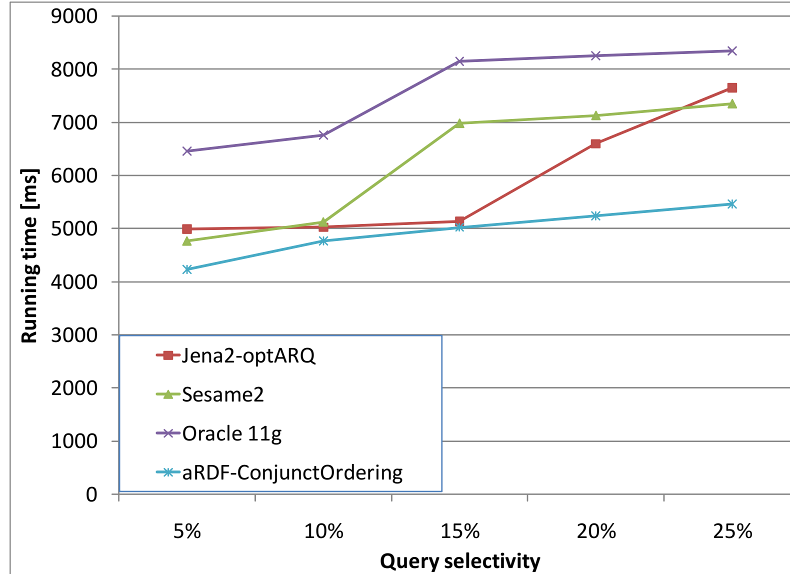
from 5 to 50 component queries, while maintaining the number of variables in the graph patterns at 25%. This was done to ensure that the selectivity of the graph patterns remains stable (approximately 7%, with a standard deviation of .12%).

The experimental results are summarized in Table 4.2. The *naiveConjunctAnswer* (not shown in the table) ran out of memory at 1,250K triples for queries with 5,10 and 20 components, at 250K for queries with 30 and 40 components and at 50K triples for queries with 50 components. The running time for *naiveConjunctAnswer* algorithm was overwhelmingly larger than the other two algorithms (for instance, taking 789ms at 250K triples with 5 components compared to 88 and 105 ms for the graph matching and ordering algorithms respectively). In Figure 4.13(a) we can also see that the ordering algorithm does slightly better than the graph matching variant in terms of running time. We also notice that both algorithms have an average-case complexity much lower than the worst case complexity (which was factorial in the size of the data for the graph matching variant).

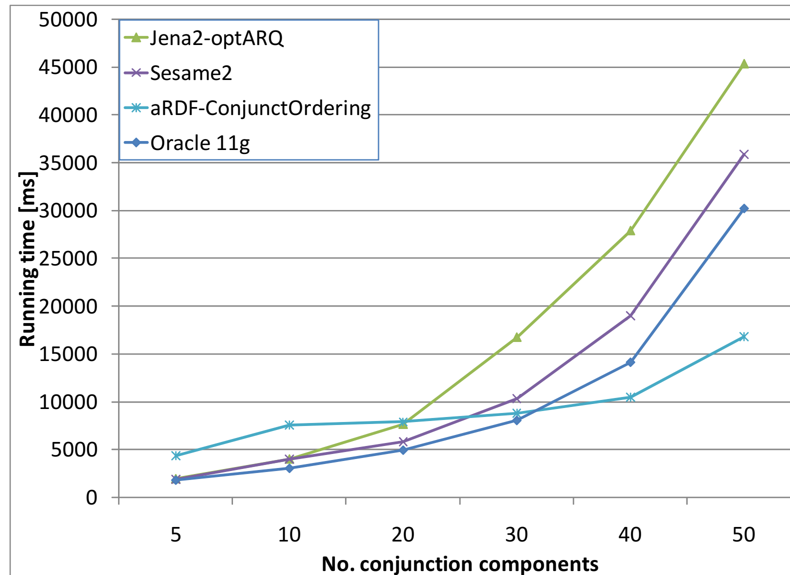
Table 4.3: Summary of view maintenance algorithms

Dataset	**Query size**	**Synthetic [ms]**						**ChefMoz [ms]**	**GovTrack [ms]**
No aRDF triples x 1000		*9*	*45*	*225*	*1,125*	*5,625*	*9,000*	*495*	*111,060*
View maintenance Insert	**5**	1	4	43	114	287	453	55	427
	10	2	13	44	246	771	808	47	509
	20	3	33	107	260	1280	1382	67	2225
	30	12	39	108	795	1000	1226	347	4310
	40	7	92	254	1123	3974	3422	557	5326
	50	35	204	546	1897	5681	2959	576	12950
View maintenance delete	**5**	0	1	11	25	78	94	7	105
	10	1	3	39	36	101	110	19	447
	20	1	11	17	69	239	275	21	271
	30	2	9	32	250	305	619	67	960
	40	3	25	197	438	759	2647	88	1670
	50	9	26	45	693	2145	2134	218	3440

Next, we studied the performance of the view maintenance algorithms. We considered the same set of conjunctive queries as in the previous step and selected uniformly at random 10% of the triples in each set to insert and respectively delete.



(a) Comparison with Jena2, Sesame2 and Oracle on varied selectivities



(b) Comparison with Jena2, Sesame2 and Oracle on varied query sizes

Figure 4.14: Comparison between aRDF and competing systems

We measured the running time of the view maintenance algorithms and give a summary of the results in Table 4.3. Note that the size of dataset is the size before any insert operation and respectively after all the deletion operations. The cells in the table represent the average running time of the corresponding view

maintenance algorithm after each insertion or deletion. The data in Figure 4.13 clearly indicates that maintaining a conjunctive query can be done much faster that re-running the entire query. We also noticed that consistently, view maintenance when deleting a triple is much faster than view maintenance when inserting the same triple. This trend is also clear in Figure 4.13(b) for the average insertion and deletion maintenance times. The fact that deletion maintenance is much faster than insertion maintenance is explained by the fact that the branching factor in $viewMaintenanceDelete$ is much lower than that of $viewMaintenanceInsert$.

Finally, we evaluated the performance of aRDF by comparing it to that of Jena2, Sesame2 and Oracle 11g on the GovTrack dataset. Since the three systems do not support transitivity for user-defined properties, we considered all properties in the dataset to be non-transitive. In the case of Jena2, we used the optimized query planner optARQ recently proposed by Stocker et al [50]. From the set of GovTrack frequent queries, we selected at random an evaluation set comprised of 20 queries for selectivity factors of 5, 10, 15, 20 and 25% [3]. The results are shown in Figure 4.14(a). We observed that the aRDF-Ordering algorithm outperformed all three systems (an average 10% improvement over Jena2-optARQ, the next best system) and its performance scales better for higher selectivity queries than the other systems. We selected a second evaluation set, also at random comprised of 20 queries each for conjunctions of size 5 through 50 in increments of 5. The results shown in Figure 4.14(b) also confirm that aRDF-Ordering outperforms the other

[3]Since these were existing queries, the selectivity cannot be always pinpointed to a multiple of 5%. Each query was assigned to the closest multiple of 5% from its actual selectivity

systems both in terms of query time and in terms of scalability.

4.4 Summary

In this chapter we described the query framework for Annotated RDF. We defined algorithms for simple and SPARQL-like conjunctive queries, as well as view maintenance techniques for aRDF databases. All the algorithms presented were proved to be correct and analyzed from the point of view of worst-case data complexity. To our knowledge, the only other algorithm for RDF annotated with time intervals was given by Gutierrez et al. [22], without empirical evaluation. We have performed a thorough series of experiments on two real-world and one synthetic dataset, as well as comparisons with leading RDF storage systems. Our results show that: (i) aRDF query processing is much faster than answering queries over reified RDF; (ii) decomposing conjunctive queries and joining the results yields significant performance improvements over graph matching algorithms and (iii) recomputing query results incrementally is always more efficient for aRDF than reprocessing the query.

Chapter 5

Indexing RDF

One of RDF's major strengths over the relational data model is the expressiveness of its query language. A RDF query expressed in SPARQL is essentially a labeled graph with zero or more variables labeling either the vertices and/or the edges. The problem is to find "matches" for this pattern in the RDF graph. Past work on RDF indexing [54] does not provide any index specialized to handle such queries. In this chapter, we describe our approach to indexing RDF and aRDF data and provide a thorough experimental evaluation showing our index structure scales very well for massive real-world and synthetic datasets. In the first part of the chapter, we propose a Graph-based RDF INdex (**GRIN** for short) [57] that improve the response time to SPARQL queries and can be used in conjunction with a variety of backend systems. In the second part, we extend **GRIN** to aRDF and show experimentally that using both the index and the aRDF query processing algorithms improves query processing time for a billion triple dataset by up to 24% compared to the best existing systems. Specifically, our contributions are the following.

1. The key problem in processing RDF queries is that we do not have any index supporting subgraph matching. Such an index structure should preserve the proximity of vertices in the RDF graph. We do this by identifying certain *center* vertices in a graph (think of these as vertices that occupy strategic

positions in the graph) and by proposing a notion of *radius* from those center vertices. All vertices within the stated radius of a center vertex are associated with that center.

2. We then define **GRIN** – an efficient tree data structure to store the regions defined by these center vertices (together with their associated radii).

3. Subsequently, we develop algorithms to answer graphical queries efficiently by using the **GRIN** data structure. The algorithms are proved correct, and their worst case computational complexity is stated.

4. We identify and theoretically analyze the **GRIN** characteristics that have impact on query performance and propose a framework for optimizing **GRIN** index structures.

5. We then show how **GRIN** can be extended to handle aRDF triples and how aRDF query algorithms can be used in conjunction with the index structure.

6. Finally, we conduct a detailed series of experiments on the GovTrack (26 million triples) and a set of synthetic datasets generated with the Lehigh University Benchmark (LUBM) up to a billion triples. We compare against Jena2, Sesame2, RDFBroker, Oracle 11g and the column store LucidDB. We measure the effectiveness of **GRIN** along three dimensions: (i) how large is the index – we show that **GRIN** is the smallest compared to the other systems; (ii) how long does it take to answer queries – our results indicate using **GRIN** over a relational database store is already faster than the other systems, especially

so when combined with the aRDF query algorithms; (iii) how long does it take to build the index – the time taken by **GRIN** is comparable to the best competing systems.

5.1 SPARQL graph patterns

Conjunctive queries are formulated in SPARQL through graph patterns. Since we do not discuss rarely used SPARQL constructs such as UNION or OPTIONAL, we will refer to SPARQL graph patterns as *RDF queries*. In this section, we introduce a formal syntax for an RDF query and provide its semantics by defining what constitutes a correct answer. We employ the same notation as before, $\mathcal{R}$ for the set of resources and blank nodes, $\mathcal{P}$ for the set of properties and $\mathcal{L}$ for the set of literals. For most of this chapter, we will assume that all inferences on transitive properties, *rdfs:subClassOf* and *rdfs:subPropertyOf* have already been computed and the corresponding triples added to the database. In Section 5.6.2 we will show how **GRIN** can process queries without precomputing transitivity inferences. Furthermore, we will assume the RDF graph O is connected; if that is not the case, we can build separate **GRIN** index structures for every connected component.

Definition 5.1 (RDF query)**.** *A* RDF query *is a 4-tuple* (N, E, V, λ_n) *where:*

- N *is a set of vertices.*
- V *is a set of variables.*
- $E \subseteq N \times N \times (V \cup \mathcal{P})$ *is a set of edges.*
- $\lambda_n : N \to \mathcal{R} \cup \mathcal{L} \cup V$ *is a vertex labeling function.*

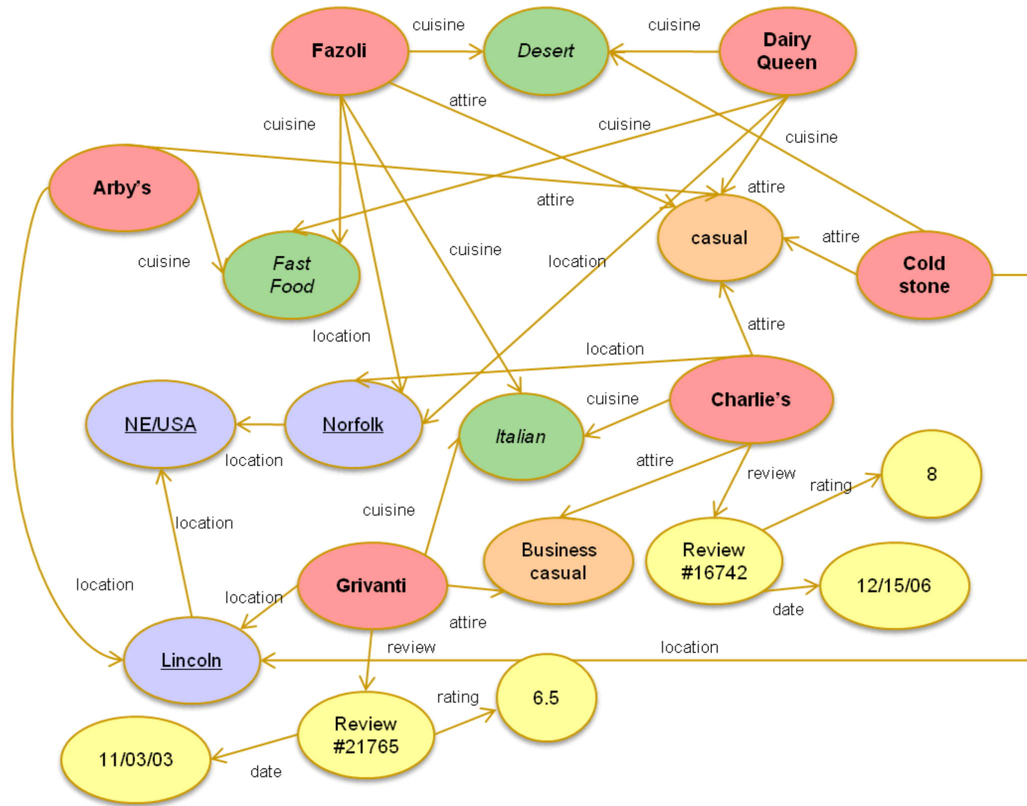


Figure 5.1: A RDF graph from the ChefMoz dataset

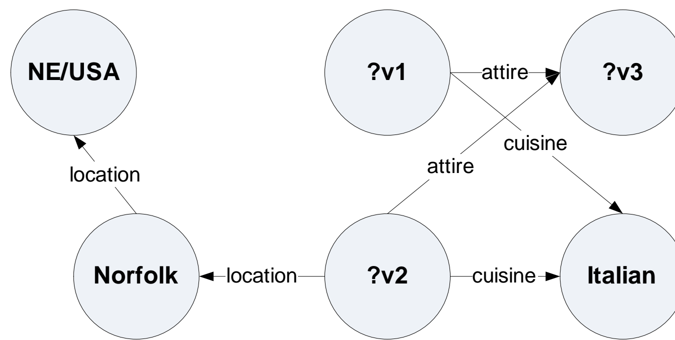


Figure 5.2: RDF query example

We refer to each edge in the query graph pattern as a query atom.

Example 5.2. *Figure 5.1 contains an example graphical depiction of an RDF graph extracted from the ChefMoz dataset* [1]*. The RDF data contains six restaurants (**bold vertices**) in two locations in NE, USA (underlined vertices) for three different cui-*

[1]Some URIs were shortened for readability.

*sine styles (italicized vertices); in addition, the data contains the type of attire required, as well as two restaurant reviews. We will use this RDF graph as a running example for presenting **GRIN**.*

The query graph in Figure 5.2 informally says: find restaurants $?v1$, $?v2$ *with the same attire* $?v3$*, such that both restaurants serve Italian food and* $?v2$ *is in Norfolk, which is located in NE, USA The query can be expressed in SPARQL as:*

$$\begin{aligned} SELECT \quad & ?v1\ ?v2\ ?v3 \\ WHERE \quad & \{\{(?v1\ attire\ ?v3)\ .\ (?v1\ cuisine\ Italian)\} \\ & \{(?v2\ attire\ ?v3)\ .\ (?v2\ cuisine\ Italian)\ . \\ & (?v2\ location\ Norfolk)\} \\ & \{(Norfolk\ locatedIn\ NE/USA)\}\} \end{aligned}$$

To answer an RDF query over a database O, we are looking for all possible substitutions for the query variables in V such as the query graph after the proper substitutions is entailed by O. As in the case of aRDF, an RDF graph O' is entailed by O if and only if any satisfying interpretation of O is a satisfying interpretation to O'. In addition to the entailment conditions in RDF, aRDF semantics also requires that paths on transitive properties and the corresponding annotations be taken into account when deciding if a database entails a given triple (Definition 3.6).

Definition 5.3 (RDF query answer)**.** *The answer to an RDF query* $q = (N, E, V, \lambda_n)$ *w.r.t. a database* O*, denoted* $Ans_q(O)$*, is a set of variable substitutions* $\{\theta_1, \ldots, \theta_k\}$*, with* $\theta_i : V \rightarrow \mathcal{R} \cup \mathcal{L}$ *such that the following conditions hold:*

1. *(Soundness). For all* $i \in [1, k]$ *and for all query atoms* $q_j \in q$, $O \models q_j\theta_i$*, where* $q_j\theta_i$ *denotes the application of the substitution* θ_i *to query atom* q_j*.*

2. *(Completeness). For all substitutions θ such that $O \models q_j\theta$ for all query atoms q_j, there is a substitution $\theta_j \in Ans_q(O)$ that is more general than θ.*[2]

Note that the query operations we specified are akin to relational selection. We have not defined anything that is equivalent to projection over RDF databases (i.e., we do not select a subset of variables we are interested in). Experimentally, we have determined that unlike the relational case, projection does not help much with the query running time (which is dominated by searching for subgraphs matching the query). Projection can be therefore applied after finding $Ans_q(O)$ in linear time in the size of the answer.

Example 5.4. *Consider the query in Example 5.2 w.r.t. the RDF graph in Figure 5.1. The possible substitutions are: ($?v1 \leftarrow$ Grivanti , $?v2 \leftarrow$ Charlie's, $?v3 \leftarrow$ businessCasual) and ($?v1 \leftarrow$ Fazoli, $?v2 \leftarrow$ Charlie's, $?v3 \leftarrow$ casual).*

A naive algorithm for answering the RDF query q on a database O is:

1. For each query atom $q_j \in q$, compute the set Θ_j of substitutions where O entails $q_j\Theta_j$.
2. Consider all possible elements of $\Theta_1 \times \cdots \times \Theta_n$ and select those elements $(\theta_1, \ldots, \theta_n)$ for which **all** substitutions θ_i with $i \in [1, n]$ are compatible. Two substitutions are compatible if and only if they do not assign different constant (resource or literal) values to the same variable.

[2]In the case of RDF, a substitution that maps a variable to a blank node is more general than a substitution that maps the same variable to a resource or literal. The "generality" of a substitution was first defined in [41].

The clear disadvantage of this algorithm is that it has to compute a Cartesian product (essentially a join of n relations), which is prohibitively expensive for complex queries. In fact, we show experimentally in Section 5.7 that some of the current RDF database systems do not scale well to large or high selectivity queries.

Instead, let us look at Example 5.4 again. The entire ChefMoz dataset this example is extracted from contains over 800,000 triples, and yet the answer to our query can be found in a very small portion of the entire database. Therefore, a better strategy is to (i) identify the smallest portion of the database that is guaranteed to contain the answer and (ii) perform subgraph matching on that portion. To accomplish this, we define the **GRIN** index structure for RDF.

5.2 The GRIN index

GRIN is based on the intuition that vertices that are "close" together in the RDF graph are more likely to appear together in the answer to an arbitrary query, and therefore should be stored on the same page (in the same index node). Moreover, we want a way of quickly determining whether a given neighborhood contains the answer to the query q. To accomplish both goals, let us assume $d : (\mathcal{R} \cup \mathcal{L}) \times (\mathcal{R} \cup \mathcal{L}) \rightarrow I\!N$ is a metric[3] defined on the set of resources and literals in the RDF graph. Since d is a metric, it has the triangle property, i.e., $d(x, y) \leq d(x, z) + d(y, z)$. There are many such metrics, the typical examples being the length of the shortest or longest cycle-free path between two resources or literals in the RDF graph. For

[3]A metric over a space X is a function $d : X \times X \rightarrow \Re$ that is non-negative ($d(x) \geq 0$ for all $x \in X$), has the identity of indiscernibles property ($d(x) = d(y)$ if and only if $x = y$), is symmetric ($d(x, y) = d(y, x)$) and has the triangle property ($d(x, z) \leq d(x, y) + d(y, z)$).

simplicity, we will assume for the rest of the chapter that d is the minimum path length between two resources. For instance, the minimum distance between *Fazoli* and *NE/USA* is 2 in Figure 5.1.

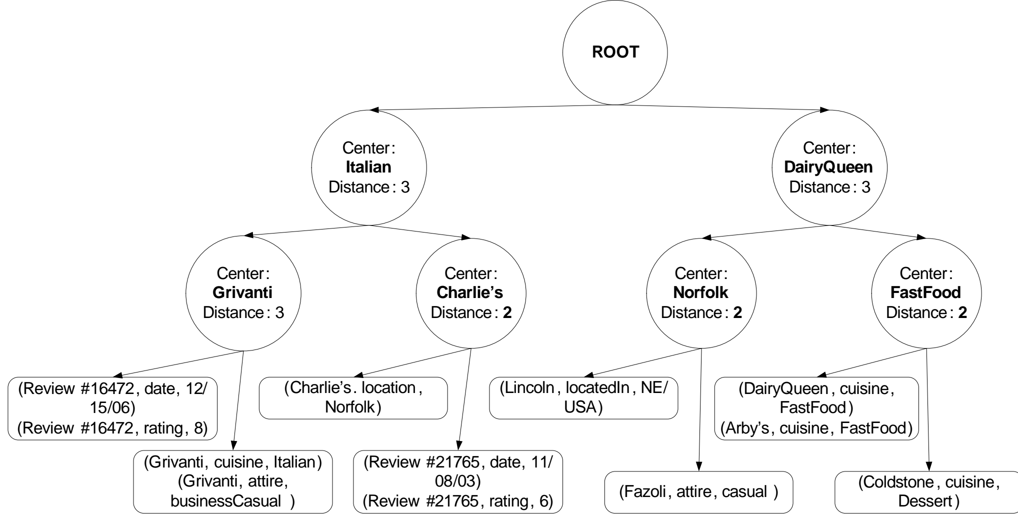


Figure 5.3: GRIN index example

Definition 5.5 (GRIN index)**.** *A* GRIN index *is a* balanced *tree such that:*

- *Each leaf node ℓ contains a set $N_\ell \subseteq \mathcal{R}$ of vertices s.t. for all leaf nodes $\ell' \neq \ell$, $N_\ell \cap N_{\ell'} = \emptyset$, and $\cup_{\ell \in L} N_\ell = \mathcal{R}$;*
- *Each non-leaf node t is represented as a pair (c, r), with $c \in \mathcal{R}$ and $r \in \mathbb{N}$. Intuitively, this is a very succinct representation of the set of resources in the graph at distance at most r of the resource c according to the metric d. We say each non-leaf node t contains the set of resources $N_t = \{c' \in \mathcal{R} | d(c, c') \leq r\}$.*
- *For any nodes x, y in the tree such that x is a parent of y, $N_x \supseteq N_y$.*
- *Any non-leaf node t except for the root has at most M and at least $\frac{M}{2}$ children, where M is a constant fixed apriori.*

The set of leaf nodes in the **GRIN** tree form a partition of the set of triples in the RDF graph O. Interior nodes are constructed by finding a "center" triple, denoted c, and a radius value, denoted r. An interior node in the binary tree *implicitly* represents the set of all vertices in the RDF graph that are within r units of distance (i.e., less than or equal to r links) from the center. The condition requiring that each non-leaf node has between m and M children is typical of index tree structures such as B-trees [5] or R-trees [23]. The condition requires that a node is always at least "half full" (has at least $\frac{M}{2}$ children). It also implies that two half-full nodes can be joined to make a legal node and a full node can be split into two half-full nodes (potentially the split requires one child be pushed up to the parent). The limitations imposed on inner nodes allow resources to be inserted or deleted from the tree in linear time in the height of the tree in most cases.

Example 5.6. *Figure 5.3 shows an example GRIN index structure for the RDF graph in Figure 5.1, for $m = M = 2$. Note that the leaf nodes consist of clusters of resources – for this example, there is more than one possible GRIN structure, since membership of a resource to a cluster or another is often tied. The intermediate node* (Grivanti, 3) *signifies the set of resources in the graph with a minimum path less than or equal to three from the vertex* Grivanti.

Assuming resources are stored as URIs[4], storing a circle requires approximately the same amount of space as storing a resource (in addition to its center, a **GRIN** circle also has to store a radius and children addresses). Let Max be the maximum number of RDF graph vertices that can be stored on a single page. Note that Max

[4]Uniform Resource Identifiers are the usual way to uniquely identify resources in RDF.

can be easily computed since we know the page size and can compute the maximum amount of space required by a resource. We found empirically that, reserving space for radii and children pointers, we can store approximately $\frac{3}{4}Max$ circles per page. We can therefore choose $M = \lfloor \frac{3}{4}Max \rfloor$ to ensure that all the children of a given circle are stored on the same page, therefore minimizing the number of page faults.

Next, we will determine the number of leaf nodes of the tree in the following way. Let us denote the number of leaf nodes by C. Since we would like to build a balanced tree, C must be a power of M. We know that $\frac{|\mathcal{R}|}{C} \leq \frac{3}{4}Max$, hence $C \geq \frac{|\mathcal{R}|}{Max}$. The smallest power of M for which this inequality holds is $C = M^{\lceil log_M \frac{\mathcal{R}}{Max} \rceil}$.

We use $d_c : 2^{(\mathcal{R} \cup \mathcal{L})} \times 2^{\mathcal{R} \cup \mathcal{L}} \rightarrow I\!N$ to denote an arbitrary, but fixed, inter-cluster distance function based on the metric d. d_c takes two sets of resources and returns a numeric value. Three well-known inter-cluster distances are often used in clustering algorithms: (i) **Single link** defines $d_c(S, S') = \min_{x \in S, y \in S'}(d(x, y))$; (ii) **Complete link** defines $d_c(S, S') = \max_{x \in S, y \in S'}(d(x, y))$ and (iii) **Average link** defines $d_c(S, S') = \frac{\sum_{x \in S, y \in S'} (d(x,y))}{|S| \times |S'|}$. Experimentally we have found that average link provides the best query processing time.

The algorithm that builds the **GRIN** index is shown in Figure 5.4. The algorithm builds the index structure bottom up. Initially, we cluster the vertices in the graph into C disjoint sets using a modification of the *partitioning around medoids* (PAM) clustering algorithm [31] (line 1). PAM starts by choosing C random vertices from the graph as cluster centroids. It then assigns all vertices in the graph to one cluster, based on their distances to the chosen cluster centroids. After C clusters

have been formed, the centroids are re-computed and the process is repeated until an equilibrium is reached. We modified the original algorithm to ensure no cluster contains more than M vertices. Ties (cases when a vertex could be assigned to more than one cluster) are broken uniformly at random.

For each intermediate level in the tree, *GRINBuild* chooses a random node u from the available nodes and computes the "closest" $k = \frac{M}{2} - 1$ nodes to u in terms of the inter-cluster distance d_c (lines 7–8). Let these nodes be $v_1, \ldots, v_k$. u and $v_1, \ldots, v_k$ are then assigned a new parent node (c, r). The values of the center c and radius r are computed based on the set of vertices $\bigcup_{i \in [1,k]} N_{v_i} \cup N_u$ (lines 9–10). The process is repeated until we build the root of the tree, which corresponds to a set of resources encompassing the entire graph (loop condition on line 3).

We point out that **GRIN** does not commit to a particular representation of the data. In fact, in Section 5.7 we will show that GRIN works well over PostgresSQL, Jena2 and LucidDB. Also, the index structure can be constructed to contain the actual resources in the database, or alternatively let the resources be represented in a relational database and store pointers to the actual data. We have implemented both approaches, but chose the latter variant to have a fair comparison to database-backed RDF systems.

Complexity of building the GRIN Index. The set of vertices represented by a GRIN node is at most $|\mathcal{R}|$. For a level of the GRIN tree containing k nodes, the most time-consuming operation is the computation of inter-cluster distance, which can be done in parallel for the entire level in time $\mathcal{O}(|\mathcal{R}|^2 \cdot k^2)$. The number of leaf nodes C is $\mathcal{O}(|\mathcal{R}|)$ and the height of the tree is $\mathcal{O}(\log_M(|\mathcal{R}|))$. This leads to a worst

Algorithm GRINBuild$(C,\ Max,\ O)$

Input: C is the number of leaf nodes, Max is the maximum number of vertices on a page, O is the RDF database.

Output: The GRIN index structure G.

1: $L_0 \leftarrow PAM(O, C, Max)$
2: Create leaf nodes in G from clusters in L_0
3: **for** $i \in [0, \lceil \log_M C - 1 \rceil]$ **do**
4: $\quad F \leftarrow L_i$
5: $\quad L_{i+1} \leftarrow \emptyset$
6: $\quad$ **while** $F \neq \emptyset$ **do**
7: $\quad\quad$ Pick a random node $u \in F$
8: $\quad\quad$ Find $v_1, \ldots, v_k \in F$ with $k = \frac{M}{2} - 1$ that minimize $d_c(N_u, N_{v_i})$
9: $\quad\quad$ Compute centroid c and radius r for $\bigcup_{i \in [1,k]} N_{v_i} \cup N_u$
10: $\quad\quad$ Create node $p = (c, r)$ in G as a parent of u and $v_1, \ldots, v_k$
11: $\quad\quad L_{i+1} \leftarrow L_{i+1} \cup \{p\}$
12: $\quad\quad F \leftarrow F - \{u, v_1, \ldots, v_k\}$
13: $\quad$ **end while**
14: $\quad$ Add level L_{i+1} to G
15: **end for**
16: **return** G

Figure 5.4: An algorithm to build the GRIN index

case complexity for building the index of $\mathcal{O}(|\mathcal{R}|^4 \log_M(|\mathcal{R}|))$. However, we will show experimentally that building the index is generally much faster than the worst case.

5.3 Answering queries with GRIN

In this section, we show how to evaluate an RDF query $q = (N, V, E, \lambda_n)$ against the **GRIN** structure. We start by showing how to derive a set of inequality constraints $cons(q)$ from the query. The constraints will be evaluated against the nodes of the **GRIN** index to identify the smallest subgraph that contains answers to q. We derive $cons(q)$ in the following way. Let G_q be the graph corresponding to query q. For any path (not necessarily a p-path) of length l in G_q between a resource c (constant) and a variable v, we add the constraint $d(c, v) \leq l$ to $cons(q)$.

The same rule applies for paths from v to c.

Example 5.7. *Consider the example query in Figure 5.2. The query leads to the following set of constraints:* $d(?v1,\ NE/USA) \leq 4$, $d(?v2,\ NE/USA) \leq 2$, $d(?v2,\ Norfolk) \leq 1)$, $d(?v1,\ Norfolk) \leq 3$, $d(?v1,\ Italian) \leq 1$, $d(?v2,\ Italian) \leq 1$, $d(?v3,\ NE/USA) \leq 3$, $d(?v3,\ Norfolk) \leq 2$, $d(?v3,\ Italian) \leq 2$.

Algorithm GRINAnswer$(O,\ G,\ q,\ n_I)$

Input: RDF database O, GRIN index G and query $q = (N, V, E, \lambda_n)$, GRIN node n_I. *subgraphMatch* is a subgraph matching method that finds an isomorphism between the query graph q and a graph H and returns a set of substitutions Θ for the variables in q.

Output: A set of answers Θ.

1: $\Theta \leftarrow 0$
2: **if** n_I is a leaf node **then**
3: $H \leftarrow$ the subgraph of O containing the resources in N_{n_I}
4: **return** *subgraphMatch*(q,H)
5: **else if** n_i is not rejected by checking rules (R1), (R2) against $cons(q)$ **then**
6: $\Theta \leftarrow \bigcup_{m \text{ child of } n_I} GRINAnswer(O, G, q, m)$
7: **if** $\Theta = \emptyset$ **then**
8: $H \leftarrow$ the subgraph of O containing the resources in N_{n_I}
9: **return** *subgraphMatch*(q,H)
10: **else**
11: **return** Θ
12: **end if**
13: **else**
14: **return** $\emptyset$
15: **end if**

Figure 5.5: An algorithm to answer queries over the GRIN index

We use the constraints generated from the query to identify nodes in the **GRIN** structure that may contain answers to the query. On any **GRIN** node, we have the option of *accepting* the node (which means it may contain answers to the query) or *rejecting* the node (which means it is guaranteed not to contain answers to the query). Consider a **GRIN** node corresponding to the circle (c, r). We will

define two rules to decide whether (c, r) should be rejected.

The first rule is straightforward: *for any constant (resource) x in q, **reject** (c, r) if $d(c, x) > z$* (R1). Intuitively, we are rejecting the circle represented by the **GRIN** node if any constant factors in the query are outside it.

Let's consider the case of a constraint $d(x, v) \leq l$ involving variable v and resource x. Since d is a metric, $d(c, v) \leq d(c, x) + d(x, v) \leq d(c, x) + l$. Note that $d(c, x)$ is a constant. If $d(c, x) + l \leq z$, we are sure that v is inside the circle (c, r). If this case, we say (c, r) *definitely satisfies* v. Also from the fact that d is a metric we can write $d(c, x) - l \leq d(c, x) - d(x, v) \leq d(c, v)$. If $z \leq d(r, c) - l$ then we are sure v is outside the circle (c, r). In this case, we say (c, r) *does not satisfy* v. If any variable is not satisfied, then we cannot find an answer to the query within (c, r).

We may also have situations in which neither of the two cases hold – we do not know for certain whether v is inside or outside the circle solely on the constraints derived from the query. We will take a conservative approach and only look for solutions in nodes that *definitely satisfy* **all** variables. This has the potential downside of stopping at circles that are larger than necessary, but we have found experimentally that this policy is very effective. The second rule is: *if there exists $v \in V$ such that (c, r) does not **definitely satisfy** v, then **reject** (c, r)* (R2). Intuitively, we want to find the smallest sets of vertices that definitely satisfy all variables – or the equivalent, nodes lowest in the **GRIN**tree that satisfy all variables. Note that only one constraint per variable needs to be satisfied in order for the variable to be satisfied.

Example 5.8. *Consider the node* (*Grivanti,* 2) *in the index in Figure 5.3 and the constraint that says* $d(?v2, \textit{Norfolk}) \leq 1$*. This constraint is trivially* not satisfied, *since Norfolk is not in the circle specified by the node. However, the variable* $?v1$ *is satisfied from* $d(\textit{Grivanti},?v1) \leq d(\textit{Grivanti, Italian}) + d(?v1, \textit{Italian}) \leq 2$,

Figure 5.5 contains the query evaluation algorithm. Given a query q and a node n_I of a GRIN index G, *GRINAnswer* evaluates q over the subtree rooted in n_I. Answering a query over the database O is equivalent to calling *GRINAnswer(q,root(G))*.

The *GRINAnswer* algorithm locates the smallest index node that is guaranteed to contain the answers to the query and calls *subgraphMatch* to answer the query over the resources in the index node. *subgraphMatch* can be any method of answering conjunctive queries over RDF graphs. We experimented with two such methods: the subgraph matching algorithm by Cordella et al. [9] and the Jena2 ARQ package. Experimentally, we found that the new Jena2 ARQ algorithms were more efficient in practice.

If *GRINAnswer* is invoked for a leaf node, it will simply match the query graph q with the subgraph of O contained in n_I (lines 2–4). Otherwise, if n_I is a potential candidate (line 5 checks (R1), (R2)), we will attempt a recursive call on children of n_I (line 6). Given that we stop only when all variables are *definitely satisfied*, we are guaranteed one of two outcomes.

(i) One of the recursive calls will return a non-empty answer. This implies that there exists a descendant (c, r) of n_I that passes both rules (R1) and (R2). In

turn, this implies that all answers to the query are guaranteed to be inside the (c, r). All we need to do is return the answer found by subgraph matching while analyzing (c, r) (line 11).

(ii) We have not found an answer for any descendant, in which case we will attempt to run the subgraph matching on n_I itself and return the results (line 8–9).

Cordella et al. [9] show the memory complexity of the subgraph matching algorithm to be $\Theta(N)$ (with a small constant factor), where N is the total number of vertices in the graphs to be matched, whereas time complexity ranges from $\mathcal{O}(N)$ in the best case to $\mathcal{O}(N!)$ in the worst case. The ARQ algorithm has the same worst case complexity. The *GRINAnswer* algorithm therefore has a worst-time complexity of $\mathcal{O}(|\mathcal{R}|!)$. However, we have discovered in practice that *GRINAnswer* is able to identify very small circles on which to match very efficiently. This makes the value of N very small compared to $|\mathcal{R}|$ in practice. Our experimental results show that *GRINAnswer* is significantly faster than Jena2, Sesame2, Oracle 11g and LucidDB.

Example 5.9. *To better understand how query evaluation works, consider the query in Figure 5.2 without the node NE/USA. We start off at the root of the tree. We recursively call the evaluation for the nodes (Italian, 3) and (DairyQueen, 3). (DairyQueen, 3) can be quickly eliminated since we cannot determine whether* $?v1$ *and* $?v2$ *are definitely satisfied; this follows from* $d(\textit{DairyQueen, Italian}) = 3$, $d(?v1, \textit{Italian}) \leq 1$ *and the fact that the radius of the current circle is 3. On the other hand, (Italian, 3) definitely satisfies all variables, but its children do not. This means (Italian, 3) is the* ***GRIN*** *node the algorithm is looking for, namely the node*

that definitely satisfies *all variables and is at the lowest possible level in the tree. When we match the query graph against the circle (Italian, 3), we obtain the two substitutions in Example 5.4, namely* $(?v1, ?v2, ?v3)$ *= (Grivanti, Charlie's, businessCasual) and* $(?v1, ?v2, ?v3)$ *= (Fazoli, Charlie's, casual).*

5.4 GRIN optimization

In this section we analyze two structural features of the **GRIN** index that impact query performance. We define the notions of *coverage* and *overlap* and provide a theoretical analysis of the complexity of optimizing these measures. We then define a new heuristic index construction algorithm that constructs a near-optimal index tree. Throughout the section, we will refer often to the *level* of a GRIN index tree. By convention, the leaf nodes of the index are on level 0, whereas the root is on level h, where h is the height of the index.

5.4.1 Coverage and overlap

Definition 5.10 (Coverage and overlap)**.** *Let G be a* ***GRIN*** *index tree and let $0 < l < h$ be an arbitrary but fixed level in G. Let $C_1 = (c_1, r_1), \ldots C_m = (c_m, r_m)$ be the circle nodes on level l.*

- *The set of overlapped nodes for level l is defined as $L = \{r \in \mathcal{R} | \exists C_i, C_j, C_i \neq C_j \text{ s.t. } d(r, c_i) \leq r_i \text{ and } d(r, c_j) \leq r_j\}$. Intuitively, L is the set of nodes that are implicitly part of more than one circle on level l.*
- *The set of uncovered edges for level l is defined as $U = \{(r, p, v) \in O | \nexists C_i$ s.t.*

$d(r, c_i) \leq r_i$ and $d(v, c_i) \leq r_i)\}$. Intuitively, U is the set of edges that do not have both the subject and the value in the same circle.

- *The overlap for level l is defined as $overlap(l) = \frac{|L|}{|N(O)|}$, where $N(O)$ is the set of vertices in the database O.*

- *The coverage for level l is defined as $coverage(l) = \frac{|O|-|U|}{|O|}$, where $|O|$ is the number of triples in the database O.*

The concepts of coverage and overlap are conceptually related to their counterparts in R-trees [23], but will be used quite differently in **GRIN**.

Example 5.11. *Consider the GRIN index in Figure 5.3. At level 2 of the index, all vertices except* 8 *and* 12/15/06 *are not overlapped, hence $overlap(2) = \frac{18}{20} = \frac{9}{10}$. The only edges that are not covered are* (Review#16742, rating, 8) *and* (Review#16742, date, 12/15/06)*. Therefore $coverage(2) = \frac{27}{29}$.*

The following two results show that index structures with high coverage and low overlap can process queries faster. Intuitively, high coverage means there is a greater chance that an arbitrary query will "hit" inside a single circle on a given level l. Similarly, low overlap leads to larger circles, hence to less efficient queries.

Proposition 5.12. *Let G_1, G_2 be two GRIN index trees of equal heights h over the same database O, such that for all levels $0 < l < h$, $coverage_{G_1}(l) < coverage_{G_2}(l)$, where $coverage_{G_i}$ denotes the coverage in the index G_i. Let q be an arbitrary but fixed query over O. Then the probability that q executes faster on G_2 is higher than 0.5.*

Proof. Let l be an arbitrary level. The probability that an edge in query q will match one of the uncovered edges on level l in G_i is equal to the number of edges in the query $|q|$ divided by the number of uncovered edges, $|O| - coverage_{G_i}(l) * |O|$. Hence the probability that the query q will be answered at a level $l' > l$ is $\frac{|q|}{|O| - coverage_{G_i} * |O|}$. Since $coverage_{G_1}(l) < coverage_{G_2}(l)$ for all levels l except the root and leaf nodes, for any arbitrary l it is more likely that the query q will be answered at level l in G_2 than in G_1.$\square$

Proposition 5.13. *Let G_1, G_2 be two GRIN index trees of equal heights h over the same database O, such that for all levels $0 < l < h$, $overlap_{G_1}(l) < overlap_{G_2}(l)$, where $overlap_{G_i}$ denotes the overlap in the index G_i. Let q be an arbitrary but fixed query over O such that the smallest circle containing the answer to the query is on level j in both G_1 and G_2. Then the probability that q is executes faster on G_1 is higher than 0.5.*

Proof. Let l be an arbitrary level and let C_i be an arbitrary but fixed circle on level l of G_i. Let m_l be the number of circles on level l. The probability that a random vertex belongs to the circle C is $p(x \in C) \geq \frac{1}{m_l} + \frac{1}{coverage_{G_i}(l)} \cdot \frac{1}{m_l - 1}$. Intuitively, the larger the overlap, the higher the probability $p(x \in C)$. As a consequence, an arbitrary circle in G_1 is less likely to contain more vertices than an arbitrary circle in G_2. Therefore, it is likely that q's smallest circle is larger in G_2 than in G_1. Since query execution time is monotonic with the size of the smallest circle, q is likely to execute faster on G_1.$\square$.

Since high coverage and low overlap are desirable, we investigated the possibil-

ity of optimizing either one when building the index bottom up. Let us assume that $C_1, \ldots, C_m$ are the circles at level l. We would like to build the parent circles on level $l+1$ such that we maximize coverage and/or minimize overlap. The following result states that minimizing overlap is NP-complete.

Theorem 5.14. *Let $\mathcal{C} = \{C_1, \ldots, C_m\}$ be a set of circles at level l in the index. The problem of finding $\frac{2m}{M}$ parent circles at level $l+1$ such that overlap$(l+1)$ is minimal is NP-complete.*

Proof. We will prove the theorem through reduction from the known NP-complete problem of maximal cut through a weighted graph. For simplicity, we will assume we must find only two parent circles. The general case of $\frac{2m}{M}$ parent circles can be solved by applying the two parent-circle problem repeatedly.

Let $H = (V, E, \lambda)$ be a connected weighted graph, where V is the set of vertices and $|V| = m$, E is the set of edges and $\lambda : E \rightarrow \Re_+$ is the edge labeling function. The mincut problem is that of finding a set of edges $E' \subseteq E$ such that after removing E' from H, H contains exactly two connected components and $\sum_{e \in E'} \lambda(e)$ is maximal. We reduce maximal cut to our theorem in the following way. We want to associate a circle to each vertex in H, such that $\forall v_1, v_2 \in V$, $\lambda(v_1, v_2)$ is the overlap between the circles corresponding to v_1 and v_2 respectively. Let $C(v)$ denote the circle associated with a vertex $v \in V$. For each $(v_1, v_2) \in E$, we write the constraints $|N_{C(v_1)}| \geq \lambda(v_1, v_2)$ and $|N_{C(v_1)}| \geq \lambda(v_1, v_2)$. The set of inequations has an infinity of solutions, hence we can associate circles to each vertex in H, which completes our reduction.□

Since optimizing overlap is NP-complete, the best we can hope to do is a near-optimal solution. We empirically observed the following:

1. Overlap is almost always high near the root of the tree, where there are fewer but larger circles.

2. Coverage will always be low near the bottom of the tree, where there are many circles. Since we assumed O is connected (otherwise we can build separate index structures for its connected components), then the number of uncovered edges on level l is at least equal to the number of circles on l minus 1. For a fully balanced **GRIN** tree, level l contains $\left(\frac{M}{2}\right)^l$ circles, hence there are at least $\left(\frac{M}{2}\right)^l - 1$ uncovered edges.

These observations suggest that we should focus on improving overlap where it is possible, i.e., near the leaves and emphasize coverage near the root of the index tree. We accomplish this by optimizing a linear combination of coverage and overlap called a *signature*. Let C_1, C_2 be two circles on level l of a index of height h. We overload notation and denote by $overlap(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$ and by $coverage(C_1, C_2)$ the percentage of covered edges between C_1 and C_2. The signature is defined as $signature(C_1, C_2) = \frac{l}{h} \cdot \frac{1}{overlap(C_1, C_2)} + \frac{h-l}{h} \cdot coverage(C_1, C_2)$. Note that near the leaf level, the signature almost entirely depends on overlap, whereas near the root it almost entirely depends on coverage. Our goal is the minimize the signature of a level as much as possible. We introduce an updated index construction algorithm in Figure 5.6.

The $GRINBuildOptimized$ algorithm is very similar in concept to $GRINBuild$.

Algorithm GRINBuildOptimized$(C,\ Max,\ O)$

Input: C is the number of leaf nodes, Max is the maximum number of vertices on a page, O is the RDF database.

Output: The GRIN index structure G.

1: $L_0 \leftarrow PAM(O, C, Max)$
2: Create leaf nodes in G from clusters in L_0
3: **for** $i \in [0, \lceil \log_M C - 1 \rceil]$ **do**
4: $\quad L_{i+1} \leftarrow \emptyset$
5: $\quad$ Create a new labeled undirected graph $H = (V, V \times V, \lambda)$
6: $\quad V \leftarrow L_i$
7: $\quad$ **for** $m, n \in V$, $m \neq n$ **do**
8: $\quad\quad \lambda(m, n) \leftarrow signatureL_i$
9: $\quad$ **end for**
10: $\quad$ Eliminate any edges from H labeled with 0.
11: $\quad$ Compute the maximum $\lfloor \frac{M}{2} \rfloor$-cut through H using Goemans-Williamson
12: $\quad$ **for** each $V_j \subseteq L_i$ connected component **do**
13: $\quad\quad$ Create node $p = (c, r)$ in G as a parent of the circles $C_k \in V_j$
14: $\quad\quad L_{i+1} \leftarrow L_{i+1} \cup \{p\}$
15: $\quad$ **end for**
16: $\quad$ Add level L_{i+1} to G
17: **end for**
18: **return** G

Figure 5.6: An algorithm to build the optimized GRIN index

Instead of using the inter-cluster distance to find the closest neighbors of a randomly selected circle, the new algorithm builds a graph in which each vertex is a circle on the current level and each edge between two circles is labeled with the signature of the two circles. In order to minimize the signatures on the next level, we should compute a number of maximum cuts through the graph that create a number of connected components. The circles in each of the connected components will have a common parent. Since maximal cut is a NP-complete algorithm, we use the approximation algorithm discovered by Goemans and Williamson [18]. The approximation factor of the algorithm is 0.872, which was the best approximation factor for the maxcut problem we have found available.

5.5 Handling updates

Suppose now that the RDF database O is updated by inserting a new triple (r, p, v). Since we are only considering connected graphs, this means at least one of r or v should already be in the database. Since both cases are identical from the point of view of the distance metric, we will assume r is already in O. To maintain the properties of the **GRIN** index tree, we would like to insert v into the cluster C such that $\frac{\sum_{x \in C} d(x,v)}{|C|}$ is minimized, i.e., C is on the average the cluster "closest" to v. We can do this by simply traversing the graph in a depth-first fashion, at each step choosing the child circle whose center is closest to v.

Once the desired cluster has been located, if there is space for v, we simply insert it into the cluster and then adjust the radii for all circles from C to the root of the index. However, in case cluster C is full, then we have to split C into two distinct clusters. This may cause the parent of C to split, if it already has M children. The split can propagate all the way to the root, but it cannot go further since the root can have more than M children. The insertion maintenance algorithm for **GRIN** is presented in Figure 5.7. Note that when an inner node must be split, the simplest what to ensure the maintenance of child/parent containment property is to simply erase the inner node and allow its child nodes to form two new parents.

In case of a deleted triple (r, p, v), let us assume that r will be the resource that will remain in O (if no resource remains in O, the graph was disconnected). If the cluster containing v is now empty, we must propagate a notification to its parent that the cluster has disappeared. If the parent remains with less than $\frac{M}{2}$ children

method adjustUpwards(n_I)

Input: Index node n_I

1: **if** n_I is the root **then**
2: **return**
3: **else**
4: recompute the radius of n_I
5: adjustUpwards(parent(n_I))
6: **end if**

method splitUpwards(n_I, X)

Input: Index node n_I, newly created child X

1: **if** n_I is the root **then**
2: add X to the children of n_I
3: **else**
4: **if** $|children(n_I)| < M$ **then**
5: Add X to the children of n_I
6: adjustUpwards(n_I)
7: **else** {This node should be split as well}
8: Re-cluster $children(n_I) \cup \{X\}$ into two parent nodes n'_I, n''_I
9: Add n'_I to $children(parent(n_I))$
10: **end if**
11: $adjustUpwards(parent(n_I))$
12: **end if**

Algorithm GRINMaintainInsert(O, G, (r, p, v))

Input: RDF database O, GRIN index G and newly inserted triple (r, p, v)
Output: An updated index G.

1: $X \leftarrow root(G)$
2: **while** X is not a leaf node **do**
3: **if** the children of X are leaves **then**
4: $X \leftarrow$ the child C of X that minimizes $\frac{\sum_{x \in C} d(x,v)}{|C|}$
5: **else**
6: $X \leftarrow$ the child (c, r) of X that minimizes $d(c, v)$
7: **end if**
8: **end while**{We have found the desired cluster}
9: **if** $|X| \leq Max$ **then**
10: Add v to X
11: adjustUpwards(parent(X))
12: **else**
13: Add v to X
14: Split X into two equal clusters X', X'' such that $signature(X', X'')$ is minimized
15: Add X' to the $children(parent(X))$
16: $splitUpwards(parent(X), X')$
17: **end if**

Figure 5.7: **GRIN** insert maintenance

method adjustUpwards(n_I)

Input: Index node n_I

1: **if** n_I is the root **then**
2:     **return**
3: **else**
4:     recompute the radius of n_I
5:     adjustUpwards(parent(n_I))
6: **end if**

method coalesceUpwards(n_I, X)

Input: Index node n_I, newly deleted child X

1: **if** n_I is the root **then**
2:     remove X from $children(n_I)$
3: **else**
4:     remove X from $children(n_I)$
5:     **if** $|children(n_I)| = 0$ **then**
6:         $coalesceUpwards(parent(n_I), X)$
7:     **else if** $|children(n_I)| < \frac{M}{2}$ **then**
8:         Let n'_I be the circle that maximizes $signature(n_I, n'_I)$
9:         Merge n'_I into n_I
10:         **for** $Y \in children(n'_I)$ **do**
11:             $splitUpwards(n_I, Y)$
12:         **end for**
13:     **else**
14:         $adjustUpwards(parent(n_I))$
15:     **end if**
16: **end if**

Algorithm GRINMaintainDelete(O, G, (r, p, v))

Input: RDF database O, GRIN index G and deleted (r, p, v)
Output: An updated index G.

1: $X \leftarrow$ leaf node containing v
2: **if** $|X| > 1$ **then**
3:     remove v from X
4:     $adjustUpwards(parent(X))$
5: **else**
6:     remove v from X
7:     $coalesceUpwards(X)$
8: **end if**

Figure 5.8: **GRIN** delete maintenance

clusters, then it must also seek to coalesce with another node on the same level. The deletion maintenance algorithm for **GRIN** is depicted in Figure 5.8. In case two inner nodes have to coalesce (according to the signature minimization rules), it might be the case that the total number of children after the merge is greater than M. When we merge n_I and n'_I, we simply add the children of n'_I to n_I one by one. One n_I is full, adding the next child will trigger a new split.

5.6 Extending GRIN to aRDF

In this section, we show how **GRIN** can be extended to aRDF datasets. The **GRIN** index structure was built on the idea that resources that are "close" in the graph are more likely to appear together in a query answer and therefore should be stored on the same page (preferable in the same index node). However, in an aRDF triple, "close" can have two meanings: close in terms of the distance in the aRDF graph or "close" in terms of the the annotation values. For instance, when using $\mathcal{A}_{time-int}$, intuitively triples whose validity intervals are close would be more likely to appear together in the same answer than two triples who are temporally far apart. When handling aRDF, we must take into account both the graph-based distance metric and the distance between the annotations on the edges.

5.6.1 Distance metrics for aRDF

Graph distance metric. We can use either the shortest or the longest path in the undirected RDF graph as the graph distance metric $d_G(\cdot, \cdot)$. We observed

empirically that the shortest path gives better performance for queries and therefore omit the longest path from the rest of the discussion.

Annotation distance metric. The annotation distance metrics combine the distance between consecutive annotations path between two resources. First, we need to characterize the distance, $\delta(a, a')$, between two annotation values $a, a' \in \mathcal{A}$. We require that δ satisfies the following axioms:

1. $\delta(a, a) = 0$.

2. $\delta(a, a') = \delta(a', a)$.

3. $\delta(a, a') \leq \delta(b, b')$ if $b \preceq b$, $b' \preceq a'$

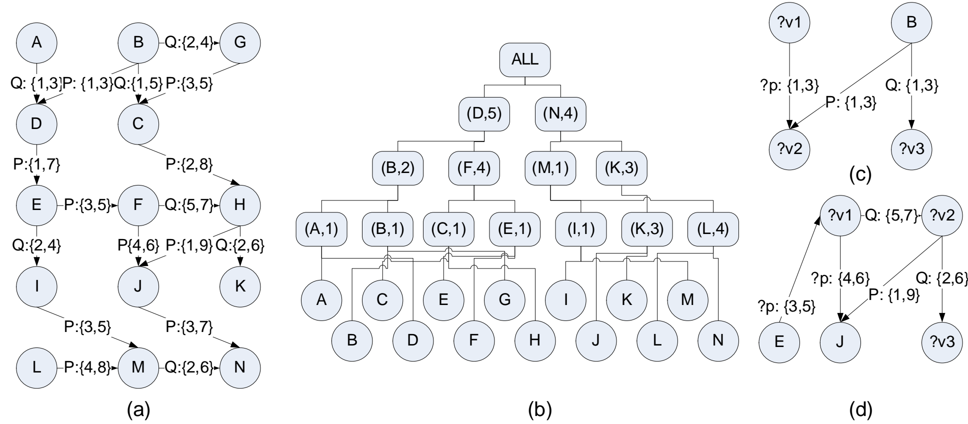


Figure 5.9: (a) Synthetic aRDF database with $\mathcal{A}_{time-int}$; (b) Example **GRIN** index; (c), (d) Example aRDF queries.

There are numerous distance functions that satisfy these properties. For instance, in the case of $\mathcal{A}_{time-int}$, if T_i and T_j are temporal intervals, any of the following are acceptable δ functions ($T_i.s$ is the start point of interval T_i and $T_i.e$ is its end timepoint):

1. $\delta(T_i, T_j) = \left| \frac{T_i.e - T_i.s}{2} - \frac{T_j.e - T_j.s}{2} \right|$ (interval centers).
2. $\delta(T_i, T_j) = |T_i.s - T_j.s|$ (start points).
3. $\delta(T_i, T_j) = |T_i.e - T_j.e|$ (end points).
4. $\delta(T_i, T_j) = |T_i.s - T_j.e|$ if $T_i \preceq T_j$, otherwise $\delta(T_i, T_j) = |T_j.s - T_i.e|$ (leftmost and rightmost point).

In the general case of a lattice $(\mathcal{A}, \preceq)$, a possible $\delta(a, a')$ can be defined as the distance between a and a' in the lattice if a is comparable to a' or the sum of the distances from a and a' to their least upper bound otherwise. Given a δ, we can then define an annotation metric as follows.

Definition 5.15 (Annotation distance metric)**.** *Let O be a aRDF database, $x, y \in \mathcal{R}$, $p = (e_1, \ldots, e_n)$ be a path between x and y in the undirected aRDF graph, and a_j be the annotation on the edge e_j. If $n = 1$, then we define $d_a^p(x, y) = 0$. Otherwise, we define $d_a^p(x, y) = \sum_{j \in [2,n]} \delta(a_j, a_{j-1})$. Finally, the annotation distance between x and y is the minimum over all the possible paths $d_a(x, y) = \min_p(d_a^p(x, y))$.*

GRIN distance metric. Since both $d_G(\cdot, \cdot)$ and $d_a(\cdot, \cdot)$ are metrics, we can use a norm function to produce a single metric $d(\cdot, \cdot)$. For **GRIN**, we use the k-norm $d(x, y) = [(d_G(x, y))^k + (d_a(x, y))^k]^{\frac{1}{k}}$. We will discuss the choice of k in Section 5.7.

Once we defined the distance metric, we can use $GRINBuild$ and $GRINAnswer$ in the same way we did for standard **GRIN**. We illustrate the process through an example.

Example 5.16. *Consider the graph in Figure 5.9(a). Let δ be defined as the distance between interval center points, d_G be the shortest path distance and $k = 1$. Clearly,*

$d_G(B, F) = 3$. There are two different paths between B and F: $\{B, D, E, F\}$ and $\{B, C, H, F\}$. On the first path, $d_a(B, F) = 2$ and on the second $d_a(B, F) = 3$. We take the minimum and obtain $d(B, F) = 4$.

Consider the example query in Figure 5.9(c). To the following set of constraints can be derived from the query: $d(?v1, B) \leq 2)$, $d(?v2, B) \leq 1$, $d(?v3, B) \leq 1$. For the query in Figure 5.9(d), we can deduce the following (not a complete list): $d(?v1, E) \leq 1$, $d(?v2, J) \leq 1$ and $d(?v3, J) \leq 3$.

We will now show how how query processing works for aRDF *on the data and queries in Figure 5.9. For the query in Figure 5.9(c), we start at the root node and recursively call $GRINAnswer$ on the child index nodes until we reach $(B, 2)$. From cons(q) we already know that $d(?v1, B) \leq 2)$, $d(?v2, B) \leq 1$ and $d(?v3, B) \leq 1$, hence all variables are in the circle centered in B of radius 2. By running the subgraph matching algorithm on this portion of the graph (which contains vertices $\{A, B, C, D, G\}$), we obtain the answer to the query: $?v1 \leftarrow A$, $?v2 \leftarrow D$, $?v3 \leftarrow C$ and $?p \leftarrow P$.*

The processing of the query in Figure 5.9(d) is similar – once we recursively reach the analysis for the node $(F, 4)$, we see that none of the children satisfies the constraints for ***all*** *the variables. From $d(?v1, E) \leq 1$ and $d(E, F) = 1$ we have $d(?v1, F) \leq 2$, from $d(?v2, J) \leq 1$ and $d(F, J) = 1$ we deduce that $d(?v2, F) \leq 2$ and from $d(?v3, J) \leq 3$ and $d(J, F) = 1$ we obtain that $d(?v3, F) \leq 4$. Since $?v1$, $?v2$, $?v3$ are all in $(F, 4)$, we can apply the subgraph matching step of the algorithm on $(F, 4)$ and obtain $?v1 \leftarrow F$, $?v2 \leftarrow H$, $?v3 \leftarrow K$ and $?p \leftarrow P$.*

5.6.2 Queries with transitive properties

So far, we have assumed that all the transitive inferences in the database O had been performed apriori. In this section, we show how to extend $GRINAnswer$ to handle transitivity inferences while it is processing a query. Let q be the query that refers to a transitive properties. For simplicity, we will assume that only one atom q_t of q involves a transitive property. q_t can be one of the following (x, y, z are constants).

$q_t = (x, y, z)$. In this case, since y is transitive we simply perform all transitive inferences on y starting at x. If z is reached, then this query atom is already entailed by the RDF graph. Otherwise, q_t cannot be entailed by the RDF graph and thus the answer to q is $\emptyset$.

$q_t = (x, ?v, z)$. In this case, we simply perform all transitive inferences starting at x on all its adjacent transitive properties. If for any such property p, we infer (x, p, z), then p is a possible substitution for the variable $?v$.

$q_t = (x, y, ?v)$ (this case is symmetrical to $q_t = (?v, y, z)$). In this case, we simply perform all the transitive inferences starting from x on property y until there are no more inferences to be performed. Any resources r that is reachable from x via $y - paths$ is a possible substitution for $?v$.

$q_t = (x, ?v, ?v')$ (symmetrical to $q_t = (?v, ?v', z)$). In this case,

1. For any triple $(x, p, r) \in O$, p is a possible substitution for $?v$ and r a possible substitution for $?v'$.

2. Perform one transitive inference for any transitive property adjacent to x.

Repeat from 1.

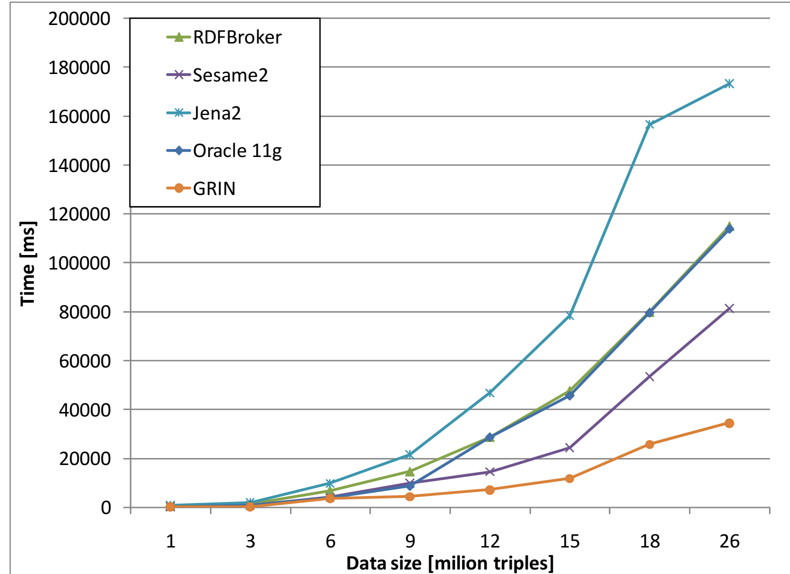
5.7 Experimental evaluation

GRIN was implemented in Java and on a Core2 Duo 2.8Ghz processor machine with 3GB of RAM running the openSuse 10.2 operating system. All running times reported in this section are averaged over three independent executions using warm caches. We compared the performance of **GRIN** against some of the leading RDF storage systems: Jena2, Sesame2, Oracle 11g, RDFBroker, 3store[5] and the column store LucidDB. Besides the standard version of **GRIN**, we also evaluated an implementation for data annotated with temporal validity intervals (we labeled this approach **tGRIN** [47]). **tGRIN** uses the aRDF-Ordering algorithm to find the answers to queries.

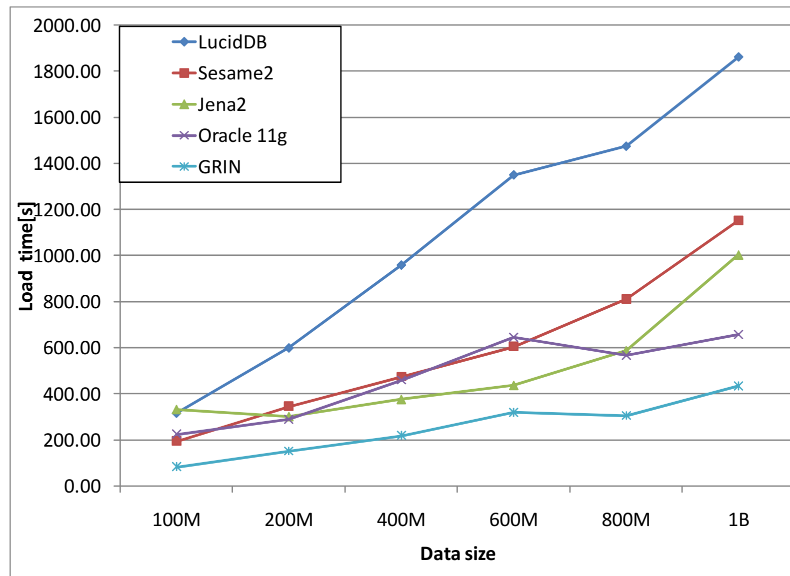
We evaluated our approach on the real-world dataset GovTrack consisting of approximately 26 million triples and a series of datasets generated with the Lehigh University Benchmark (LUBM) up to 1 million triples. For **tGRIN**, we converted the GovTrack dataset to an aRDF database using the $\mathcal{A}_{time-int}$ annotation in the same way as in Chapter 4. When comparing **tGRIN** to some of the competing systems, we enhanced their relational database backend with a set of temporal index structures including R-trees, SR-trees, the ST-index and finally the simple reified format of data. For each system, we report the variant that performed the best.

In all our experiments, **GRIN** keeps references to RDF resources and triples

[5]3store was not particularly efficient for standard RDF queries and hence its results are only included for the aRDF dataset.



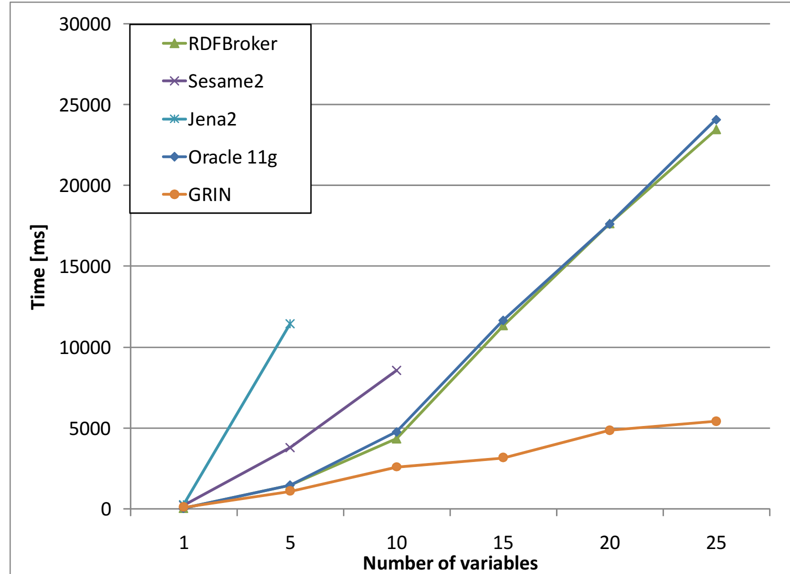
(a) GovTrack dataset.



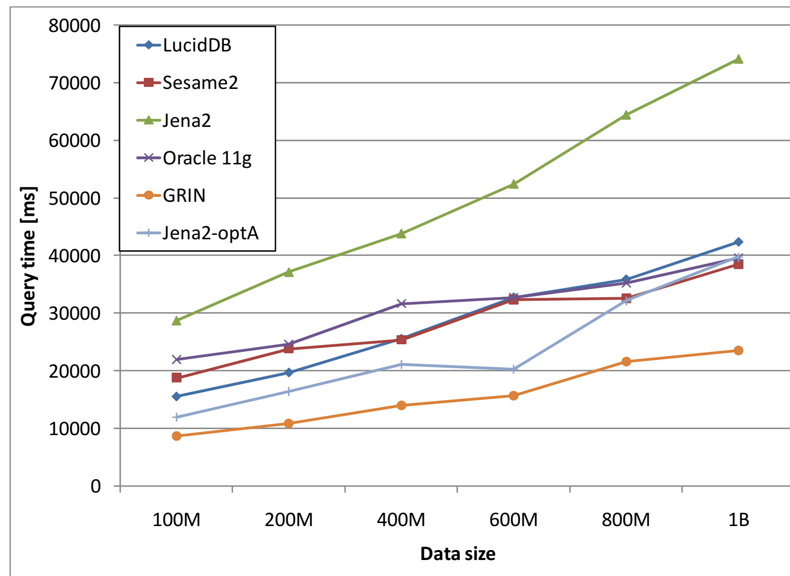
(b) LUBM dataset.

Figure 5.10: **GRIN** load time

that are stored in a PostgreSQL 8.0 database using the same schema as Jena2. The index itself is stored on disk in a set of separate files; reported running times include disk I/O. We chose our queries for GovTrack from the list of frequent queries available at `www.govtrack.us` depending on the requirements of each experiment.



(a) GovTrack dataset.

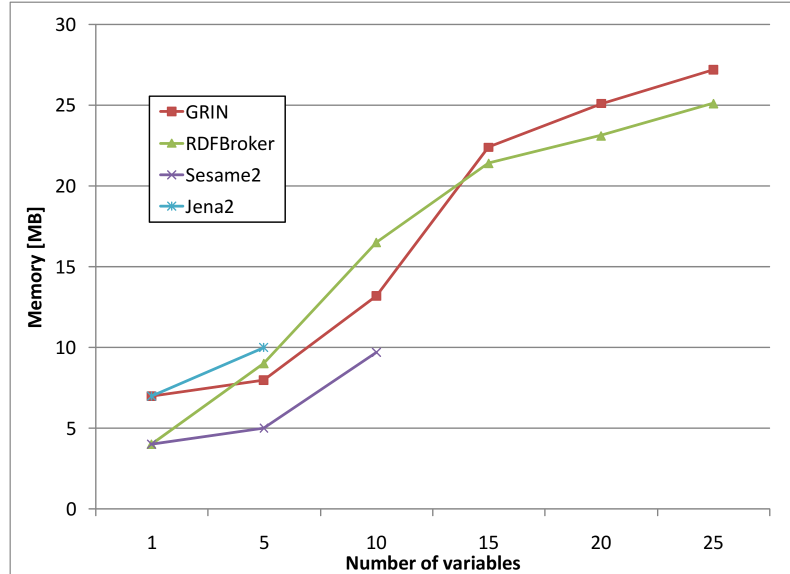


(b) LUBM dataset.

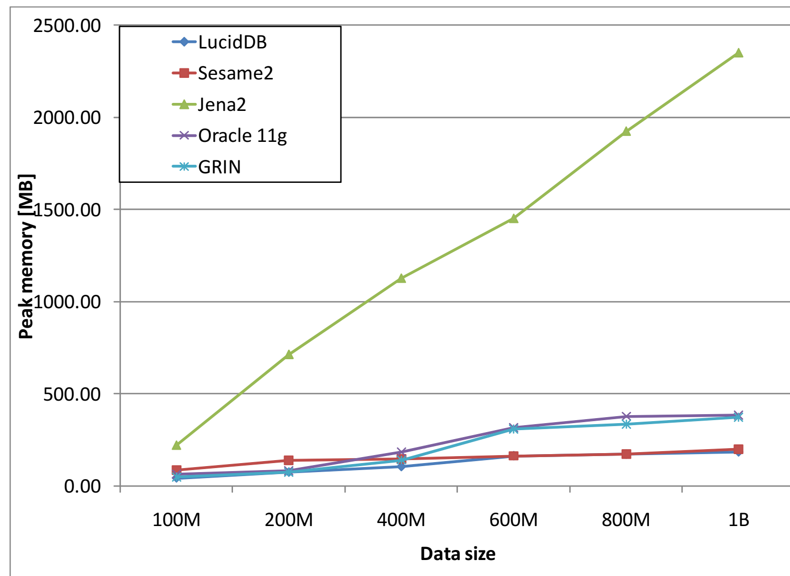
Figure 5.11: **GRIN** query processing time

The queries for the LUBM dataset are generated at random to match the selectivity and/or query complexity criteria of the evaluation.

In our first set of experiments we measured the index load time for increasingly larger subsets of both the GovTrack dataset (Figure 5.10(a)) and the LUBM dataset



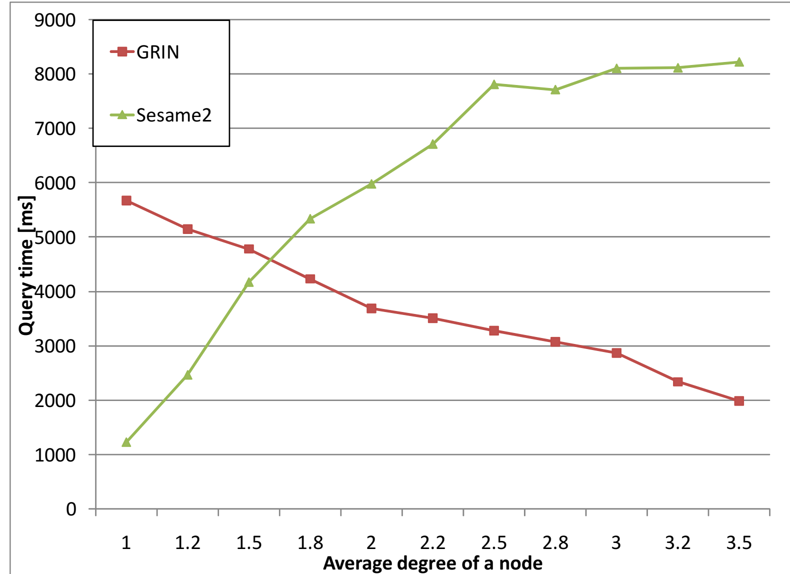
(a) GovTrack dataset.



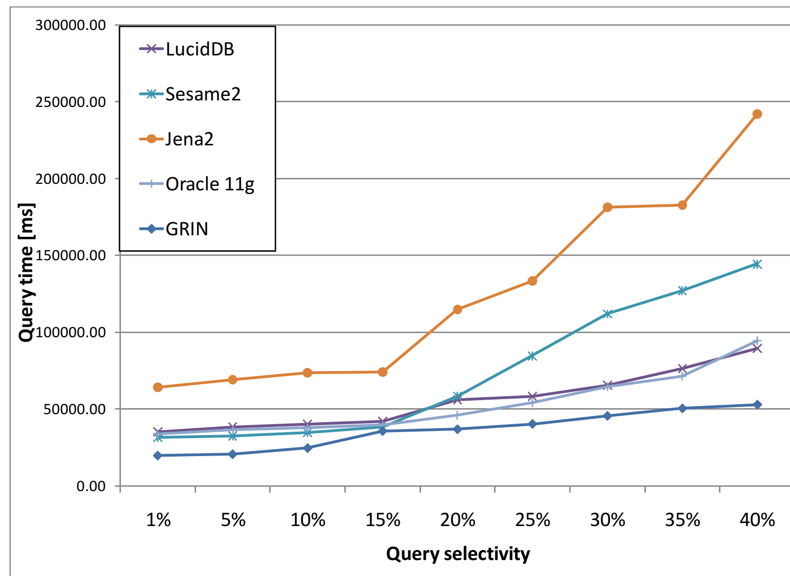
(b) LUBM dataset.

Figure 5.12: **GRIN** peak memory usage

(Figure 5.10(b)). We notice that in both cases, **GRIN** creates the index faster than the other systems. RDFBroker was not able to load datasets above 200MB for the LUBM benchmark and is therefore omitted. **GRIN** was able to create the index for the entire 1 billion triple dataset in approximately 415 seconds.



(a) Average degree of a node (number of constraints in the query).



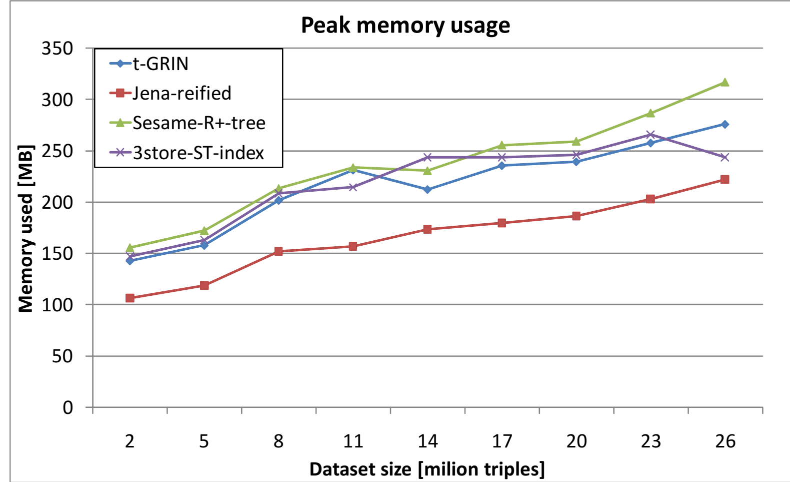
(b) Selectivity.

Figure 5.13: Selectivity and number of constraints analysis

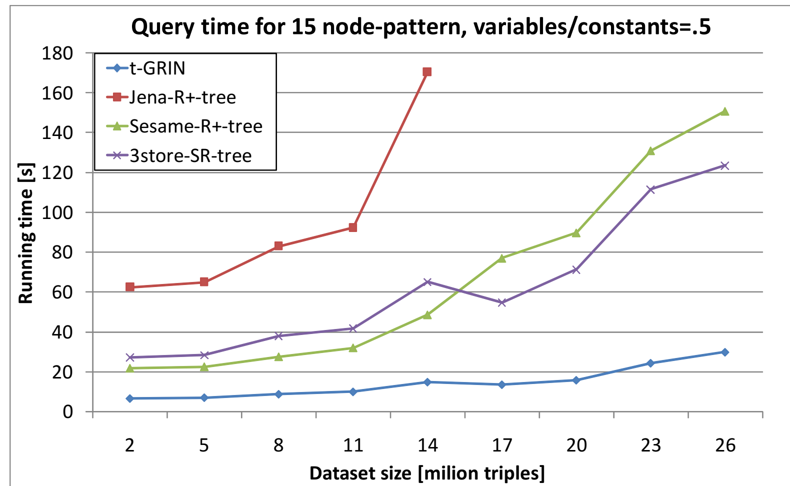
Next, we selected 50 queries of selectivities between 10 and 20% (5–10 variables) for the LUBM dataset and 50 queries of the same selectivities, but with an increasing number of variables up to 25. We show the dependence of the query processing time on the complexity of the query (number of variables) and data size are

given in Figure 5.11(a) and (b) respectively. We observed that **GRIN** outperformed all the other systems, but the difference in processing time seems to remain relatively constant for larger database sizes. On the other hand, when we increase the number of variables from 5 to 25 for a relatively smaller dataset (Figure 5.11(a)), some systems such as Jena2 and Sesame2 unexpectedly crash for more than 10 variables. Moreover, their scalability in terms of increased query complexity is much poorer than that of **GRIN**. We should also point out that, as we described in Section 4.3, queries with 25 variables are quite common among the existing datasets. We observe similar trends for LUBM as in the case of query processing time. However, an increase in the number of variables seems to have a much greater effect on memory requirements than the increase in database size. At this stage, we also measured the peak memory consumption and plotted its dependence on the query complexity (number of variables) for GovTrack and on the data size for LUBM. The results are shown in Figure 5.12(a) and (b). **GRIN** uses slightly more memory than the other systems for queries of increased complexity. This is due to the fact that we keep intermediate results in memory, whereas RDFBroker stores intermediate results in the database.

In the next set of experiments we chose a set of 20 queries for GovTrack and 20 queries for each selectivity level between 5 and 40% in 5% increments for LUBM. The purpose of the GovTrack experiment was to keep the number of the variables in the query constant, but vary the number of constraints – i.e., the number of edges in the query. As the number of constraints increases, we expect the query processing time to decrease. However, as Figure 5.13(a) shows, this is not case for Sesame2.



(a) Memory peak usage.

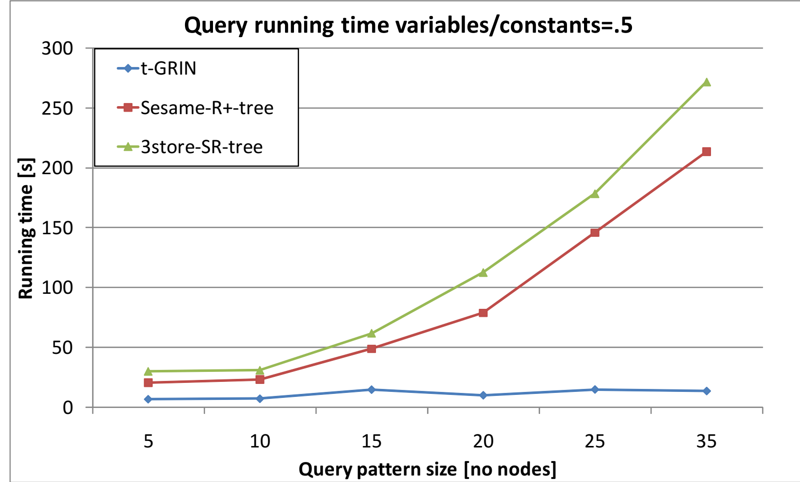


(b) Query processing time.

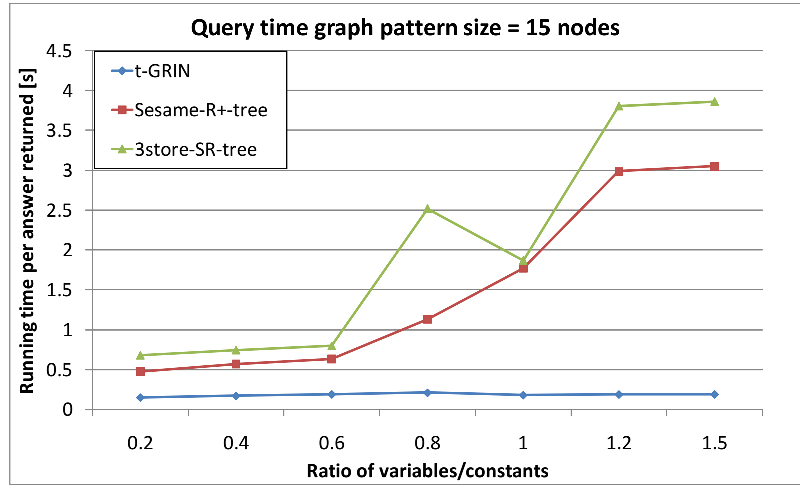
Figure 5.14: **tGRIN** memory requirements and query processing time

The other relational-backed systems exhibited the same trends. This unexpected behavior can be traced to the SPARQL-to-SQL translation. Figure 5.13(b) shows the dependence of the query processing time on the selectivity of the query. Most systems were relatively stable; **GRIN** in particular seems to be affected the least by query selectivity.

In the next experiment, we measure the memory requirements, query processing time, dependence on query size and on the variable/constant ratio for **tGRIN**



(a) Dependence on query size.



(b) Dependence on variable/constant ratio.

Figure 5.15: **tGRIN** performance for complex queries

(**GRIN** for aRDF annotated with temporal intervals). The results are shown in Figure 5.14 and Figure 5.15. We observed that **tGRIN** gives a significantly better response time than any of the other systems; we traced this back to the use of the aRDF-Ordering algorithm to answer conjunctive queries. Figure 5.14(b) shows the scalability of **tGRIN** with respect to database size, whereas Figure 5.15(a) and (b) show its scalability when query complexity increases. We used two measures of query complexity. In Figure 5.15(a), we increase the query size, whereas in Figure

5.15(b) we maintain the query size constant, but we gradually increase the number of variables in the query. As in the previous experiments, **tGRIN** is not very much affected by the increase in query size. The increased complexity of larger queries is offset by the fact that these queries have more edges, hence more constraints can be derived from the query. The same observation applies to the dependence on the variable/constants ratio. We remind the reader that only a few constants can be sufficient for **tGRIN** to quickly locate a small portion of the database that is guaranteed to contain the answers.

Chapter 6

Conclusions and future work

In the previous chapters, we have identified some of the critical factors that affect scalability in RDF databases. We presented significant contributions at three different levels of a scalable RDF database system.

At the knowledge representation level, we have empirically identified reification as a technique that causes the degradation of query performance in existing RDF database systems. Studying several real-world datasets led to the conclusion that reification can be avoided by allowing triples to be annotated with values from a partial order. We introduced a new representation language, Annotated RDF, which generalizes the RDF syntax and model theoretic semantics to allow triple annotations and transitive properties in the database. We developed a SPARQL-like query language for Annotated RDF and defined its formal semantics.

At the query processing level, we defined an extensive set of query algorithms covering atomic and general conjunctive queries; all our algorithms were proved to be correct and their worst-case complexity analyzed. Many of the real-world datasets we encountered had a large set of frequent queries; for such scenarios, it is often preferable to store the answer to a frequent query and recompute its answer incrementally when the database is updated. This led to the development of view maintenance algorithms that can handle insertions into and deletions from

the database. Last, we evaluated our algorithms and compared to the leading RDF database systems on both real-world and benchmark datasets. Annotated RDF proved to be more efficient in answering queries than Oracle 11g, Jena2 and Sesame2.

At the indexing level, we introduced the **GRIN** index structure, an innovative way to reduce the search space when answering SPARQL queries. We provided algorithms for index construction and query processing and presented a theoretical analysis of index optimization, as well as an algorithm for building coverage and overlap-optimized index structures. **GRIN** was easily extensible to the Annotated RDF semantics. The empirical evaluation of our methods on both of real-world and benchmark datasets showed that **GRIN** scales well to massive amounts of data and to queries of high complexity.

6.1 Future work

In this dissertation, we have layed the groundwork for making Semantic Web databases scalable. However, to achieve the same success as relational databases, this research should be the starting point of a larger effort. Furthermore, our techniques need to be extended to distributed RDF database systems and to OWL ontologies.

The first problem that has to be addressed is that of RDF and OWL *ontology design and quality*. Unlike the relational systems, where the normal forms used in database normalization provide an indicator of the quality of the database, ontology systems do not have a technique for estimating the quality of a database. Also,

there is no established community practice on how ontologies should be designed. Guarino and Welty [19] were the first to give a methodology for establishing which real-world concepts should be classes and which should be properties. This duality is quite problematic in many ontologies because it may impede inference capabilities. The main issue with providing guidelines on designing ontologies is that ontology quality can only be expressed in terms of their intended semantics. There are two possible avenues of investigation. First, the possibility of designing an upper ontology (such as DOLCE – `www.loa-cnr.it/ontologies`) that encompasses those general concepts and properties that span many domains. Such an ontology can be designed with care by a group of experts, in the hope that domain specific ontologies that import it will be more inclined to use the same good design practices. Although it is unlikely that a single such ontology will exist, having several ontologies representing different views of the world is achievable. The second avenue of investigation is that of looking at the structural properties of class and property taxonomies in ontologies and suggest a set of good design practices, similar to good coding practices. This can only be done with a community-wide effort. One design practice we can suggest at this time refers to datasets that are represented in RDF or OWL although they are very well suited for a direct relational representation. Examples abound, but we will mention just a few:

- The CIA World Factbook[1] is essentially a relational dataset. However, it is often transformed to RDF and queried via SPARQL for research and application demonstrations.

[1] `http://www.aktors.org/interns/2005/cia/index.php`

- `www.rdfdata.org` contains a list of bibliographical RDF datasets. However, the large majority of these have a simple structure that is easily representable in a relational database. Entries generally list a paper, linked to a set of values containing its authors, title, and so on. The author names themselves are literals, which means papers will generally not be connected in the graph representation of the dataset.

The second issue we see in the near future is that of RDF query optimization. In this dissertation, we have shown that the solutions for RDF scalability are generally very different than the methods for the relational model. The same observation applies to query optimization, which has two essential problems. The first problem is that of data partitioning. Unlike relations, which can be partitioned horizontally or vertically, partitioning RDF graphs is more complex. One way of doing this is to split the set of properties in the database and then take the subgraphs on each set of properties as a partition. Finding the split of the set of properties has to be done in a principled way that accounts for the potential costs of the split. This leads into the second problem of query optimization, which is finding good cost estimate functions for queries. A cost function should allow us to estimate both the cost of running query components – for instance, by estimating their selectivity – and to estimate the cost of joining the results. Good cost functions should be based on probabilistic models of RDF graph neighborhoods that take into account the estimated values of resource in- and out-degrees, the density of triples and possibly historical information about previous queries. The cost modeling framework we just

described should also be coupled with low-level access estimates such as disk access cost and number of page faults. Very recently, Stocker et al. [50] made the first steps in this direction by proposing very recently a query optimization model than builds on traditional relational optimization techniques to take into account the structure of RDF data. Unlike relations, which can be partitioned horizontally or vertically, partitioning RDF graphs is more complex. One way of doing this is to split the set of properties in the database and then take the subgraphs on each set of properties as a partition. Finding the split of the set of properties has to be done in a principled way that accounts for the potential costs of the split. This leads into the second problem of query optimization, which is finding good cost estimate functions for queries. A cost function should allow us to estimate both the cost of running query components – for instance, by estimating their selectivity – and to estimate the cost of joining the results. Good cost functions should be based on probabilistic models of RDF graph neighborhoods that take into account the estimated values of resource in- and out-degrees, the density of triples and possibly historical information about previous queries. The cost modeling framework we just described should also be coupled with low-level access estimates such as disk access cost and number of page faults. Very recently, Stocker et al. [50] made the first steps in this direction by proposing very recently a query optimization model than builds on traditional relational optimization techniques to take into account the structure of RDF data.

The third item of future work is related to the types of RDF databases that will develop over time. We hypothesize that in the near future we will see a pattern of development similar to that of relational databases. Specifically, we will see a

combination of large RDF databases deployed in production environments, as well as a number of smaller, distributed RDF databases. While we have shown successful approaches for the former, we have yet to discuss the latter. A distributed RDF database systems poses two important challenges:

1. *Schema mapping.* Although any merge of two RDF databases is consistent (with the exception of potential data type clashes), it is still helpful to merge resources from the two databases if they refer to the same real-world entity. This has the clear advantage of improving query recall. So far, RDF and OWL schema mapping algorithms have very high complexity, taking minutes, even hours to complete. Such complexity impedes them from being used in a distributed system setting. A clearer framework that provides quality and running time guarantees for RDF and OWL mapping is a definite requirement.

2. *Query re-writing.* Assuming good schema mapping has been achieved, queries will have to be re-written once they are transmitted from one processing node to another. Query re-writing has to take into account the cost models we previously discussed to minimize the computational costs of the re-written query.

Last, but not least, our techniques will have to be extended to the richer representation language OWL. A similar summarization approach as the one used in **GRIN** was recently used by Dolby et al. [12] in their SHER project to summarize the semantics of OWL ontologies and to efficiently answer membership queries (which classes is a given resources an instance of?) over very large OWL ontologies.

GRIN and SHER can be used together for a wider class of queries over OWL ontologies. This is especially important given the high complexity of OWL queries – for instance, we know that the complexity of queries over OWL-Lite (the simpler of the OWL family of languages) is EXPTIME-complete. The complexity of conjunctive queries over OWL-DL (the intermediate level language) is still open.

Bibliography

[1] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very Large Data Bases (VLDB)* (2007), VLDB Endowment, pp. 411–422.

[2] Abello, J., and Kotidis, Y. Hierarchical graph indexing. In *Proceedings of the 2003 International Conference on Information and Knowledge Management (CIKM)* (2003), pp. 453–460.

[3] Baader, F., and Nutt, W. *Basic description logics*. Cambridge University Press, 2003, pp. 43–95.

[4] Baader, F., and Sattler, U. An overview of tableau algorithms for description logics. *Studia Logica 69* (2001), 5–40.

[5] Bayer, R. Binary b-trees for virtual memory. In *Proceedings of the ACM-SIGFIDET Workshop* (San Diego, California, 1971), pp. 219–235.

[6] Brickley, D., and Guha, R. V. RDF Vocabulary Description Language 1.0: RDF Schema. W3c recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[7] Broekstra, J., Kampman, A., and van Harmelen, F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the International Semantic Web Conference* (2002), pp. 54–68.

[8] Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. Named graphs, provenance and trust. In *Proceedings of the World Wide Web Conference* (2005), pp. 613–622.

[9] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*, 10 (2004), 1367–1372.

[10] D. Dubois, M., and Prade, H. Possibilistic uncertainty and fuzzy features in description logic: a preliminary discussion. In *Proceedings of the Workshop on Fuzzy Logic and the Semantic Web* (2005), pp. 5–7.

[11] Daniel J. Abadi, Samuel R. Madden, N. H. Column-stores vs. row-stores: How different are they really? In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Vancouver, Canada, 2008).

[12] Dolby, J., Fokoue, A., Kalyanpur, A., Kershenbaum, A., Schonberg, E., Srinivas, K., and Ma, L. Scalable semantic retrieval through summarization and refinement. In *AAAI* (2007), AAAI Press, pp. 299–304.

[13] Farquhar, A., Fikes, R., and Rice, J. The ontolingua server: a tool for collaborative ontology construction. *International Journal on Human-Computer Studies 46*, 6 (1997), 707–727.

[14] Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., and Patel-Schneider, P. F. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems 16*, 2 (2001), 38–45.

[15] Fitting, M. Bilattices and the semantics of logic programming. *Journal of Logic Programming 11*, 2 (1991), 91–116.

[16] Floyd, R. Algorithm 97: Shortest path. *Communications of the ACM 5*, 6 (1962), 345.

[17] Gergatsoulis, M., and Lilis, P. Multidimensional RDF. In *Proceedings of the International Conference on Ontologies, Databases, and Semantics (ODBASE)* (2005), vol. 3761, Springer, pp. 1188–1205.

[18] Goemans, M. X., and Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM 42*, 6 (1995), 1115–1145.

[19] Guarino, N., and Welty, C. Evaluating ontological decisions with ONTOCLEAN. *Communications of the ACM 45*, 2 (2002), 61–65.

[20] Guo, Y., Pan, Z., and Heflin, J. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics 3*, 2–3 (2005), 158–182.

[21] Gupta, A., and Mumick, I. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

[22] Gutiérrez, C., Hurtado, C. A., and Vaisman, A. A. Temporal RDF. In *Proceedings of the European Semantic Web Conference (ESWC)* (2005), pp. 93–107.

[23] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1984), pp. 47–57.

[24] Haase, P., Broekstra, J., Eberhart, A., and Volz, R. A comparison of rdf query languages. In *Proceedings of the International Semantic Web Conference (ISWC)* (NOV 2004).

[25] Harris, S., and Gibbins, N. Semantic Web storage with 3store. Technical Report. Department of Electronics and Computer Science, University of Southampton, Oct. 2003.

[26] Hayes, P. RDF Semantics. Tech. rep., W3C, 2004.

[27] Hlaoui, A., and Wang, S. A new algorithm for inexact graph matching. In *Proceedings of the International Conference on Pattern Recognition* (2002), pp. 180–183.

[28] Horrocks, I., Patel-Schneider, P., and van Harmelen, F. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics 1*, 1 (2003), 7–26.

[29] Hung, E., Deng, Y., and Subrahmanian, V. S. RDF Aggregate Queries and Views. In *Proceedings of the International Conference on Data Engineering (ICDE)* (2005), pp. 717–728.

[30] Jahama, S. *A general theory of semi-unification*. PhD thesis, Boston University, Boston, MA, USA, 1994.

[31] Kaufman, L., and Rousseeuw, P. Clustering by Means of Medoids. In *Statistical Data Analysis based on the L1 norm*. North Holland/Elsevier, 1987, pp. 405–416.

[32] Kifer, M., Lausen, G., and Wu, J. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42*, 4 (1995), 741–843.

[33] Kifer, M., and Subrahmanian, V. S. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming 12*, 4 (1992), 335–367.

[34] Lassila, O., and Swick, R. Resource description framework (RDF) model and syntax specification. See http://www.w3.org/ TR/WD-rdf-syntax.

[35] Leach, S. M., and Lu, J. J. Query processing in annotated logic programming: Theory and implementation. *Journal of Intelligent Information Systems 6*, 1 (1996), 33–58.

[36] Lenat, D. B., and Guha, R. V. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[37] Liu, B., and Hu, B. Path queries based RDF index. *Proceedings of the International Conference on Semantics, Knowledge and the Grid 1* (2005), 91.

[38] Maduko, A., Anyanwu, K., Sheth, A., and Schliekelman, P. Estimating the cardinality of RDF graph patterns. In *Proceedings of the International World Wide Web Conference (WWW)* (2007), pp. 1233–1234.

[39] Magkanaraki, A., Tannen, V., Christophides, V., Plexousakis, D., Scholl, M., and Tolle, R. Viewing the semantic web with RVL lenses. In *Proceedings of the International Semantic Web Conference (ISWC)* (2003).

[40] Manola, F., and Miller, E. RDF Primer, W3C Recommendation, 2004.

[41] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (1982), 258–282.

[42] Matono, A., Amagasa, T., Yoshikawa, M., and Uemura, S. An efficient pathway search using an indexing scheme for RDF. *Genome Informatics 14* (2003), 374–375.

[43] McGuinness, D. L., Fikes, R., Hendler, J., and Stein, L. A. DAML+OIL: An ontology language for the Semantic Web. *IEEE Intelligent Systems 17*, 5 (2002), 72–80.

[44] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems 8*, 4 (1990), 325–362.

[45] Pérez, J., Arenas, M., and Gutierrez, C. Semantics and Complexity of SPARQL. In *Proceedings of the International Semantic Web Conference (ISWC)* (2006), pp. 30–43.

[46] Preece, A., Flett, A., Sleeman, D., Curry, D., Meany, N., and Perry, P. Better knowledge management through knowledge engineering. *IEEE Intelligent Systems 16*, 1 (2001), 36–43.

[47] Pugliese, A., Udrea, O., and Subrahmanian, V. S. Scaling RDF with time. In *Proceedings of the International World Wide Web Conference (WWW)* (New York, NY, USA, 2008), ACM, pp. 605–614.

[48] Shadbolt, N., Motta, E., and Rouge, A. Constructing knowledge-based systems. *IEEE Software 10*, 6 (1993), 34–38.

[49] Sintek, M., and Kiesel, M. RDFBroker: A Signature–Based High–Performance RDF Store. In *Proceedings of the European Semantic Web Conference (ESWC)* (2006), pp. 363–377.

[50] Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the International World Wide Web Conference (WWW)* (New York, NY, USA, 2008), ACM, pp. 595–604.

[51] Straccia, U. Towards a fuzzy description logic for the semantic web. In *Proceedings of the Workshop on Fuzzy Logic and the Semantic Web* (2005), pp. 3–3.

[52] Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. Index structures and algorithms for querying distributed RDF repositories. In *Proceedings of the International World Wide Web Conference (WWW)* (New York, NY, USA, 2004), ACM, pp. 631–639.

[53] Su, X., and Ilebrekke, L. A comparative study of ontology languages and tools. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)* (2002), pp. 761–765.

[54] Theoharis, Y., Christophides, V., and Karvounarakis, G. Benchmarking Database Representations of RDF/S Stores. In *Proceedings of the International Semantic Web Conference (ISWC)* (2005), pp. 685–701.

[55] TPC-C Online Transaction Processing Benchmark. http://www.tpc.org/tpcc/, 2007.

[56] Trissl, S., and Leser, U. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (2007), pp. 845–856.

[57] Udrea, O., Pugliese, A., and Subrahmanian, V. S. GRIN: A Graph-based RDF Index. In *Proceedings of the Conference on Artificial Intelligence (AAAI)* (2007), AAAI Press, pp. 1465–1470.

[58] Udrea, O., Recupero, D. R., and Subrahmanian, V. S. Annotated RDF. In *Proceedings of the European Semantic Web Conference (ESWC)* (2006), pp. 487–501.

[59] Udrea, O., Subrahmanian, V. S., and Majkic, Z. Probabilistic RDF. In *Proceedings of the International Conference on Information Reuse and Integration (IRI)* (2006), pp. 172–177.

[60] Ullman, J. D. *Principles of database and knowledge-base systems, Vol. I.* Computer Science Press, Inc., 1988.

[61] uniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/, 2007.

[62] Volz, R., Oberle, D., and Studer, R. Towards views in the semantic web. In *Proceedings of the International Workshop on Databases, Documents, and Information Fusion* (2002).

[63] Wilkinson, K., Sayers, C., Kuno, H. A., and Reynolds, D. Efficient RDF Storage and Retrieval in Jena2. In *Semantic Web and Databases Workshop* (2003), pp. 131–150.

[64] Yan, X., Yu, P. S., and Han, J. Graph indexing: a frequent structure-based approach. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (2004), pp. 335–346.