Theses and Dissertations

2013

A Federated Query Answering System for Semantic Web Data

Yingjie Li
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation

A FEDERATED QUERY ANSWERING SYSTEM FOR SEMANTIC WEB DATA

by

Yingjie Li

A Dissertation

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

January 2013

This dissertation is accepted in partial

fulfillment of the requirements for the degree of

Doctor of Philosophy.

_____
(Date)

_____
Jeffrey D. Heflin (Advisor)
Associate Professor

_____
Hector Munoz-Avila
Associate Professor

_____
Brian D. Davison
Associate Professor

_____
Abir Qasem (Bridgewater College)
Assistant Professor

Acknowledgements

I would like to take this opportunity to express my thanks to those who helped me in one way or another on conducting research and the writing of this dissertation.

I want to express my heartfelt thanks to my advisor, Jeff Heflin. The work presented in this dissertation is not possible without his support and advice. I also want to thank Hector Munoz-Avila, Brian Davison and Abir Qasem for being on my defense committee and for their useful feedback and questions. I especially thank Abir Qasem for his precursor work on Ontology-based Information Integration (OBII) and Lehigh Customizable and Data-driven Benchmark (LCDBM), and his help as I expanded upon the foundation that he laid.

I gratefully acknowledge the help from many members who are present and former of the Semantic Web and Agent Technologies (SWAT) Laboratory, as well as from other institutions. I thank Yang Yu for his co-author work on LCDBM, Zhengxiang Pan for his help on Description Logic Databases (DLDB), and Xingjian Zhang and Dezhao Song for their useful dicussions and inspiration for this dissertation. Additionally, I thank Jie Bao for his helpful disscusions about my cyclic axiom handling algorithm and its correctness proof.

While working towards my degree, I was also an intern of Pitney Bowes, Samsung

Contents

List of Tables

List of Figures

Abstract

The Semantic Web extends the Web as a global information space from a Web of documents to a Web of data. Currently, there are billions of triples publicly available in the web data space of different domains. These data become more tightly interrelated as the number of links in the form of mappings is also growing. Typically, these data are heterogeneous, distributed and prone to dynamic changes. Although centralized knowledge bases and/or triple stores can be used to collect and query large volumes of heterogeneous Semantic Web data, they suffer from many disadvantages. First, they will become stale unless they are frequently reloaded with fresh data. Second, they can require significant disk space, especially for triple stores that use multiple triple indices to optimize queries. Finally, there may be legal or policy issues that prevent one from copying data or storing it in a centralized place. Therefore, this dissertation explores ways to address the above challenges from the perspective of building a federated query answering system for semantic web data. The system can quickly and effectively find relevant data sources and further answer queries. It employs an automated mechanism for creating an inverted index used in determining source relevance. Then, a hybrid approach to answering queries that involves ideas from information retrieval, information integration and knowledge bases is applied.

First, the dissertation formally defines a group of concepts to describe a federated query answering problem for the Semantic Web. Guided by the theoretical framework, it then presents and implements an efficient, IR-inspired inverted index named term index to integrate semantic web data sources and determine source relevance. Based on this term index, four query answering algorithms are proposed.

1

Each of them is optimized in order to overcome the drawbacks of the previous ones. The non-structure algorithm takes a set of query subgoals as inputs and dynamically loads all relevant sources into a reasoner to solve the original query. The flat-structure algorithm optimizes source selection and dynamically answers queries by reformulating the original conjunctive query into a list of conjunctive query rewritings. The tree-structure algorithm answers queries by reformulating the original conjunctive query into an AND/OR tree, generating a query execution plan on the fly and dynamically executing a bottom-up greedy source collection. The dynamic cyclic axiom handling algorithm is to make the tree-structure algorithm still return complete query answers when cyclic axioms are considered. Experiments conducted using synthetic data and real world data and the theoretical correctness proof of algorithms have demonstrated that a system based on these algorithms can effectively and correctly scale to dynamic, web-scale knowledge bases.

Chapter 1

Introduction

1.1 Motivation

The World Wide Web has radically altered the way we share knowledge by lowering the barrier to publishing and accessing documents as part of a global information space. Hypertext links allow users to traverse this information space using Web browsers, while search engines index the documents and analyse the structure of links between them to infer potential relevance to users' search queries [11]. This functionality has been enabled by the generic, open and extensible nature of the Web [34], which is also seen as a key feature in the Web's unconstrained growth.

The inarguable success of the World Wide Web, particularly search engines such as Google, may lead one to believe that the Web has reached its full potential as a global knowledge repository. However, with the amount of data available on the Web increasing rapidly in recent years, some of our information needs cannot be met by even today's state of the art web technologies. In such cases, we still need

significant human intervention to find what we need. One example is that if a user is looking for some information about the Chairman of RPI's Computer Science Department, a reasonable action may be for him to type in "RPI Computer Science Chair" in a search engine's query box. However, the search engines will not list the most relevant documents in the first few result records. The reason is that at RPI's websites the department chairs are more commonly referred to as department heads. In retrieving documents in response to a query, search engines fail to recognize similar concepts when they are expressed using different terminologies. In the example we just discussed, the search engine was unable to recognize that the words "Department Chair" and "Department Head" were similar concepts. Another example is: "find all journals that academic descendants of Marvin Minsky have published in". In this example, the search engine fails to find answers because no single data source can completely satisfy the example information need. Yet, with the integration of the data sources DBLP [1], Citeseer [2] and AI Genealogy Project (AIGP) [3], an answer in principle can be obtained: DBLP and Citeseer contain publication metadata such as authors along with their affiliations, and information about academic descendants of Marvin Minsky can be found in AIGP. More sophisticated queries like "a list of academic papers written by Marvin Minsky's advisees who live in Washington DC" or "a list of all 4 year colleges with Computer Science Departments within 170 miles radius of the zip code 18015" are also well beyond the capacities of present day search technology. However, in all these cases, we intuitively know that the Web does contain the information we are seeking; we just do not have a fully automated

---

[1]http://dbis.uni-trier.de/DBL-Browser/
[2]http://citeseerx.ist.psu.edu/index
[3]http://aigp.eecs.umich.edu/

way of getting them.

The reason of the inability of existing search technology to meet our information needs, as exemplified by the queries above, is that contemporary search technology is based on keyword matching, not designed for answering structured queries. This leads to some limiting assumptions for today's web. One assumption is that in the Web, a document is deemed to be relevant to a user's query if its content is "similar" to the text entered by users. The basic determinant of this similarity is the one between textual representation of the words in the user's query and the documents in the web. However, as shown in the query about RPI's Computer Science Chair, the success of the search depends on how correctly we have expressed our query in terms of the documents available. This is the reason why the contemporary search technology cannot find "Department Head" when the query is "Department Chair".

Another limiting assumption is that the Web is viewed as a collection of documents that are connected via hypertext links. Once a set of relevant documents is identified using a matching criteria, it is left up to the user to inspect the document and process the information that is contained in the documents. As a consequence, when a user's information needs span multiple documents, he has to manually process each document from each result, and merge together the relevant information by himself. This is the reason why contemporary search technology cannot obtain integrated information from multiple documents, such as Minsky's advisees and 4 year colleges mentioned early on.

To help the Web reach its true potential, the Semantic Web has suggested a way of extending the existing web with strucutre and providing a mechanism to specify formal semantics that are machine-readable and shareable. In this way, the web

information can be readily interpreted by machines, so machines can perform more of the tedious work involved in finding, combining and acting upon information on the web without human intervention. They do so via the use of ontologies. An ontology is a formal logic-based description of a vocabulary that allows one to talk about a domain of discourse. The vocabulary is articulated using definitions and relationships among the defined concepts. As ontologies use formal logic they can describe a domain unambiguously as long as the information is interpreted using the same logic. Further, the use of logic makes it possible to use software to "infer" implicit information in addition to what is explicitly stated.

One of the most exciting things about the Semantic Web is to drive the evolution of the Web as a global information space from a Web of documents to a Web of data, where not only documents but data is also linked. Underpinning this evolution is a set of best practices for publishing and connecting structured data on the Web known as Linked Data [4]. These data are often independently generated, distributed in many locations, heterogeneous from diverse domains such as biology, government, geography, etc. or in different representation formats such as databases, XML files, spreadsheets and others, and in large volume as well. In today's Semantic Web, there are billions of semantic data triples publicly available in different domains such as people, companies, books, scientific publications, films, music, television and radio programmes, genes, proteins, drugs and clinical trials, online communities, statistical and scientific data, and reviews. These data become more tightly interrelated as the number of links in the form of mappings is also growing. The process of interlinking open data sources is actively pursued within

---

[4]http://linkeddata.org/

Linking Open Data (LOD) [33], which is a project that aims to extend the Web with a data commons by publishing various open datasets as RDF on the Web and by setting RDF links between data items from different data sources. Typically, Semantic Web data exhibits the following features:

- Heterogeneity: data sources cover different, possibly overlapping domains. Data contained in different sources might be redundant, complementary or conflicting. Also, the time required to obtain the same amount of data might vary greatly due to network latency.

- Smallness: data sources in RDF are potentially many small files, around 50 triples according to web data statistics using Sindice [5]. This can be shown by the facts that many large LOD RDF data sources provide an interface for their individual data objects: e.g. DBpedia has a separate RDF page for each entry, GeoNames has a separate page for each place, DBLP Berlin has a separate page for each author and publication, etc. These pages typically have a small number of RDF triples.

- Dynamicity: data sources are added and removed and sources' content changes rapidly over time. Due to this dynamic, it is no longer safe to assume that information about all sources can be obtained. In particular, sources might be a priori unknown and can only be discovered at run-time.

- Scalability: The amount of data on the Web is ever increasing. The LOD project alone already contains roughly 31 billion RDF triples in more than 20

---

[5]http://www.sindice.com/

domains. Clearly, efficient query answering that can scale to this amount of data is essential for the data search on the web.

The development of a data Web opens a new way for addressing complex information needs for Web search - query answering instead of document relevance. As exemplified in the second paragraph, contemporary information retrieval (IR) services such as Google are excellent at finding the most relevant documents for specialized terms like the names of people and organizations, but are unable to provide direct answers to specific queries, especially if the answer to the query cannot be found in any single source. Therefore, the problem of query answering over the Semantic Web data is becoming more and more important.

Traditionally, query answering has been conducted on relatively small and closed corpora. Following the idea of LOD, the Web has extended data sources to include information freely available from the Web, which presents an enormous potential for integrated querying over multiple distributed data sources. The general procedure to work with multiple, distributed linked data sources is to load the desired data (for instance, in the form of dumps) into a local, centralized warehouse and process queries in a centralized way against the merged data set. One representative solution is the Data Warehouse, which is a database used for reporting and data analysis [64]. The data stored in the data warehouse are uploaded from the operational systems, cleansed, transformed, and placed into the data warehouse or data mart according to a schema, such as the star schema. The data marts store subsets of data from a warehouse. The star schema is a logical arrangement of tables in a multidimensional database. The goal of data warehouse is to integrate applications at the data level and create a centralized and unified view of enterprise

data holdings. However, accounting for the decentralized structure of the Semantic Web, the centralized approach may not always be practically feasible or desired. For example, in some cases a complete dump of the data sources may not be available, instead the data source may only be accessible via a query endpoint. In the case of frequently changing data sources, the synchronization with the centralized store becomes a problem. Typically, the centralized approach suffers from the following disadvantages:

- First, the systems will become stale unless they are frequently reloaded with fresh data, which can be especially expensive if the knowledge-bases rely on forward-chaining that starts with the available data and uses inference rules to extract more data until a goal is reached.

- Second, the systems can require significant disk space, especially for triple stores that use multiple triple indices to optimize queries. For example, Hexastore [80] replicates each triple six times.

- Finally, there may be legal or policy issues that prevent one from copying data or storing it in a centralized place.

In order to solve the above problems, one can observe a recent paradigm shift towards federated approaches over the distributed data sources. In this approach, a query against a federation of data sources is split into queries that can be answered by the individual data sources and the results are merged by the federator. From the user perspective this means that data of multiple heterogeneous sources can be queried transparently as if residing in the same database. Sometimes this approach is therefore referred to as virtual integration. Approaches to federated query processing

over linked data are still in their infancy. Some first proposals exist, but none of them have been practically used on a large scale. In this dissertation, I have designed and developed a federated Semantic Web query answering solution that recognizes the reality, that despite our best efforts, heterogeneity, scalability and dynamicity issues are always inherent in any (Semantic) Web query answering system. Therefore, in my research, I looked for a practical solution that performs reasonably well, despite these issues.

1.2 Contributions

In this dissertation, I will explore ways to build an ontology-based information integration system that can quickly and effectively find relevant data sources and further answer queries on the fly. This system will receive queries from users, determine which sources are relevant to the query, retrieve these sources in real-time, and use them to answer the query. In this system, I designed an automated mechanism for creating the index to determine source relevance. Then, I furthermore proposed a hybrid approach that involves ideas from information retrieval, information integration and knowledge base systems to answer queries. Specifically, my dissertation makes the following technical contributions:

- I have formally defined a group of concepts to describe a federated query answering problem for the Semantic Web. In answering a Semantic Web query, these concepts can be used to reason only with the subset of a knowledge base that is necessary to answer the given query. As the size of a knowledge base significantly impacts reasoning time, the source selection framework provides

efficiency gains by enabling a novel, query based, pruning of the knowledge base.

- Guided by my theoretical framework, I first designed and implemented an efficient, IR-inspired inverted index named term index to integrate semantic web data sources and determine source relevance. Based on this term index, I have designed and implemented a non-structure query answering algorithm, which takes a set of query subgoals as inputs and dynamically loads all relevant sources into a reasoner to solve the original query. As demonstrated by my empirical evaluations, this algorithm has gained better source selectivity and faster query response time compared to the precursor work done by Qasem et al.[62].

- In order to overcome the drawbacks of the non-structure algorithm, I successively proposed two query optimization algorithms: the flat-structure algorithm and the tree-structure algorithm. The flat-structure algorithm optimizes source selection and answers queries by reformulating the original conjunctive query into a list of conjunctive query rewritings and dynamically generate a query execution plan by selecting relevant sources for each query rewrite. The tree-structure algorithm answers queries by reformulating the original conjunctive query into an AND/OR tree using the Peer Data Management System (PDMS) algorithm [28] and dynamically plan the query execution through selecting relevant sources over the tree. Furthermore, in order to make the tree-structure algorithm still return complete query answers when cyclic axioms are considered, I also proposed a dynamic cyclic axiom handling algorithm

by improving the tree-structure algorithm. As demonstrated by my empirical evaluations, all my proposed algortihms perform better than the non-structure algorithm over average query response time, source selectivity, index accesses and scalability.

- I have theoretically proved the soundness and completeness of the non-structure algorithm, the flat-structure algorithm, the tree-structure algorithm and the cyclic axiom handling algorithm.

In order to appropriately scope my dissertation, I will not consider automated ontology alignment algorithms [71], although the work described herein can benefit from any advances in the area. I will also not consider issues of trust and provenance [3], although these will clearly be important in the long term. Finally, I will not address user interface issues [37] [49], assuming instead that front-ends can translate the user input into a common query language.

1.3 Thesis Overview

The dissertation contains the following chapters (in additional to this introduction chapter):

- Chapter 2 provides the readers with an overview of various technologies and research areas that I have explored, used and benefited from in this disserta-tion. In this chapter, I discuss various technologies that are the building blocks of the Semantic Web. In addition, I also survey work from related research

areas such as Information Retrieval, Query Optimization and Information Integration, etc. and describe how the work summarized is similar to or different from the work I have done in my dissertation.

- Chapter 3 lays the theoretical foundation of my work, describes my problem decomposition and presents an IR-inspired indexing scheme - term index to index Semantic Web data. Furthermore, based on the term index, it defines two basic functions to explain how to use the term index to select potentially relevant sources for the query answering. This work has been published in WI 2010 [45].

- Chapter 4 discusses the query answering algorithms for my system. This chapter describes and compares three algorithms: the non-structure algorithm, the flat-structure algorithm and the tree-structure algorithm. For each of them, the correctness proof is given. These algorithms have been published in WI 2010 [45], CIKM 2010 [42] and ISWC 2010 [43].

- Chapter 5 examines the cyclic axiom handling by improving the tree-structure algorithm. This chapter first gives an formal definition of the cyclic axiom, then explains why the original tree-strucuture algorithm is incomplete in presence of cyclic axioms and describes how the original tree-structure algorithm can be improved to dynamically handle cyclic axioms. After that, the correctness proof of the proposed algorithm is given. This work has been published in SSWS 2011 [44].

- Chapter 6 describes how I empirically evaluated my algorithms from two aspects: the heterogeneity evaluation using multiple ontologies and the large

scale evaluation. I have developed a multi-ontology benchmark - Lehigh Customizable Data-driven Benchmark (LCDBM) which I used to do the heterogeneity evaluation of my algorithms. The benchmark work have been published in IWEST 2010 [46] and ORE 2012 [47].

- Chapter 7 summarizes the work, gives open problems of the work and sets directions for future work.

Chapter 2

Background and Related Work

This chapter serves two purposes. First it provides some background information about some specific technologies that I have either used or benefited from in this dissertation. This is not a detailed tutorial of these subject matters. However, wherever appropriate, I point the interested readers to relevant references. In Section 2.1, I provide a brief introduction to the Semantic Web languages. Most of the materials in this section are not necessarily a prerequisite to understanding the details of my thesis. They are mainly presented to give the reader a bird's-eye view of the myriads of Semantic Web languages. In Section 2.2, I survey the rich area of centralized Semantic Web knowledge system. Since a key element of this dissertation is a new scheme for indexing distributed semantic data, I mainly focus on the literature on indexing schemes used among them. In Section 2.3, I describe the related work on query optimization, especially those in Database and Semantic Web areas since my later proposed query optimization algorithms got inpired by some of them. Finally, in Section 2.4, I research the rich area of information integration

from a Semantic Web perspective. I mainly discuss two categories of related work: the traditional information integration taking Global-As-View (GAV) and Local-As-View (LAV) as representatives and the ontology related information integration. In each of this four sections, I mention how the work summarized is similar to or different from the work I have done in my dissertation.

2.1 Semantic Web Languages

The Semantic Web is promising to be the next generation of the Web. Largely based on HTML, the current Web provides the information that is only human understandable rather than machine understandable. The goal of the Semantic Web is to provide a common framework that allows data and knowledge to be shared and reused across applications, enterprises and communities by making the web documents' meaning explicit. To do this, Semantic Web researchers have developed OWL and OWL 2, which are ontology languages that extend Resource Description Framework (RDF) [40] and RDF Schema [10]. In this section, I first describe RDF, RDF query language - SPARQL and RDF Schema. Then, I briefly introduce OWL, OWL 2 and their logical basis - Description Logics.

2.1.1 RDF

RDF is a data model that is based on the idea of making statements about "resources" in the form of subject-predicate-object expressions. These statements are often referred to as triples. In RDF, a resource can be anything that is uniquely identifiable via a Uniform Resource Identifier (URI). Note: URIs are more general

than the well-known web resource identifier URLs. Specifically, there is no require-
ment that a URI needs to point to a resource that is accessed via the Internet. In
an RDF triple, the subject denotes the resource that we want to make a statement
about. The predicate denotes traits or aspects of that resource and expresses a rela-
tionship between the subject and the object. The object can be a resource identified
by a URI, however it can alternatively be a literal value like a string or a number.

RDF is closely related to Semantic Networks. The Semantic Networks is a well-
known and very flexible knowledge representation mechanism. Similar to Semantic
Networks, RDF statements can be expressed in a graph with labeled nodes con-
nected by directed and labeled edges. Essentially, the subject of a RDF statement
is the source node of the edge, the object is the target node of the edge and the edge
is the predicate relating the subject and the object. Consider, for example, that we
want to say "Jeff Heflin" advises "Yingjie Li". This statement will be represented
in an RDF graph with a source that denotes "Jeff Heflin" spreading out two edges:
one directed edge from source to destination that denotes the "rdf:type" relation-
ship and a destination that denotes a class "Professor" and the other directed edge
from source to destination that denotes the "advises" relationship and a destina-
tion that denotes "Yingjie Li". In RDF we need URIs (or URLs) to refer to the
namespace "ex" of our example, the entity "Jeff Heflin", the "advises" relationship,
the "rdf:type" relationship, the class "Professor" and the entity "Yingjie Li". The
following is one version of the RDF triples represented in XML syntax.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```
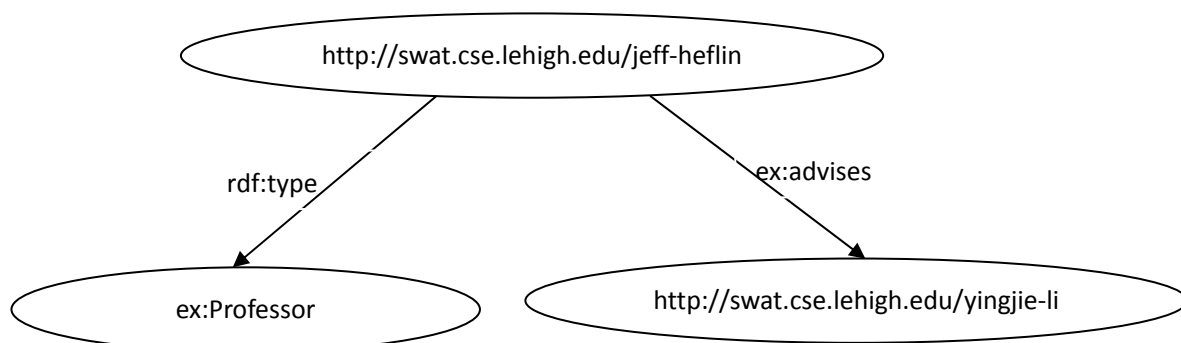
Figure 2.1: RDF graph

```
    xmlns:ex="http://swat.cse.lehigh.edu.com/"

>

    <rdf:Description rdf:about="http://swat.cse.lehigh.edu/jeff-heflin">

        <rdf:type rdf:resource="http://swat.cse.lehigh.edu/Professor"/>

        <ex:advises rdf:resource="http://swat.cse.lehigh.edu/yingjie-li"/>

    </rdf:Description>

</rdf:RDF>
```

Corresponding to the triple representation, Figure 2.1 is the graphical representation.

2.1.2 SPARQL

As stated in Section 2.1.1, RDF is a directed, labeled graph data format for representing information in the Web. The Simple Protocol and RDF Query Language (SPARQL) is a SQL-like language for querying RDF data. SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns.

A triple pattern is like an RDF triple, but with the option of a variable in place of RDF terms (i.e., URIs, URLs, literals or blank nodes) in the subject, predicate or object positions. A set of triple patterns written as a sequence of triple patterns with conjunction or disjunction relations composes a Basic Graph Pattern (BGP). BGP allows applications to make queries where the entire query pattern must match for there to be a solution. Since regular, complete structures cannot be assumed in all RDF graphs, the optional patterns provides the facility if the optional part does not match, it creates no bindings but does not eliminate the solution. An example of a SELECT query is as follows:

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

The first line defines a namespace prefix, the last two lines use the prefix to express a RDF graph pattern to be matched. Identifiers beginning with question mark ? identify variables. In this query, we are looking for resource $?x$ participating in triples with predicates $foaf{:}name$ and $foaf{:}mbox$ and want the subjects of these triples.

In addition to specifying graph to be matched, constraints can be added for values using FILTER construct. An example of string value restriction is $FILTER$ $regex(?mbox,$ "$company$") that specifies regular expression query. An example of number value restriction is $FILTER(?price < 20)$ that specifies that $?price$ must be less than 20. A few special operators are defined for the FILTER construct. They

include isLiteral for testing whether a variable is literal, bound to test whether variable was bound and others.

The matching part of the query may include OPTIONAL patterns. If the triple to be matched is optional, it is evaluated when it is present, but the matching does not fail when it is not present. Optional sections may be nested. It is possible to make UNION of multiple matching graphs - if any of the graphs matches, the match will be returned as a result. The FROM part of the query is optional and may specify the RDF dataset on which query is performed.

The sequence of result may be modified using the following keywords with the meaning similar to SQL:

- ORDER BY: ordering by variable value.

- DISTINCT: unique results only.

- OFFSET: offset from which to show results.

- LIMIT: the maximum number of results.

There are four query result forms. In addition to the possibility of getting the list of values found it is also possible to construct RDF graph or to confirm whether a match was found or not.

- SELECT: returns the list of values of variables bound in a query pattern.

- CONSTRUCT: returns an RDF graph constructed by substituting variables in the query pattern.

- DESCRIBE: returns an RDF graph describing the resources that were found.

- ASK: returns a boolean value indicating whether the query pattern matches or not.

The CONSTRUCT form specifies a graph to be returned with variables to be substituted from the query pattern, such as in the following example that will return graph saying that Alice knows last two people when ordered by alphabet from the given URI (the result in the RDF graph is not ordered, it is a graph and so the order of triples is not important).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { <http://example.org/person#Alice> foaf:knows ?x }
FROM <http://example.org/foaf/people>
WHERE { ?x foaf:name ?name }
ORDER BY desc(?name)
LIMIT 2
```

The DESCRIBE form will return information about matched resources in a form of an RDF graph. The exact form of this information is not standardized yet, but usually a blank node closure like for example Concise Bounded Description (CBD) is expected. In short, all the triples that have the matched resource in the object are returned; when a blank node is in the subject, then the triples in which this node participates as object are recursively added as well.

The ASK form is intended for asking yes/no questions about matching - no information about matched variables is returned, the result is only indicating whether matching exists or not.
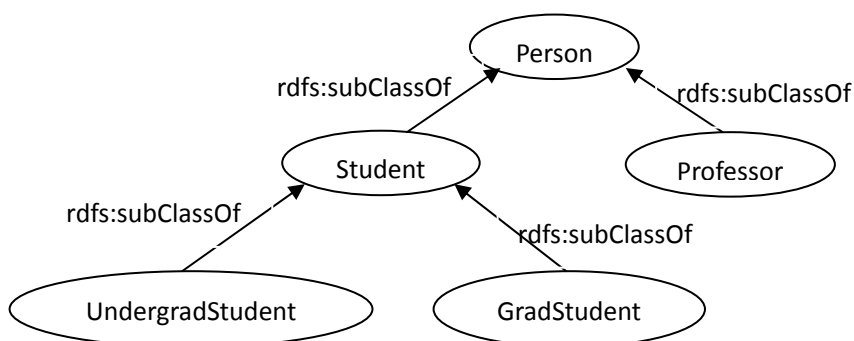
Figure 2.2: RDFS graph

2.1.3 RDF Schema

As shown in section 2.1.1, RDF is a simple data model and as such it does not have any significant semantics. In order to address this shortcoming, on top of RDF, RDF Schema (RDFS) as a W3C standard provides additional modeling primitives to define classes, subclass relationships between classes, properties, subproperty relationships between properties, and restrictions on property domains and ranges, and so on. In this way, RDFS provides simple functions to build vocabularies for RDF statements and thus for associating metadata to each other.

Figure 2.2 depicts how a set of RDFS subClassOf statements can be used to express the hierarchy of a set of concepts. As with RDF, RDFS statements can be also expressed in XML. The statements below define the classes and their hierarchy shown in Figure 2.2, and in addition, specify the domain class and range class for the property advises.

```
<rdfs:Class rdf:ID = "Person"/>
<rdfs:Class rdf:ID = "Student"/>
```

```
    <rdfs:subClassOf rdf:resource = "#Person"/>

</rdfs:Class>

<rdfs:Class rdf:ID = "Professor">

    <rdfs:subClassOf rdf:resource = "#Person"/>

</rdfs:Class>

<rdfs:Class rdf:ID = "UndergradStudent">

    <rdfs:subClassOf rdf:resource = "#Student"/>

</rdfs:Class>

<rdfs:Class rdf:ID = "GradStudent">

    <rdfs:subClassOf rdf:resource = "#Student"/>

</rdfs:Class>

<rdfs:Property rdf:ID = "advises">

    <rdfs:domain rdf:resource = "#Professor"/>

    <rdfs:range rdf:resource = "#Student"/>

</rdfs:Property>
```

The limited semantics provided by RDFS limited primitives however did not prove sufficient to handle real world modeling needs. For instance, they cannot model class disjointness and intersection relationships, property symmety, cardinality, etc. This is one of the reasons for the development of more expressive languages such as OWL and OWL 2 that will be introduced in the next section.

2.1.4 OWL and OWL 2

RDFS only has limited expressiveness. More powerful description languages are proposed for establishing vocabularies for describing Semantic Web data. These

vocabularies usually known as ontologies, define terms in a domain and their relationships, both within the same ontology and cross ontologies. According to the official ontology description [59], an ontology consists of classes which denote a set of instances, properties which denote binary relationships between instances and axioms that relate classes, properties and instances. Further, as ontologies are web documents, they also have an unique document identifier (a URL).

Among those ontology languages, OWL (Web Ontology Language) has become the W3C recommendation. OWL is based on RDF and RDFS and adds more language constructs for describing classes and properties, including more relations between classes (e.g. disjointness), cardinality (e.g. exactly one), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

```
<owl:Class rdf:ID = "Chair">

    <owl:intersectionOf rdf:parseType = "Collection">

        <owl:Class rdf:about = "#Person"/>

        <owl:Restriction>

            <owl:onProperty rdf:resource =  "#headOf"/>

            <owl:someValuesFrom rdf:resource = "#Department"/>

        </owl:Restriction>

    </owl:intersectionOf>

</owl:Class>
```

The above OWL statements defines the class "Chair" as the intersection of the class "Person" and an anonymous class of instances for which at least one value of the property "headOf" is an instance of the class "Department". Simply speaking,

this expresses the constraint that a chair has to be a person who is the head of a department.

In order to make OWL more expressive, the W3C Recommendation for OWL 2 was published in 2009. This document essentially clarifies some ambiguities in OWL and adds some useful features such as property composition, which is needed to define axioms for statements such as "an uncle is the brother of a parent.".

```
<rdf:Description rdf:about = "hasUncle">

   <owl:propertyChainAxiom rdf:parseType = "Collection">

      <owl:ObjectProperty rdf:about = "hasParent"/>

      <owl:ObjectProperty rdf:about = "hasBrother"/>

   </owl:propertyChainAxiom>

</rdf:Description>
```

The above OWL 2 statements defines the property "hasUncle" to be the composition of two properties: "hasParent" and "hasBrother". Technically, this means that we want "hasUncle" to connect all individuals that are linked by a chain of two properties of "hasParent" and "hasBrother".

In OWL 2, there are three sublanguages. OWL 2 EL is a fragment that has polynomial time reasoning complexity and is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. OWL 2 QL is designed to enable easier access and query to data stored in databases and is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. OWL 2 RL is a rule subset of OWL 2 and aimed at applications that require scalable reasoning without sacrificing too much expressive power.

2.1.5 Description Logics

OWL and OWL 2 are both based on Description Logics (DL). Generally, DLs are a family of logics that are decidable fragments of first-order predicate logic. First-order logic is a formal system and with a specified domain of discourse over which the quantified variables range, one or more interpreted predicate letters, and proper axioms involving the interpreted predicate letters are defined [67]. DLs focus on describing classes and roles, and have a set-theoretic semantics. Different DLs include different subsets of logical operators.

DLs can be used to represent the knowledge of an application domain in a structured and formally well understood way. In DLs, a knowledge base has two components, the TBox and the ABox. The TBox introduces the vocabulary (terminology) of an application domain, while the ABox contains facts (assertions) about named individuals in terms of this vocabulary. The vocabulary consists of concepts, which denote sets of individuals, and roles, which denote binary relationships between individuals.

The semantics of DL is described using the set theoretic approach. In describing the semantics one considers interpretations I that consist of a nonempty set Δ^I (domain of interpretation) and an interpretation function that assigns to every atomic concept A a set $A^I \subseteq \Delta^I$ and to every role R a binary relation $R^I \subseteq \Delta^I \times \Delta^I$. Table 2.1 lists various OWL and OWL 2 constructors, their DL syntax, and the semantic conditions. We say that an interpretation is a model of a statement iff the semantic conditions specified in table 2.1 for that statement hold in the interpretation. An interpretation is a model of a knowledge base or ontology iff it is a model of every statement/axiom in the knowledge base/ontology.

Constructor/ Axiom name	DL	Semantics	Constructor/ Axiom name	DL
owl:Thing	$\top$	Δ^I	owl:sameAs	$a \equiv b$
owl:Nothing	$\bot$	$\emptyset$	owl:differentFrom	$a \neq b$
rdfs:subClassOf	$C_1 \sqsubseteq C_2$	$C_1^I \subseteq C_2^I$	rdfs:domain	$\top \sqsubseteq \forall P^-.C$
rdfs:subPropertyOf	$P_1 \sqsubseteq P_2$	$P_1^I \subseteq P_2^I$	rdfs:range	$\top \sqsubseteq \forall P.C$
owl:intersectionOf	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^I = C_1^I \cap C_2^I$	owl:equivalentClass	$C_1 = C_2$
owl:unionOf	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^I = C_1^I \cup C_2^I$	owl:equivalentProperty	$P_1 = P_2$
owl:complementOf	$\neg C$	$(\neg C)^I = \Delta^I \setminus C^I$	owl:disjointWith	$C_1 \equiv \neg C_2$
owl:TransitiveProperty	$P \sqsubseteq P_+$	$P^I = (P^I)^+$	owl:SymmetricProperty	$P \sqsubseteq P^-$
owl:inverseOf	$P \sqsubseteq S^-$	$(P^I)^-$	owl:FunctionalProperty	$\top \sqsubseteq \leq 1P$
owl:allValuesFrom	$\forall P.C$	$(\forall P.C)^I = \{x \vert \forall y, \langle x,y \rangle \in P^I \to y \in C^I \}$	owl:InverseFunctional Property	$\top \sqsubseteq \leq 1P^-$
owl:dataAllValuesFrom	$\forall DP.DR$	$(\forall DP.DR)^I = \{x \vert \forall y, \langle x,y \rangle \in DP^I \to y \in DR \}$	owl:cardinality owl:dataCardinality	$\leq nP \sqcap \geq nP$ $\leq nDP \sqcap \geq nDP$
owl:someValuesFrom	$\exists P.C$	$(\exists P.C)^I = \{x \vert \exists y, \langle x,y \rangle \in P^I \text{ and } y \in C^I \}$	owl:disjointUnionOf (C, C_1,...,C_n)	$C^- = C_1^- \sqcup ... \sqcup C_n^-$ and $C_i^- \sqcap C_j^- = \emptyset$
owl:dataSomeValuesFrom	$\exists DP.DR$	$(\exists DP.DR)^I = \{x \vert \exists y, \langle x,y \rangle \in DP^I \text{ and } y \in DR \}$	owl:ReflexiveProperty	for each $1 \leq i, j \leq n$ such that $i \neq j$ $\forall x : x \in \Delta^I \to (x,x) \in P$
owl:minCardinality	$\geq nP$	$(\geq nP)^I = \{x \vert \neq (\{y, \langle x,y \rangle \in P^I \}) \geq n \}$	owl:IrreflexiveProperty owl:AsymmetricProperty	$\forall x : x \in \Delta^I \to (x,x) \notin P$ $\forall x, y : (x, y) \in R \to (y, x) \notin P$
owl:dataMinCardinality	$\geq nDP$	$(\geq nDP)^I = \{x \vert \neq (\{y, \langle x,y \rangle \in DP^I \}) \geq n \}$	propertyDisjointWith (P_1,...,P_n)	$P_i \sqcap P_j = \emptyset$ for each $1 \leq i, j \leq n$ such that $i \neq j$
owl:maxCardinality	$\leq nP$	$(\leq nP)^I = \{x \vert \neq (\{y, \langle x,y \rangle \in P^I \}) \leq n \}$	propertyChainAxiom owl:hasSelf	$P_1 \circ ... \circ P_n \sqsubseteq P$ $\{x \vert (x,x) \in P\}$
owl:dataMaxCardinality	$\leq nDP$	$(\leq nDP)^I = \{x \vert \neq (\{y, \langle x,y \rangle \in DP^I \}) \leq n \}$		
owl:oneOf	$\{o_1,....,o_n\}$	$\{o_1,....,o_n\}^I = \{o_1^I,....,o_n^I\}$		

Table 2.1: OWL and OWL 2 syntax and semantics

The statements in the TBox and in the ABox of most DL languages can be represented by formulas in first-order logic (FOL). From a FOL point of view a concept can be viewed as a unary predicate whereas a role can be viewed as a binary predicate. In addition to atomic concepts and roles, all description logic languages allow their users to build complex descriptions of concepts and roles. Note: the TBox can be used to assign names to these complex descriptions. Each description logic language is distinguished by the type of complex descriptions it allows the user to use.

A system built on a certain DL not only stores terminologies and assertions, but also offers services that reason about them. Typical reasoning tasks for a TBox are

to determine whether a description is non-contradictory (satisfiablity) or whether one is subsumed by the other (subsumption). A typical reasoning task for an ABox is consistency checking. DLs use set operators and standard logic symbols for their description constructors. Following is an example of a DL knowledge base.

TBox

Computer $\sqsubseteq$ Electronics

USComputer $\equiv$ Computer $\sqcap$ $\exists$madeIn.{US}

USComputerSoldOnline $\equiv$ USComputer $\sqcap$ $\exists$soldBy.OnlineStore

NonUSComputer $\equiv$ Computer $\sqcap$ $\neg$USComputer

ABox

USComputer(DELL-XPS), soldBy(DELL-XPS,AMAZON),

madeIn(TOSHIBA-A136,JAPAN), Computer(TOSHIBA-A136),

OnlineStore(AMAZON)

In the example above, Electronics is a simple concept description and descriptions like Computers, USComputers etc. are complex concept descriptions as they are defined in terms of other concepts. Note: once they are defined, a complex DL concept can be used to build even more complex descriptions. Given the above knowledge base a DL reasoner can perform reasoning tasks like finding all the subclasses of Electronics. Note although there is only one explicit subclass of Electronics, the reasoner will be able to infer all other implicit subclasses from the descriptions available in the knowledge base.

2.2 Information Retrieval for Semantic Web

Given that a key element of this dissertation is a new IR-inspired scheme for indexing distributed RDF data, it is useful to survey the application of IR for Semantic Web and particularly the indexing schemes that were designed for RDF data storage. Based on my survey, the major disadvantage of the former is that they primarily focus on document search rather than query answering. The major disadvantages of the latter are that they rely on centralized knowledge bases and that the indexes (or replication) are quite expensive in terms of space. My goal is to leave the original data at its source, to have compact local representations that help us locate this data and to directly answer queries. In this section, I will first introduce some work about the application of IR for Semantic Web. Then, I will summarize the related work on indexing schemes for RDF data storage.

I categorize the related work on IR for Semantic Web into two categories: search for documents or ontological elements and search for ontologies. In the first category, Kandogan et al. developed a semantic search engine - Avatar, which combines the traditional text search engine with use of ontology annotations [36]. Avatar has two main functions: a) extraction and representation - by means of UIMA framework, which is a workflow consisting of a chain of annotators extracted from documents and stored in the annotation store; b) interpretation - process of automatically transforming a keyword search to several precise searches. Avatar consists of two main parts: semantic optimizer and user interaction engine. When a query is entered into the former, it will output a list of ranked interpretations for the query; then the top ranked interpretations are passed to the latter, which will display the interpretations and the retrieved documents from the interpretations.

Bhagwat et al. proposed a semantic-based file system search engine- Eureka, which uses an inference model to build the links between files and a File Rank metric that is to rank the files according to their semantic importance [8]. Eureka has two main parts: a) crawler which extracts file from file system and generates two kinds of indices: keywords' indices that record the keywords from crawled files, and rank index that records the File Rank metrics of the files; b) when search terms are entered, the query engine will match the search terms with keywords' indices, and determine the matched file sets and their ranking order by an information retrieval-based metrics and File Rank metrics.

Jiang et al. proposed a full-Text Search Engine for the Semantic Web called OntoSearch [35]. It was developed to allow simple keyword based query of ontologies by passing keywords to Google, packaged in such a way as to only return ontological data in RDF. In this approach, search starts with a term-based query which yields to a set of documents, from these documents semantic metadata is extracted and used for a spreading activation search in an ontology. The extended set of concepts is used to rank the search results of the term based search. Ranking is done using the cosine measure with concepts from the ontology being introduced as additional dimensions in the vector space.

In the second category, Swoogle [18] is a Semantic Web Search Engine developed at the University of Maryland. Swoogle crawls and indexes all types of Semantic Web Documents (SWDs); these documents are indexed and stored in a triple store database. Swoogle allows this database to be queried using a simple keyword based interface; all ontologies which match the keywords are returned to the user, with additional contextual descriptions providing information from linked SWDs. The

interface to Swoogle limits its usefulness as a query tool. Only simple keyword based searches are possible, and additional work must be performed if one requires ontologies which contain certain structures or if instance data for a particular class is required.

OntoKhoj [58] is a system developed by the University of Missouri. It crawls the Web searching for ontologies which it aggregates and classifies using an intelligent algorithm which is trained using the DMOZ database [1]. The latter contains a large number of websites, sorted into human classified categories. Its search rankings are performed using an algorithm influenced by the Pagerank algorithm developed by Google. The ontologies are ranked using a calculated weighting based on the number of hyperlinked references to the ontology from other Semantic Web Documents. These are prioritised by the type of relationship: instantiation, sub-class and domain/range. OntoKhoj suffers from the same drawback for ontology searching as Swoogle, in that only keyword based searching is possible. OntoKhoj differs from Swoogle in that OntoKhoj only allows searching of ontologies, not of other Semantic Web documents which reference ontologies.

As for the indexing schemes for RDF data storage, Harth and Decker proposed storing RDF data based on multiple indices, while taking into consideration context information about the provenance of the data [29]. It constructs six indexes that cover all $2^4 = 16$ possible access patterns of quads in the form $\{s, p, o, c\}$, where c a is the context of triple $\{s, p, o\}$. This scheme allows for the quick retrieval of quads conforming to an access pattern where any of s, p, o, c is either specified or a variable. Thus, it is also oriented towards simple statement-based queries, but it

---

[1]http://www.dmoz.org/

does not allow for efficient processing of more complex queries such as conjunctive queries.

A similar multiple-indexing approach has been suggested by Wood et al. in the Kowari system [82]. Kowari also stores RDF statements as quads, in which the first three items form a standard RDF triple and a fourth, meta item, describes which model the statement appears in. Like the work of Harth and Decker [29], Kowari identifies six different orders in which the four node types can be arranged such that any collection of one to four nodes can be used to find any statement or group of statements that match it. Thus, each of these orderings acts as a compound index, and independently contains all the statements of the RDF store. Kowari solution does not consider the $4! = 24$ possible permutations of the four quad items, neither the $3! = 6$ possible permutations of the three items in a triple. Thus, if the meta nodes are ignored, the number of required indices is reduced to 3, defined by the three cyclic orderings $\{s, p, o\}$, $\{p, o, s\}$, and $\{o, s, p\}$. These indices cannot provide, for example, a sorted list of the subjects defined for a given property. Thus, Kowari does not allow for efficient processing of more complex queries either.

Then, in order to support complex queries using a multi-index approach, Hexastore [80] attempted to achieve scalability by replicating each triple six times: one for each sorting order of subject, predicate and object. Then, there are six indexing schemes: $\{s, p, o\}$, $\{s, o, p\}$, $\{p, s, o\}$, $\{p, o, s\}$, $\{o, s, p\}$ and $\{o, p, s\}$. Meanwhile, in order to save storage space, Hexastore employs a dictionary encoding. Instead of storing entire strings or URIs, it stores shortened versions or keys. In particular, it maps string URIs to integer identifiers. Thus, apart from the six indices using

identifiers (i.e., keys) for each RDF element value, a Hexastore also maintains a mapping table that maps these keys to their corresponding strings. The Hexastore has been demonstrated that this strategy results in good response time for conjunctive queries.

YARS2 [31] is another native RDF store system where index structures and query processing algorithms are designed from scratch and optimized for RDF processing. Like the work of Harth and Decker [29], YARS2 stores RDF data in the form of quads $\{s, p, o, c\}$ by extending standard RDF data model (subject, predicate, object) with the context. The proposed index structure in YARS2 is based on lexicon and quad indexes. The lexicon indexes operate on the string representation of RDF nodes and allow the retrieval of their object identifiers (OIDs). It consists of nodeoid, oidnode and the keyword indexes. While the nodeoid and the oidnode are used to map OIDs to node values and vice versa, the keyword indexes keep an inverted index to string literals in order to speed up full-text searches operations. The quad indexes allow the management of more complex queries. As each element of the quad $\{s, p, o, c\}$ can be either specified or a variable, there is $2^4 = 16$ possible access patterns. In order to save storage space, an optimized solution was proposed allowing the coverage of all access patterns using only 6 indexes rather than 16. For instance, the *poc* index is used to process not only $\{?, p, ?, ?\}$ but also $\{?, p, o, ?\}$ and $\{?, p, o, c\}$. The novelty of YARS2 lies in the use of multiple indexes to cover different access patterns. However, in YARS2, if more efficient query processing can be achieved, more disk space will be still needed.

GRIN [78] is a novel index developed specifically for graph-matching queries in RDF. It identifies selected central vertices and identifies the distance of other nodes

from these vertices. Basically, the GRIN index is a binary tree. The set of leaf nodes in the tree form a partition of the set of triples in the RDF graph. 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. Compared to the previously mentioned indexing approaches, the GRIN index is more compact, but it still is not clear how it could be adapted for a distributed context.

MonetDB [72] exploits the fact that RDF data typically has many fewer predicates than triples, thereby vertically partitioning the data for each unique predicate and sorting each predicate table on subject, object order. In other words, MonetDB applies a column-oriented data storage instead of a relational data storage. Here, the column corresponds to the predicate and works as an predicate based index. According to this way, MonetDB is very good at RDF queries with lots of predicates since only the predicates relevant for the query need to be accessed.

RDF-3X [55] employs an exhaustive-indexing approach by building clustered B^+-trees on all six S(subject), P(predicate) and O(object) permutations and also all permutations of six binary and three unary projections. For the projection indexes, the missing component(s) (S, P, or O) are replaced by count aggregates, for fast statistical lookups. In all indexes, all S, P, O components are implemented as integer identifiers rather than the original literals (URLs or string constants). RDF-3X uses a dictionary with literal-to-identifier and identifier-to-literal mappings to speed up full-text search. RDF-3X uses query optimization techniques such as selectivity estimation, sort-joins and hash-joins.

2.3   Query Optimization

Since my query optimization algorithms got inspired by Database query optimization techniques, in this section, I mainly survey the query optimization literatures in Database and RDF SPARQL query areas.

In Database, query optimization is the process of selecting the most effcient query-evaluation plan for a query. To choose among different query-evaluation plans, the optimizer has to estimate the cost of each evaluation plan. Computing the precise cost of evaluation of a plan is usually not possible without actually evaluating the plan. Instead, optimizers make use of statistical information about the relations, such as relation sizes and index depths, to make a good estimate of the cost of a plan. Once the query plan is chosen, the query is evaluated with that plan, and the result of query is output. During this process, we first need a cost estimation model used to evaluate the cost of each plan. Then, we also need an enumeration mechanism to examine each query plan, assign costs and chooses the one with lowest cost. In this step, the join ordering is particularly important because the performance of a query plan is determined largely by the order in which the relations are joined.

The database query optimizer generally makes use of statistical information stored in the DBMS catalog to estimate the cost of a plan. The relevant catalog information about relations includes:

- The number of tuples in the relation.

- The number of blocks containing tuples of the relation.

- The size of a tuple of the relation in bytes.

- The blocking factor of the relation - that is, the number of tuples of relation that fit into one block.

- The number of distinct values that appear in the relation for some specific attribute.

- The selection cardinality of one attribute of the relation.

In real implementation, since the update of the above catalog information incurs a substantial amount of overhead, most systems do not update them on every modification. Instead, the updates are done during periods of light system load.

In the join ordering optimization, there are a number of join algorithms that can potentially be used, depending on what catalog information used to compute the join cost such as the number of tuples of the input tables, the number of rows that match the join condition, and the operations required by the rest of the query. A brief explanation of the more commonly occuring join types and their corresponding improvements is described as follows:

- Nested loops: For each tuple in the outer join relation, the entire inner relation is scanned and any tuples that match the join condition are retained. If either of the tables is very large, the efficiency of this algorithm drops substantially. There are many improvements of it have been proposed. The Block nested loops join is one such work [81]. It only scans the entire relations for each block in the outer relation, as opposed to for each tuple in the nested loop. This results in more computation for each tuple in the inner relation, but requires far less scans of it.

- Sort-based join: This join technique [17] sorts both joining relations on the join attributes into two sorted lists and then merges these two sorted lists. Tuples are joined on the fly in the merging phase. As soon as two tuples from the two relations have their join attribute values match, these two tuples are joined and output and merging resumes in the tuples (they are sorted and therefore are ordered by their join attribute values) that follow. Otherwise, the tuple with the smaller attribute value between the two is dropped because it has no hope to be joined with other tuples. This join is useful for joins between large relations without indexing supports [81], especially when the join predicate does not offer much filtering (the join result is large).

- Hash join: A hash function is applied to the join attribute of the smaller relation, and a hash table is built [81]. The larger table is then scanned and the relevant rows found by looking into the hash table. This is done by computing the same hash value on the hash key (join attribute) and checks for a match in the hash table. The advantage of this join is that it is only necessary to read each table once and no sorting is necessary. Ideally, the smaller relation should be able to fit into main memory.

If both relations are sorted on the join attribute, then the Sort-based join is the most efficient. If one of the relations is very large and indexes are used, nested loops are preferred. For cases where one of the relations is small enough to fit into main memory, block loops or hash joins are favoured. Hash joins work best when there is a very large difference in the size of the relations. The efficiency of the Sort-based join is one of the reasons why we're interested in the order or the relation that results from a join.

Besides, another important query optimization approach is *Magic Sets* [4], which is a general algorithm for rewriting queries to compute the fix point of cyclic axioms. Here, a cyclic axiom is one that references the same (or equivalent) classes (or properties) on both sides of the subsumption relation. For instance, $\exists P.C \sqsubseteq C$ is a cyclic axiom, where C is a class and P is a property. The *Magic Sets* applies the sideways information passing strategy (SIPS), executes a top-down evaluation of a query by modifying the original program by means of additional rules and improves the query answering efficiency by restricting the computation to facts that are related to the query. The SIPS strategy describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. For instance, let V be an atom that has not yet been processed, and Q be the set of already considered atoms, then a SIPS specifies a propagation $Q \rightarrow_X V$, where X is the set of the variables bound by Q, passing their values to V.

In RDF SPARQL query optimization, since SPARQL queries can be executed as SQL queries, a lot of SQL query optimization techniques have been applied to improve the performance of SPARQL queries. Stocker Markus et al. [74] proposed a selectivity-based query optimization approach, which calculates the selectivity of each query triple pattern in the Basic Graph Pattern (BGP) and then uses the selectivity information to direct the query planning. The selectivity heuristics they used are calculated as follows:

- Variable Counting (VC): the selectivity of a triple pattern is computed according to the type and number of unbound components and is characterized by the ranking $sel(s) < sel(o) < sel(p)$, i.e. subjects are more selective than objects and objects more selective than predicates.

- Variable Counting Predicates (VCP): it is very close to VC. The only difference is a default selectivity of 1.0 for triple patterns joined by bound predicates.

- Triple Pattern Selectivity (TPS): it is estimated by the formula $sel(t) = sel(s) \times sel(p) \times sel(o)$, where $sel(t)$ denotes the selectivity for the triple pattern t, $sel(s)$ the selectivity for the subject s, $sel(p)$ the selectivity for the predicate p, and $sel(o)$ the selectivity for the object o. The (estimated) selectivity is a real value in the interval $[0, 1]$ and corresponds to the (estimated) number of triples matching a pattern.

- Join Triple Pattern Selectivity (JTPS): given the upper bound size S_P for a joined triple pattern P, the selectivity of P is estimated as $sel(P) = \frac{S_P}{T^2}$, where S_P denotes the upper bound size of P and T^2 denotes the square of the total number of triples in the RDF dataset. Given two related properties p_1, p_2 and their join relation (relation type), the S_P is estimated as the number of triples satifying a BGP of p_1 and p_2 such as $\{\langle ?x, p_1, ?y\rangle, \langle ?x, p_2, ?z\rangle\}$.

Michael Schmidt [68] proposed a scheme for the constraint-based query optimization of SPARQL queries. Generally speaking, the key idea of constraint-based optimization is as follows. Given a query and a set of integrity constraints over the database, the goal is to find more efficient queries that are semantically equivalent to the original query for each database instance that satisfies the constraints. The constraints that are given as input may have been specified by the user, automatically extracted from the underlying database, or may be implicitly given by the semantics of RDFS when SPARQL is coupled with an RDFS inference system. Michael's work first translates And-only blocks (or full And-only queries), into conjunctive

queries. In a second step, it then uses the Chase & Backchase (C&B) algorithm [16] to minimize these conjunctive queries and translate the minimized conjunctive queries (i.e., the output of the C&B algorithm) back into SPARQL, which usually gives more efficient SPARQL queries. The C&B algorithm does the following: given a conjunctive query q and a set of constraints as input, it lists all minimal (with respect to the number of atoms in the body) rewritings of q that conform to the given constraints.

Edna Ruckhaus et al. presented a cost-based SPARQL query optimization method [66]. In their approach, ontologies are modeled as a deductive database. The extensional database is comprised of meta-level predicates (e.g., subClassOf) that represent the information explicitly modeled by the ontology. The intensional database corresponds to the deductive rules that implement the semantics of the vocabulary terms (e.g., the transitive properties of the subClassOf term). Then, they proposed a hybrid cost model to estimate the cardinality and evaluation cost of the predicates that represent the ontologys extensional and intensional facts. Extensional fact estimates are computed using traditional relational database cost models. Conversely, to estimate the cost and cardinality of data that do not exist a priori, which is the case of the intensional facts, sampling techniques are applied. The cardinality is defined the number of valid instantiations of each predicate. The size of sample is re-estimated as elements are selected from the data.

Besides, Harth et al. proposed a data structure - Data Summaries (DS) - that aims to improve the efficiency of Linked Data query processing [30]. In order to deal with a large number of sources, DS uses a combined description of instance- and schema-level elements to summarise the content of data sources. It uses a

summarising index - a data summary - which represents an approximation of the whole data set. Compared to the schema-level indexes, the DS approach uses more resources, however, adds the ability to cover also query patterns including instance-level queries. Since the DS returns sources which possibly contain answers to a query directly (i.e., taking joins into account), this approach may be viewed as subsuming both direct lookups and schema-level indexes. Further, a data summary index can be updated incrementally as the query processor obtains new or updated information about sources.

2.4   Information Integration

For many years, distributed database researchers [21] have considered the problems of querying multiple databases and semantic heterogeneity [57]. The problem of information integration has been widely researched. There are two main approaches in information integration that relates sources to a query. The first approach known as Global-as-View (GAV), expresses the global schema relations as a set of views over the data source relations [14]. The second approach known as Local-as-View (LAV) expresses the source relations as views over the mediated schema [41]. In the following, I will first introduce GAV and LAV. Then, I will describe the meta-search engine. Finally, I will discuss ontology-related information integration.

2.4.1   GAV

As shown in Figure 2.3, in GAV approach, each concept $r(\bar{X})$ of the global schema is modeled to be a set of views over the data sources $(S_1, ..., S_n)$. In this way, query
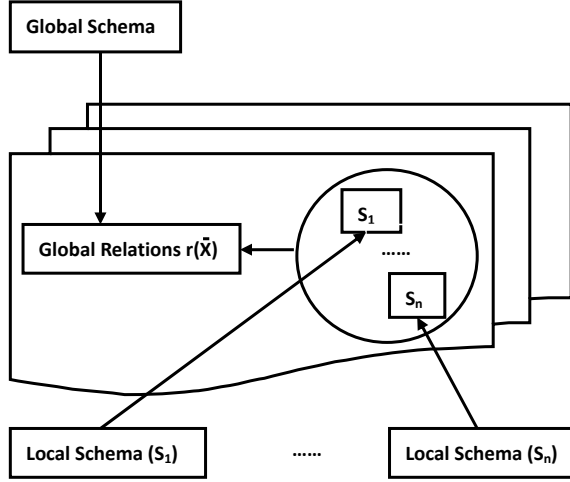
Figure 2.3: Global-As-View

processing is conceptually simpler, because it amounts to replacing (or unfolding) each global concept in the user query with the associated views over the sources and then executing the unfolded query over the sources. However, the approach does not cope well with dynamicity and changes in the sources, since such changes potentially affect all mappings and require restructuring the global view. This makes GAV a good choice when the sources are less likely to change.

Formally, GAV source descriptions have the form: $S_1(\bar{X}_1, \bar{Y}_1) \wedge ... \wedge S_j(\bar{X}_j, \bar{Y}_j) \Rightarrow r(\bar{X})$, where S_i are source relations, r is a mediated schema relation, $\bar{X}_i$ stands for the distinguished variables, $\bar{Y}_i$ stands for the non-distinguished variables and $\bar{X} = \bigcup_i \bar{X}_i$. Here, the distinguished variables are bound variables whose values would be the answers to the query. The non-distinguished variables are free variables that only specify places where the substitution may take place. They only appear in the body of the mapping formulas.

42

The GAV query processing is to translate the user queries under the control of GAV mappings. This translation process is called view unfolding (unnesting). It can be described by the following example.

Assume we have three GAV mapping rules:

- $DB_1(id, title, actor, year) \Rightarrow MovieActor(title, actor)$.

- $DB_2(id, title, cirector, year) \Rightarrow MovieActor(title, director)$.

- $DB_1(id, title, actor, year) \wedge DB_3(id, review) \Rightarrow MovieReview(title, review)$.

We also have one query "find reviews for 'DeNiro' movies" could be formalized (in respect to the global schema) as follows:

- $q(title, review) \Leftarrow MovieActor(title, \text{'DeNiro'}), MovieReview(title, review)$.

Because in GAV we have views for each schema entity, the query is processed by means of view unfolding, i.e., by expanding the atoms according to their definitions. For the above example, the subgoals $MovieActor(title, \text{'DeNiro'})$ and $MovieReview(title, review)$ will be matched with the heads of the given three GAV mapping rules. Here, the first and the third rule are selected. Then, replace the subgoals with the selected mapping rules. At the same time, apply any substitutions into the rules for unification. Here, the used substitution is "actor/DeNiro". Then, we can get the following results:

- $DB_1(id, title, \text{'DeNiro'}, year) \Rightarrow MovieActor(title, \text{'DeNiro'})$.

- $DB_1(id, title, \text{'DeNiro'}, year) \wedge DB_3(id, review) \Rightarrow MovieReview(title, review)$.

After removing the redundant terms, the query reformulation result is:

- $DB_1(id, title, \text{`DeNiro'}, year) \land DB_3(id, review) \Rightarrow q'(title, review)$.

In GAV applications, representative systems are Tsimmis [14], Garlic [13] and MOMIS [7]. Tsimmis stands for "The Stanford IBM Manager of Multiple Information Sources". It was a DARPA funded joint project of the Stanford database group and the IBM Almaden database research group. The IBM team later developed their own information integration project Garlic [13]. Tsimmis creates a hierarchy of wrappers and mediators that talk to one another. The wrappers are used to convert data from each source into a common data model called OEM (Object Exchange Model). The mediators are used to combine and integrate data exported by wrappers or by other mediators. In this framework, the global schema is essentially constituted by the set of OEM objects exported by wrappers and mediators. Mediators are defined by using a logical language called MSL (Mediator Specification Language), which is essentially Datalog extended to support OEM objects. OEM is a semistructured and self-describing data mode. Each OEM object has associated a label, a type for the value of the object and a value. Users' queries are posed in terms of objects synthesized at a mediator or directly exported by a wrapper.

The Garlic project, developed at IBM's Almaden Research Center, provides the user with an integrated data perspective by means of an architecture comprising a middleware layer for query processing and data access software called Query Services & RunTime System. The middleware layer presents an object-oriented data model based on the ODMG standard that allows data from various information sources to be represented uniformly. In such a case the global schema is simply constituted by the union of local schemas, and no integrity constraints are defined over the

OMDG objects. The objects are exported by the wrappers using the Garlic Data Language (GDL), which is based on the standard object Definition Language (ODL). Each wrapper describes data at a certain source in the OMDG format and gives descriptions of source capabilities to answer queries in terms of the query plans it supports. Note that the notion of mediator in Tsimmis is missing in Garlic, and most of the mediator tasks, as the integration of objects from different sources, are submitted to the wrappers. Users pose queries in terms of the objects of the global schema in an object-oriented query language which is an object-extended dialect of SQL.

The MOMIS system, jointly developed at the University of Milano and the University of Modena and Reggio Emilia, provides semi-automatic techniques for the extraction and the representation of properties holding in a single source schema, or between different source schemas, and for schema clustering and integration, to identify candidates to integration and synthesize candidates into an integrated global schema. The relationships are both intensional and extensional, either defined or automatically inferred by the system. In MOMIS, mediators are composed of two modules:

- The Global Schema Builder, which constructs the global schema by integrating the source descriptions provided by the wrappers, and by exploiting the intraschema and the interschema relationships.

- The Query Manager, which performs query processing and optimization. The Query Manager exploits extensional relationships to first identify all sources whose data are needed to answer a user's query posed over the global schema. Then it reformulates the original query into queries to the single sources, sends
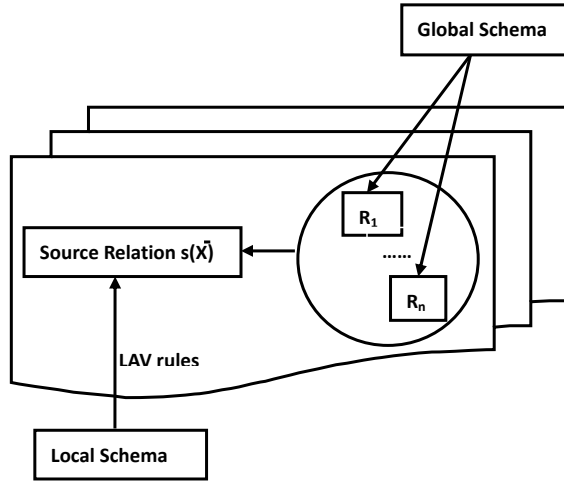
Figure 2.4: Local-As-View

the obtained sub-queries to the wrappers, which execute them and report the results to the Query Manager. Finally, the Query Manager combines the singled results to provide the answer to the original query.

2.4.2 LAV

As shown in Figure 2.4, in LAV approach, the source relations $s(\bar{X})$ are modeled as a set of views over an underlying global schema $(R_1, ..., R_n)$. The advantage of this model is that new sources can be added easily when compared to GAV. For example, a shopping agent is a good candidate for an LAV approach. However the query rewriting process is complex because the system has to choose from a set of choices to determine the best possible rewrite.

Formally, LAV source descriptions have the form: $s(\bar{X}) \Rightarrow R_1(\bar{X}_1, \bar{Y}_1) \wedge ... \wedge R_j(\bar{X}_j, \bar{Y}_j)$, where s is a source relation, R_i are mediated schema relations, $\bar{X}_i$ stands

for the distinguished variables, $\bar{Y}_i$ stands for the non-distinguished variables and $\bar{X}$ = $\bigcup_i \bar{X}_i$. Here, the definitions of distinguished and non-distinguished variables are the same meaning as that in section 2.4.1.

The LAV query processing is to translate the user queries under the control of LAV mappings. Still use the Movie example. Assume we have the following LAV mapping rules:

- $DB_1(title, year, director) \Rightarrow Movie(title, year, director, genre) \wedge American$ $(Director) \wedge year \geq 1960 \wedge genre =$ 'Comedy'.

- $DB_2(title, review) \Rightarrow Movie(title, year, director, genre) \wedge year \geq 1990$ $\wedge MoviewReview(title, review)$.

We also have one query "find reviews for comedies produced after 1950" could be formalized (in respect to the global schema) as follows:

- $q(title, review) \Leftarrow Movie(title, year, director,$ 'Comedy'$) \wedge year \geq 1950 \wedge$ $MoviewReview(title, review)$.

Because in LAV both the query and mappings target the global schema, it is not trivial to determine how to map the query to the local sources. This process is performed by means of an inference mechanism that re-expresses atoms of the global schema in terms of atoms of the sources. The final reformulation result should be as follows:

- $q'(title, review) \Leftarrow DB_1(title, year, director) \wedge DB_2(title, review)$.

Here, $q' \subseteq q$, which means the answers after query reformulation are the subset of the answers before query reformulation because both source views provide only partial answers for the original query in our example. Note, the results of this reformulation only provide reviews for comedies after 1990, but no reviews are available for older comedies. So, this reformulation is the best that the system can do.

In LAV applications, representative systems are Information Manifold [41], Infomaster [20], MiniCon algorithm [60] and SIMS [1]. The Information Manifold system provides uniform access to a heterogeneous collection of information sources on the Web. It is based on a mechanism consisting of a world view and some source descriptions. The world view is defined as a collection of virtual relations and classes, which model the features that are useful for describing and reasoning about the contents of information sources. In LAV terminology, the world view corresponds to the global schema. In source descriptions, the contents are first modeled as tuples in one or more relations. Then, these relations are described as queries over the world-view relations. The relations and its related queries over the world view essentially correspond to the LAV mapping rules. The core algorithm developed in the Information Manifold system is the bucket algorithm, which is to reformulate a user query that is posed on a mediated (virtual) schema into a query that refers directly to the available data sources. This algorithm proceeds in two steps. In the first step, the algorithm creates a bucket for each subgoal in q, containing the views (i.e., data sources) that are relevant to answering the particular subgoal. In the second step, the algorithm considers query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. The source selection process applies the LAV rules. This process can be formalized as follows:

Given a user query q and some source descriptions $\{S_i | 1 \leq i \leq n\}$.

- Find relevant sources (fill in the buckets): for each relation g in query q, find S_i where the relation g appearing in its body part. Then, check if constraints in case of S_i and q are satisfiable.

- Consider as candidate rewriting each conjunctive query q' obtained by combining $\{S_j\}$ from each bucket and check for containment ($q' \subseteq q$). If so, the candidate rewriting is added to the answer.

- If the candidate rewriting is not contained in q, before discarding it, the algorithm checks if it can be modified by adding comparison predicates in such a way that it is contained in q.

The advantages of the bucket algorithm include:

- It takes into account context to prune search space.

- It takes advantage of materialized views to reformulate queries.

However, it still has some limitations:

- It uses Cartesian product of the buckets to find rewritings. Thus, it cannot scales very well if there are many buckets or many views per bucket.

- It is hard to recover projected away attributes without additional knowledge.

- It considers each sub-goal in isolation during reformulation. Therefore, it misses some important interactions between view subgoals.

Informaster is an information integration system that provides integrated access to distributed and heterogeneous information sources. It is based on an architecture that consists of wrappers and facilitators. The wrappers convert data from each source into a common schema that resides in the facilitator. The facilitator mainly maintains the world relations that are used to describe the information sources. During the conversion, the LAV mapping rules would be generated. Here, the schema in the facilitator essentially plays the role of the global schema in LAV. The core algorithm of Infomaster is the inverse-rules algorithm, which is to construct a set of rules that invert the view definitions, i.e., rules that show how to compute tuples for the database relations from tuples of the views, and provides the system with an inverse mapping which establishes how to obtain the data of the global concepts from the data of sources. The basic idea is to replace existential variables in the body of each view definition by Skolem functions. In this way, the rewriting of a query Q using the set of views V is the logic program that includes the inverse rules for V, and the query Q. Finally, the inverse-rules algorithm returns the maximally contained rewriting of Q using V. The whole process is as follows:

- Construct an equivalent logic program whose evaluation yields the answer to the query by using two steps: rewrite the antecedent of the query and views in terms of global predicates by using the definition of the associated predicates, and reformulate the global predicates in the antecedent with the source predicates by using LAV mapping rules.

- Invert the rules simply by using standard logical manipulations.

- Use the selected source predicates in the obtained query rewritings to solve

the query.

The advantages of the inverse-rules algorithm include:

- It has conceptual simplicity and good modularity.

- It can return maximally contained rewriting of query using views.

- It scales better than the bucket algorithm.

One typical disadvantage of this algorithm is that it needs additional constant propagation to trim redundant computations.

The MiniCon algorithm combines the ideas from the bucket algorithm and inverse rules algorithm. It identifies a minimal subset of views that is required to answer a query. In the process, this algorithm creates a MiniCon (Minimal Construction) description for each set of query subgoals that cover a view. It works as follows:

- Begin like the Bucket Algorithm.

- Form the MiniCon Descriptors (MCD-s): for each subgoal g in the query Q mapped to subgoal g' in view V (bucket), look at the variables in Q and consider the join predicates to find the minimal set of subgoals in Q that must be mapped to the subgoals in V in order to make V usable.

- Combine MCD-s. This step proceeds as in the bucket algorithm but considers rewritings involving only the disjoined subgoals of the query because the join relations have been processed in last step.

Compared to the bucket algorithm and the inverse-rules algorithm, the MiniCon algorithm has the following advantages:

- It scales best with the number of available views.

- It requires less work during the combination.

- Its speed performance is improved because there is no containment check operation in combining phase.

However, the main disadvantage of MiniCon is the computation of MiniCon Descriptors for each goal mapping. In this step, it requires more preprocessing to build these Descriptors.

SIMS is a flexible and efficient information mediator that takes a domain-level query and dynamically selects the appropriate information sources based on their content and availability. The SIMS model of the application domain includes a hierarchical terminological knowledge base with nodes representing objects, actions, and states. An independent model of each information source must be described for this system by relating the objects of each source to the global domain model. The relationships clarify the semantics of the source objects and help to find semantically corresponding objects. Here, these relationships are basically LAV mapping rules.

2.4.3   Meta-search Engine

A *meta-search engine* is a system that provides unified access to multiple existing search engines. A meta-search engine does not maintain its own index of documents. However, a sophisticated meta-search engine may maintain information about the
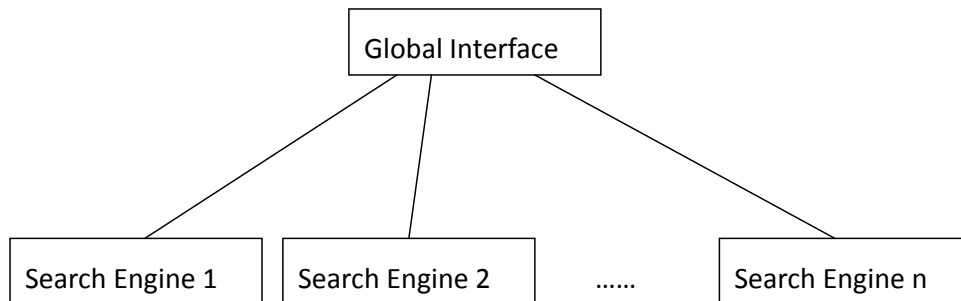
Figure 2.5: A simple meta-search architecture

contents of its underlying search engines to provide better service. In a nutshell, when a meta-search engine receives a user query, it first passes the query (with necessary reformatting) to the appropriate underlying search engines, and then collects and reorganizes the results received from them. A simple two-level architecture of a meta-search engine is depicted in Figure 2.5. This two-level architecture can be generalized to a hierarchy of more than two levels when the number of underlying search engines becomes large [83].

One core issue related to this dissertation that the meta-search engine aims to tackle is the document selection, which means to determine what documents to retrieve. A naive approach is to let each selected component search engine return all documents that are retrieved from the search engine. The problem with this approach is that too many documents may be retrieved from the component systems unnecessarily. As a result, this approach will not only lead to higher communication cost but also require more effort from the result merger to identify the best matched documents. Thus, the following four categories of approaches were proposed.

- User determination: the meta-search engine lets the global user determine how

many documents to retrieve from each component system. One representative
system to this approach is called Savvy Search proposed by Dreilinger and
Howe [19]. In this system, the maximum number of documents to be returned
from each component system can be customized by the user. Different numbers
can be used for different queries. If a user does not select a number, then a
query-independent default number set by the meta-search engine will be used.
This approach may be reasonable if the number of component systems is small
and the user is reasonably familiar with all of them.

- Weighted allocation: the number of documents to retrieve from a component
  system depends on the ranking score (or the rank) of the component system
  relative to the ranking scores (or ranks) of other component systems. As a
  result, proportionally more documents are retrieved from component systems
  that are ranked higher or have higher ranking scores.

  In D-WISE [84], the ranking score information is used. For a given query q,
  let r_i be the ranking score of component system D_i, $i = 1, \ldots, N$, where N
  is the number of selected component systems for the query. Suppose m docu-
  ments across all selected component systems are desired. Then the number of
  documents to retrieve from the system D_i is $\frac{m \times r_i}{\sum_{j=1}^{n} r_j}$.

- Learning-based approaches: these approaches determine the number of docu-
  ments to retrieve from a component system based on past retrieval experiences
  with the component system. One representative work called QC (Query Clus-
  tering) performs document selection based on past retrieval experiences. It
  utilizes a set of training queries. In the training phase, for each component

system, the training queries are grouped into a number of clusters. Two queries are placed in the same cluster if the number of common documents retrieved by the two queries is large. Next, the centroid of each query cluster is computed by averaging the vectors of the queries in the cluster. Furthermore, for each component system, a weight is computed for each cluster based on the average number of relevant documents among the top T retrieved documents (T = 8 performed well as reported in [76]) for each query in the query cluster. For a given system, the weight of a cluster indicates how well the system responds to queries in the cluster. When a user query is received, for each component system, the query cluster whose centroid is most similar to the query is selected. Then the weights associated with all selected query clusters across all systems are used to determine the number of documents to retrieve from each system.

- Guaranteed retrieval: this type of approach aims at guaranteeing the retrieval of all potentially useful documents with respect to any given query. Many applications, especially those in medical and legal fields, often desire to retrieve all documents (cases) that are similar to a given query (case). For these applications, the guaranteed retrieval approaches that can minimize the retrieval of useless documents would be appropriate. One representative work is called query modification [51]. According to this method, under certain conditions, a global query can be modified before it is submitted to a component system to yield the global similarities for returned documents. This method is essentially a query translation method for vector queries. Clearly, if a component system

can be tricked into returning documents in descending order of global similar-
ities, guaranteeing the retrieval of globally most similar documents becomes
trivial.

2.4.4   Ontology-related Information Integration

T. Tran et al. proposed Hermes, which translates a keyword query provided by the
user into a federated query and then decomposes this into separate SPARQL queries
that are issued to web data sources [77]. During this process, a number of indexes
are used, including a keyword index, a structure index and a mapping index. The
keyword index is constructed by extracting the labels of data graph elements. During
this process, a standard lexical analysis including stemming, removal of stopwords
and term expansion using Lexical Resources (e.g. WordNet) is performed. The
structure index is constructed by extracting the available schema information of
the given data graph. If no schema information is available, the summarization
techniques is applied to construct a schema graph. The mapping index is to store
the mapping relations between different elements discovered at both the data-level
and schema-level. Hermes uses these mappings to integrate publicly available data
sources. Given a query consisting of a set of keywords, Hermes first translates the
query into a set of terms using the keyword index. The structure index is employed
to contruct query graphs based on the output of the keyword index. Then, the
selected queries are decomposed into parts that can be answered using a particular
data source. Finally, the results retrieved for each query are combined as the final
answers to the original query. However, the most significant drawback to Hermes
is that it does not account for rich schema heterogeneity (mappings are basically of

the subclass/equivalent class variety).

Cosmin B. and Abraham B. proposed a system called Avalanche for querying the data of the Semantic Web [6]. According to this approach, first, some participating hosts that can potentially answer the given query are identified via means of a Search Engine or Web directory. A lightweight endpoint-schema inverted index can also be used in this step. Then, during the query execution, the given query is broken into the superset of all molecules, where a molecule is a subgraph of the overall query graph. A combination of minimally overlapping molecules covering the directed query graph is identified and all molecules in this combination are bound to physical hosts determined in the first step. In this process, an objective function considering the number of molecules, the network latency, etc. is employed to direct the generation of an optimal query execution plan to anwer the original query. After this step, the selected query plans are executed and the results are finally materialized to make the final solutions for the given query. However, Avalanche does not consider the ontology heterogeneity and ontology integration.

Gunter L. and T. Tran presented a query processing strategy for linked data on the Semantic Web [39]. This strategy employs a mixed strategy of a bottom-up approach that discovers new sources during query preprocessing and a top-down strategy that relies on complete knowledge about the sources to select and process relevant sources. According to this strategy, depending on available source descriptions, an optimal query plan considering triple pattern cardinality, join pattern cardinality, histograms, etc. is first constructed during query compilation. For evluating the query according to the query plan, each operator is run in a separate thread. A source monitor is also run in an individual thread to receive new sources

discovered in the query evaluation. Then, the source retriever requests data from the new sources at once, which is accomplished by running more than one source retrieval threads. When maximum discovery distance, maximum number of sources to load or number of results to produce is achieved, the query processing is terminated. However, this approach does not consider the ontology heterogeneity in the ontology integration. In addition, its completeness is very hard to measure due to the unpredictable nature of linked data access.

Schwarte A. et al. proposed a distributed query processing mechanism called FedX for the data of the Semantic Web [69]. According to this approach, first, a global query is formulated against a federation of data sources. Then, the global query is parsed and optimized for the distributed setting. In particular it is split into local subqueries that can be answered by the individual data sources. Results of these local queries are merged in the federator and finally returned in an aggregated form. During the query processing, optimization strategies such as join order optimization and subquery grouping are applied in order to improve the query answering performance. However, this work does not have a clear relevant source selection strategy and also not consider the ontology heterogeneity and ontology integration.

Peer-to-peer (P2P) semantic web systems like Bibster [26] and SomeWhere [65] address the distributed nature of the Web, but each is insufficient to address the problem described in this dissertation. Peers in Bibster might have different data, but use the same ontologies. Peers in SomeWhere can have different ontologies, but the data consists only of category information (from an RDF point of view, this means all triples use the rdf:type predicate). While this is useful for sharing

bookmarks, it is not very useful for query answering. Liarou et al. use Distributed Hash Tables (DHT) to provide answers to continuous, conjunctive queries issued to a network of nodes with RDF data [48]. However, like Bibster, they do not address the schema mapping issue and therefore only work in a single ontology environment. DRAGO focuses on a distributed reasoning based on the P2P-like architecture [70]. In DRAGO, every peer maintains a set of ontologies and the semantic mapping between its local ontologies and remote ontologies located in other peers. The semantic mapping supported in DRAGO is only the subsumption relationship between two atomic concepts and ABox reasoning is not supported. KAONP2P also suggests a P2P-like architecture for query answering over distributed ontologies by creating semantic mappings among different ontologies [27]. In KAONP2P, the query evaluation is performed against a virtual ontology including a target ontology to which the query is issued and the semantic mapping between the target and the other ontologies. However, all of these P2P systems have a drawback in that each node must install system specific P2P software, presenting a barrier to adoption.

Other representative work include: Stuckenschmidt et al. presented an architecture for querying distributed RDF repositories by extending the Sesame system, and proposed an index structure as well as algorithms for query processing and optimization in a distributed context [75]. They used a hierarchy of path indexes that indicate which sources have information on which query paths. A major drawback of this work is that it does not consider schema heterogeneity. Haase and Motik developed a mapping system for OWL that involves relating conjunctive queries [25]. However, since this effectively adds rules to OWL, it is undecidable as they need to introduce restrictions to achieve decidability. They do not explicitly address the

issue of distributed data, and provide no means of indexing the relevant sources.

In addition, a number of researchers are developing modular description logics [9] [23] [5]. These logics attempt to define the semantics of connecting different ontologies while preserving their context. Such work typically defines a local interpretation for each ontology and, for each pair of ontologies, a binary relation between their domains. Although such logics often provide useful features, such as preventing local inconsistencies from polluting the global interpretation, no information integration algorithms that work at web scale have been designed for them.

Finally, I conclude this section with a detail discussion of Abir Qasem et al.'s work [62] [63], which opened a precursor way for my dissertation. The main problem they tried to solve is also query answering over Semantic Web data. In order to achieve this goal, they defined an approach that relies on a form of summary information called Relevance statements, which is also an indexing scheme to index Semantic Web data. Relevance statements are influenced by the GAV and LAV models as introduced in sections 2.4.1 and 2.4.2. They defined a language OWLII, which is a subset of OWL that is compatible with GAV and LAV. This language is slightly more expressive than DHL [24]. OWLII is defined below.

Definition 1. *The syntax of OWLII consists of DL axioms of the forms $C \sqsubseteq D$, $A \equiv B$, $P \sqsubseteq D$, $P \equiv Q$, $P \equiv Q^-$, where C is an L_a class, D is an L_c class, A, B are L_{ac} classes and P, Q are properties. L_{ac}, L_a and L_c are defined as:*

- *L_{ac} is a DL language where A is an atomic class and i is an individual. If C and D are classes and R is a property, then $C \sqcap D$, $\exists R.C$ and $\exists R.\{i\}$ are also classes.*

- *L_a includes all classes in L_{ac}. Also, if C and D are classes then $C \sqcup D$ is also*

a class.

- L_c *includes all classes in* L_{ac}*. Also, if C and D are classes then* $\forall R.C$ *is also a class.*

Then, they defined two algorithms that given ontologies in OWLII, OWLII relevance statements, and a conjunctive query, would determine the set of potentially relevant sources, which are loaded into a knowledge base system to obtain sound and complete answers to the query. In identifying all relevant sources, the algorithms reformulate the query based on the mapping ontologies.

Their first source selection algorithm was heavily influenced by the PDMS algorithm [28], which was designed to integrate peers who are related by LAV and GAV rules. Unlike traditional work in information integration, the PDMS approach does not assume a global mediated schema, and thus it is a better fit for the Semantic Web. Essentially, the PDMS algorithm creates an AND/OR graph by expanding subgoal nodes based on matches with LAV and GAV mapping rules. In order to guarantee termination when there are cycles in the rules, the algorithm does not expand nodes that have ancestors with the same content (since this will just repeat the subtree created at the ancestor node). When there are no more nodes to expand, a set of queries can be read off the tree by taking every option under an AND node and replicating for each possible OR node. Their source selection algorithm builds the same tree, but instead of constructing a set of queries, it uses the leaf nodes to decide which sources are relevant. Additionally, the algorithm performs some expansions based on axioms from the domain ontologies. The entire content of these sources is then loaded into a DL reasoner which then answers the original query. An interesting benefit of this approach is that if all map ontologies are expressed

in OWLII and the reasoner is sound and complete for OWLII, then it is sound and complete. This is because the subtrees that are eliminated in order to avoid cycles will not identify any new sources - the nodes in these subtrees are identical to nodes elsewhere in the tree [62].

Their second source selection algorithm, dubbed Goal Node Search (OBII-GNS), was a result of the observation that much of the overhead of the AND/OR graph source selection algorithm was only needed when you were trying to create a set of query rewrites [63]. They found that they could simplify the problem by keeping the LAV/GAV expansion rules but instead of creating children in a tree, then they simply created new nodes for an open list. They also maintained a closed-list of nodes that were already expanded, to avoid repeating work (and performed a job similar to the cycle-check in the and/or graph). This algorithm also allowed domain ontologies to be expressed in OWLII and expanded their axioms in the same way as those of the mapping ontologies.

As demonstrated by their empirical experiments, their algorithms have gained decent performance. However, the algorithms suffer from the following drawbacks:

- The indexing scheme requires users (or a third party) to create content summary files. This seems like an unnecessary burden that lessens the likelihood that the approach will be adopted.

- The algorithms frequently select sources that do not contribute to the eventual results. This is partially due to the nature of these content summary files. For example, it would be reasonable to expect that all individual professors will specifiy that their RDF files have relevance information on courses they teach ($\exists teaches^{-1}.\{\text{prof}-\text{uri}\}$). This can be very selective for queries

such as "*teaches*(prof−uri, *?x*)". However, if on the other hand the query is "*teaches*(*?x, prog-lang*)" then the system will simply have to select the RDF files of all professors. Note, of course each professor could have a separate statement for each course, (∃*teaches*.{prog−lang}, ∃*teaches*.{*AI*}, etc.) but then this wouldn't be much of a summary. The index would repeat most of the document's original content. This brings up another problem: the RDF for most people includes a large number of single valued properties. While, based on my real world Semantic Web data statistics using Sindice by surveying 1,000 RDF data sources, the average percentage of such properties is 17.8%. Therefore, summaries of the properties do not result in much compression of the document.

- This approach is incomplete in the presence of coreference information, that is, information about which URIs denote the same objects. In OWL, coreference can by explicitly specified by means of *owl*:*sameAs*. It should be noted the Linking Open Data initiative has over four billion RDF triples and over 100 million explicit *owl*:*sameAs* statements. Many RDF users publish *owl*:*sameAs* statements with their own data to provide the means of gluing together their descriptions with those made by others. When reformulating a query, coreference information should be used to expand any constants that appear, and it should also be used when matching subgoals with sources. For example, the subgoal *author*(*X, jhendler*) should match with a content summary author (*X, jim-hender*), assuming we know *jhendler* = *jim-hendler*. However, note that this is not sufficient to guarantee complete answers. For example, consider the query *acadDescend*(*minsky, x*)∧*author*(*x, p*). There is no

guarantee that sources with information about academic descendants of Minsky use the same identifiers as sources about authors of publications. In fact, this query should be viewed as $acadDescend(minsky, x) \wedge author(y, p) \wedge x = y$, where $x = y$ is another subgoal essential to solving the problem.

- The approach is only sound and complete for a subset of OWL.

Chapter 3

Problem Definition

This chapter primarily presents how I formally frame, decompose and define my research problem. It introduces three aspects: a theoretical foundation framework, a problem decomposition and an inverted index to integrate Semantic Web data. In Section 3.1, I theoretically characterize my problem space which deals with semantic web data (ontologies and data sources that commit to them) and extensional (i.e. fact/data related) queries. This theoretical framework formally provides the conceptual foundation of my source selection algorithm(s) that I will describe later in this dissertation. In Section 3.2, I present my research problem decomposed into several components and further elaborate each one of them. Finally, in Section 3.3, I give a detail introduction to my IR inspired indexing scheme for RDF data.

3.1 Problem Space

In this dissertation, I am interested in query answering over the Semantic Web. It is well known that Semantic Web consists of a collection of web documents that

describe several OWL ontologies [73]. For convenience of analysis, in this dissertation, I decided to follow a more traditional approach and separate ontologies (i.e. the class/property definitions and axioms that relate them) and data sources (assertions of class membership or property values). As mentioned in Sections 2.1.4 and 2.1.5, since OWL is based on DL, these ontologies are essentially DL ontologies. In the discussion that follows, I use $\mathcal{L}$ to refer to a subset of OWL DL, $\mathcal{C}$ to refer to the set of all classes, $\mathcal{P}$ to refer to the set of all properties, $\mathcal{A}_t$ to refer to the set of all terminological axioms of $\mathcal{L}$, $\mathcal{A}_a$ to refer to the set of all assertional axioms of $\mathcal{L}$, $\mathcal{D}$ to refer to the set of all individuals, and $\mathcal{U}$ to refer to the set of document identifiers (URLs in the case of OWL) in the Semantic Web.

Definition 2 (Ontology)**.** *An ontology is a set of $\mathcal{A}_t$.*

Definition 3 (Data Source)**.** *A data source is a set of $\mathcal{A}_a$.*

Now, I introduce two functions. An ontology function o that maps the set of document identifiers $\mathcal{U}$ to the set of all ontologies and a source function s that maps $\mathcal{U}$ to the set of all data sources. If some u $\in \mathcal{U}$ is a data source then o(u) is an empty set and similarly if some u $\in \mathcal{U}$ is an ontology then s(u) is an empty set.

Definition 4. *(Semantic Web Space) A Semantic Web Space SWS is a tuple $\langle \mathcal{U}, o, s \rangle$, where $\mathcal{U}$ refers to the set of document identifiers, o refers to an ontology function that maps $\mathcal{U}$ to a set of ontologies and s refers to a source function that maps $\mathcal{U}$ to a set of data sources.*

Given a semantic web space SWS, I assume that there are two types of ontologies: *mapping ontologies* and *domain ontologies*. The *mapping ontologies* use logical axioms to describe the relationships between terms in heterogeneous *domain*

ontologies. I do not assume that there are mappings between all pairs of ontologies - instead I leave it to an algorithm to compose chains of mappings to discover mappings between ontologies. I will apply the OWLII language - a subset of OWL DL (Description Logics), as mentioned in Section 2.4.4, to express the mapping and domain ontologies [62].

A knowledge base K satisfies a set of logical sentences α iff each logical sentence L_α in α is true when each variable in L_α is assigned a member value of K. I define the satisfaction of $o(u)$, $s(u)$ per the official OWL semantics document [59].

Definition 5 (Satisfaction). *An interpretation $\mathcal{I}$ satisfies a Semantic Web Space $\langle \mathcal{U}, o, s \rangle$, iff for each $u \in \mathcal{U}$, $\mathcal{I}$ satisfies o(u) and s(u).*

A knowledge base entails (written $\models$) a set of logical sentences α iff every interpretation that satisfies the knowledge base also satisfies α. The notion of entailment of a SWS is defined as follows.

Definition 6 (Semantic Web Space Entailment). *Given a set of description logic sentences α, $SWS \models \alpha$ iff every interpretation that satisfies SWS also satisfies α.*

DL as a query language is more suitable for posing queries on TBox. In addition to satisfiablity and consistency checking of an ABox, the only other ABox inference available via basic DL mechanism is instance retrieval. The instance retrieval problem is as follows: given a DL Abox and a concept C find all individuals a such that Abox $\models$ *C(a).* Basically, DL query facility does not allow us to ask questions about roles, which arguably is more significant for practical data intensive applications than instance retrieval (a telephone number of a certain person as opposed to all the telephone numbers in a knowledge base).

To address this shortcoming of DL based ABox queries, Horrocks *et al.* [32] have proposed the use of conjunctive queries over DL knowledge bases. A conjunctive query is a rule whose subgoals are always extensional predicates (i.e. predicates that are actually available in a knowledge base as opposed to intensional predicates that define relationships between predicates). If a substitution of the values for the variables in the subgoals makes all the subgoals true, then the same substitution applied to the head is an inferred fact about the head's predicate i.e. an answer to the conjunctive query. The problem of finding all the answers to a conjunctive query (given a set of views) is customarily formalized using the notion of certain answers. Intuitively, a tuple t is a certain answer to a query if t is an answer for any of the possible database extensions that are consistent with the given extension of views.

It is well-known that the problem of computing certain answers and deciding query entailment can be reduced to each other and the complexity results do carry over [32, 12]. This was the basis for Horrocks *et al.*s' proposal introducing conjunctive queries over DL knowledge base. In my work I adopt this approach.

Definition 7 (Conjunctive Query Form). *A conjunctive query has the form $Q\left(\overline{X}\right)$:- $B_1\left(\overline{X}_1\right), \ldots, B_n\left(\overline{X}_n\right)$ where $\overline{X}$ is a vector of variables and/or individuals and each B_i is a query triple pattern representing a concept or role term respectively.*

Furthermore, this above defined conjunctive query corresponds to the most common SPARQL queries. Within SPARQL, each $B_i\left(\overline{X}_i\right)$ is called a query triple pattern (QTP) that is like an RDF triple, but with the option of a variable in place of RDF terms (i.e., URIs, URLs, literals or blank nodes) in the subject, predicate or object positions. I define the QTP as follows:

Definition 8 (Query Triple Pattern). *A query triple pattern is in the form of* $\langle s, p, o \rangle$, *which is the member of the set* {*(RDF-T ∪ V) × (R ∪ V) × (RDF-T ∪ V)*}, *where RDF-T is the set of RDF terms including all RDF Literals, IRIs and blank nodes, R is the set of all IRIs and V is the set of query variables.*

Within Definition 7, I refer to the left hand side of :- as the head of the query and the right hand side as the body of the query. The variables that appear in the head are called distinguished variables and describe the form of a query's answers. They are universally quantified and must appear also in the body (otherwise we end up with "undefined" variables in head). All other variables in the query are called non-distinguished variables and are existentially quantified.

I now restrict the membership of $\overline{X}$ in Q. This restriction needs to be imposed to ensure the so called DL-safety of rules introduced by Motik and Sattler [52]. According to this restriction all variables in $\overline{X}$ of Q should be mapped to individuals explicitly introduced in the data sources. Without this restriction, conjunctive query answering becomes undecidable over DL knowledge base. This is due to the possibility of non terminating reasoning process as a result of interactions between DL constructs and rules. Existential restrictions in DL create the possibility of infinite chains of inference when such DL axioms are translated in to a set of rules. Consider a DL knowledge base that contains the axioms $\{Person(Allison), Person \sqsubseteq \exists father.Person\}$. Since *Allison* must have a father, there is some $x1$ who is a Person. In turn $x1$ must have some father $x2$, who must be a Person, and infinitum.

A substitution θ is a finite set of pairs $\{x_1/t_1 \ldots x_n/t_n\}$ where x_i are distinct variables and t_i are arbitrary terms. If θ is substitution and ρ is a literal, then $\rho\theta$ is

the literal that results from simultaneously replacing each x_i in ρ by t_i. For a given query Q and substitution θ, we use $Q\theta$ as a shorthand for $B_1\theta \wedge B_2\theta \ldots \wedge B_n\theta$.

Definition 9 (Answer Set). *Given a Semantic Web Space SWS, an answer set $Ans\_SWS(Q)$ to a query Q is the set of all substitutions θ for all distinguished variables in Q such that: $SWS \models Q\theta$.*

The goal of this dissertation is to design a system that given only a conjunctive query q, the function o and some form of summary information for s, can identify a set $R \subseteq \mathcal{U}$ of data sources such that $\bigcup_{u \in U} o(u) \cup \bigcup_{u \in R} s(u)$ entails the same answers for q as does the full Semantic Web Space. Note, this selection must be done without complete knowledge of s. The assumption is that there are a large number of data sources - too many for it to be feasible to query every data source directly - and thus we need to identify a subset of sources that are potentially relevant to the query. Ideally, this set should be significantly smaller than the full set of sources, but the size will depend to some extent on the form of the summary information.

3.2 Problem Decomposition

In order to efficiently solve my problem - federated query answering over Semantic Web data, I decompose it into five components: *Indexer, GUI Convertor, Reformulator, Selector* and *Query Engine*.

In the following part, I use T to stand for the set of indexed terms, $\mathcal{U}$ to stand for the set of document identifiers as defined at Definition 4, $\mathcal{P}(\mathcal{U})$ to stand for a subset of $\mathcal{U}$, Q_{NL} to stand for the set of natrual language described queries, Q_C to stand for the set of conjunctive queries defined at Definition 7, $O = o(\mathcal{U})$ to stand

for the set of ontologies as defined at Definition 4, $\mathcal{P}(O)$ to stand for a subset of O, Q_{sub} to stand for the set of set of reformulated subgoals, *Assertions* to stand for the set of assertions extracted from $\mathcal{P}(\mathcal{U})$ in order to answer Q_C and $\mathcal{A}$ to stand for the answer set to Q_C as defined at Definition 9.

- *Indexer*: it is periodically run to create an inverted index for all of the data sources and to collect the axioms from domain and mapping ontologies. Formally, the *Indexer* is a function $I : T \rightarrow \mathcal{P}(\mathcal{U})$.

- *GUI Convertor*: a user query is input via a graphical user interface (GUI) and converted into a set of conjunctive queries. Formally, the *GUI Convertor* is a function $GC : Q_{NL} \rightarrow Q_C$.

- *Reformulator*: it uses the domain and mapping ontologies to reformulate the conjunctive query into a set of subgoals. Formally, the *Reformulator* is a function $Re : Q_C \times \mathcal{P}(O) \rightarrow Q_{sub}$.

- *Selector*: it takes the query reformulation results of *Reformulator* as inputs and uses the inverted index created by *Indexer* to identify which sources are potentially relevant to the query. Formally, the *Selector* is a function $S : Q_{sub} \times I \rightarrow \mathcal{P}(\mathcal{U})$.

- *Query Engine (QE)*: it reads the selected sources from *Selector* together with their corresponding ontologies and answers the orginal cnjunctive queries. Formally, the QE is a function: $QE : Q_C \times \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{A}$. The QE consists of two components: *Loader* and *Reasoner*.

– *Loader*: it takes the selected sources from *Selector* together with their corresponding ontologies as inputs and extracts the relevant assertions from them that are necessary to answer the original queries. Formally, the *Loader* is a function $Loader : Q_C \times \mathcal{P}(\mathcal{U}) \rightarrow Assertions$.

– *Reasoner*: a sound and complete OWL *Reasoner* is used to answer the original conjunctive queries using the extracted assertions from *Loader*. Since the selected sources are loaded in their entirety into the reasoner, any inferences due to a combination of these assertions will also be computed by the reasoner. Formally, the *Reasoner* is a function $Reasoner : Q_C \times Assertions \rightarrow \mathcal{A}$.

Corresponding to each component, I designed my system as depicted in Figure 3.1. I assume that each data source from the set $\{S_1, ..., S_n\}$ commits to one or more OWL domain ontologies from the set $\{O_1, ..., O_n\}$. Meanwhile, there are some mapping ontologies from the set $\{M_1, ..., M_n\}$ that use OWL axioms to describe the mapping relations between a pair of related domain ontologies. The choice of OWL to articulate the alignments make these mapping axioms shareable via the Web. In *Query Engine*, since *Loader* can directly be implemented using APIs provided by *Reasoner* (e.g. KAON2 in this dissertation), I will not explore it in my dissertation. As mentioned in Section 1.2, I will not address user interface issues belonging to *GUI Convertor*, assuming instead that front-ends can translate the user input into a conjunctive query. As a result, my work will be on *Indexer*, *Reformulator* and *Selector*, plus the empirical evaluation of my system.
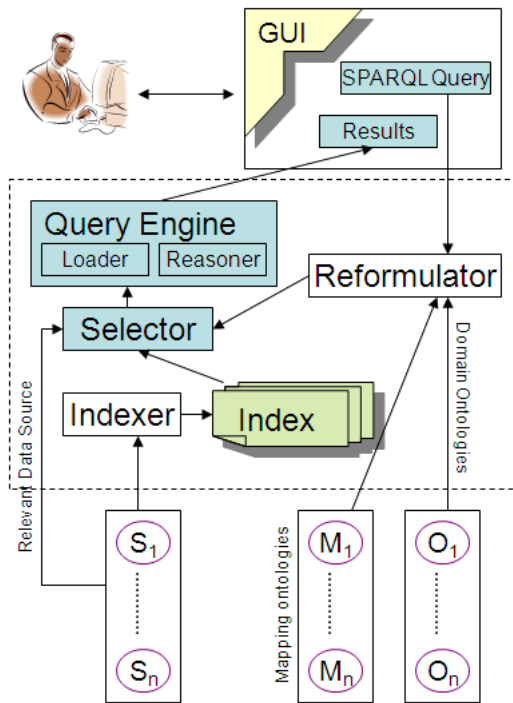
Figure 3.1: System Architecture with arrows showing the flow of information when processing a query.

3.3 IR-Inspired Indexing Scheme - Term Index

A key observation of this dissertation is that RDF documents fall somewhere between databases and free-text; they are after all semi-structured. RDF documents are more plentiful than most database solutions to information integration assume, but currently less plentiful than the number of documents of web-based information retrieval systems. For this reason, it seems that an IR-inspired approach to indexing RDF documents for use by an information integration system could address problems of automation and scale.

Recall that an RDF document is a set of triples, each with a subject, predicate and object. The subject and predicate are always URIs, but the object can be a URI or a literal. A possible lossy representation of the document is a set of URIs and literals, much in the same way that IR methods view a free-text document as a bag of words. Formally, let U be the set of URIs and L be the set of Literals, then an RDF document $d \subseteq U \times U \times (U \cup L)$. IR systems typically use an inverted index, where each term is an entry to an index that contains a posting list of documents that contain the term. To determine the terms for an RDF document, we must first tokenize the document. All tokens with the same character sequence are called types. The IR system then contains a dictionary of (possibly normalized) types. I will use the subjects, predicates, and objects of triples as the tokens. When, the object is a literal, I will further tokenize this as one would tokenize a free-text document. Instead, we can tokenize the whole literal as well. However, this way is lacking the support for string functions such as the substring search, which are often used in SPARQL queries. Therefore, I chose to tokenize each term contained in the literal, which will enhance the ability of the index to answer queries that involve

strings. Thus, the terms of the document can be formally expressed as follows:

$$terms(d) \equiv \{x| < s, p, o >\in d \wedge [x \equiv s \vee x \equiv p \vee (o \in U \wedge x \equiv o) \vee (o \in L \wedge x \in$$
$$lit - terms(o))]\}.$$

where $\langle s, p, o \rangle$ stands for a triple contained in a document d, and lit-terms() is a function that extracts terms from literals, and may involve typical IR techniques such as stemming and stopwords. The dictionary of my system is then $\bigcup_{d \in Dict} terms(d)$. Each term in the dictionary has a posting list, which is defined to be a list that records which documents contains which terms.

With the above idea, I can create an inverted index for the distributed RDF data. This index is named *Term Index*, which can be formally defined as follows:

Definition 10. *(Term Index) Given a Semantic Web Space $\langle U, o, s \rangle$, the term index is a function $I : T \rightarrow \mathcal{P}(U)$, where $T = \bigcup_{d \in U} terms(s(d))$.*

Using the term index, two basic functions (Definitions 11 and 12) are needed to determine how to select potentially relevant sources for the query answering. Note that the sources for a query triple pattern (QTP) are basically those sources that contain each constant (URI or literal term) in this QTP.

Definition 11. *(Term Evaluation) Given the set of possible query triple patterns Q and a set of constant terms T (that appear as subjects, predicates or objects of any $q \in Q$), the term evaluation function qterms: $Q \rightarrow \mathcal{P}(T)$ maps QTPs to the (non-variable) terms that appear in them.*

Definition 12. *(Source Evaluation) Given the set of possible query triple patterns Q and a set of document identifiers D, the source evaluation function is qsources: $Q \rightarrow \mathcal{P}(D)$. Given a QTP q and a term index I, qsources$(q) = \bigcap_{c \in qterms(q)} I(c)$.*
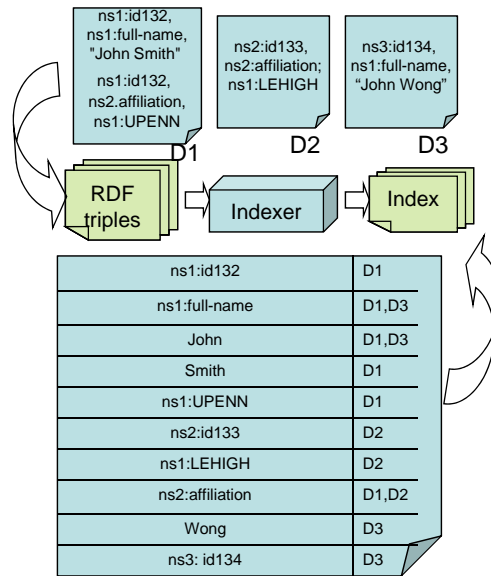
Figure 3.2: A Term Index example

An example of the term index is shown in Figure 3.2. Each token contained in the given documents D_1, D_2 and D_3 has a posting list of documents that contain it.

Since in the Semantic Web, most of the terms in RDF documents are URIs, many URIs have the same server name, and within each such set, there may be many with the same namespace, they should be very amenable to a straight-forward id compression technique, which identifies common prefixes of URIs and assigns them a special character in order to reduce the size of the dictionary string and improve the query efficiency. The id compression is a type of encoding compression algorithm whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated. As a result, it can be used to compress the lexicons used in search indexes. This can be verified in Chapter 6.

Till now, we have learned that the term index is a lookup index that indexes

3.3. IR-INSPIRED INDEXING SCHEME - TERM INDEX

RDF documents on the Semantic Web and able to tell which documents contain which terms. Sindice [56], as a well known semantic web search engine, is also a lookup index that craws and indexes resources on the Semantic Web. Thus, I will end this section by first giving a brief introduction to Sindice and then making a comparison between my term index and Sindice.

The purpose of Sindice is to allow applications to automatically retrieve sources with information about a certain resource. It offers a full-text search and also indexes SPARQL end points. Sindice regards the Semantic Web as a large collection of RDF documents. Thus, it indexes all identifiers in URIs and literal words in the graph, allows lookups over these identifiers and returns pointers to sources that mention these terms. With respect to inference support, since the relevant inferences come from OWL vocabulary, Sindice first recursively fetches and imports all the referenced schemas and then performs the inference computations. During data crawling, to balance the data ownership problem, Sindice not only employs the traditional "pull" model of Web crawlers, but a "push" model, which means data providers are offered a way to notify the indexing service of new data in order to be indexed. Regarding index construction, to be able to construct and store the index in a scalable manner, Sindice clusters machines into a parallel architecture with shared storage space, which allows to address the scarcity of processing power and storage through simple scaling of commodity hardware.

Compared to Sindice, my term index has the following characteristics:

- Similar to Sindice, my term index also indexes all identifiers in URIs and literal words in the Semantic Web and allows lookups over these identifiers. However, it does not return pointers to sources to client users as Sindice does. Instead,

it returns them to *Loader* component in Figure 3.1.

- In the inference support, both my term index and Sindice need to first fetch and preload all the referenced schemas.

- In the data crawling, my term index has not yet considered which crawling model ("push" or "pull") should be chosen. However, both models are applicable for the term index if needed.

- In the index construction, my current term index is in a centralized storage instead of a cluster storage. However, if needed, the term index can also apply a cluster storage as Sindice does in order to scale.

- In the system goal, my term index is a component in a query answering system for Semantic Web data. Thus, it does not cover SPARQL end points, which is actually already a query answering mechanism over some data sources. However, if we do not consider the index of SPARQL end points in Sindice, Sindice could play an *Indexer* role as my term index in my system.

Even though my term index and Sindice have the above similaries, Sindice cannot be an alternative to my term index in my system because of the following reasons:

- I have more control over my term index than Sindice. Thus, in my dissertation, I am able to design and implement my query optimization algirithms more easily and flexibly using my term index.

- In my system, Sindice could be a plugin to play an *Indexer* role as my term index does. However, due to Sindice's web access latency, my term index performs more effciently than Sindice because of its local disk index accesses.

Chapter 4

Source Selection

In order to retrieve relevant sources using the term index, any given conjunctive query needs to be converted into Boolean retrieval queries. Consider the mapping ontologies and domain ontologies, we need to first reformulate the original conjunctive query into a set of subgoals. Since each subgoal has a different number of sources that could contribute to solving this subgoal, we can use this heuristic to optimize the query planning. Therefore, this chapter starts with the discussion of conjunctive query reformulation. Since my query reformulation is based on the well known Peer Data Management System (PDMS) algorithm [28], I will first give a brief introduction to PDMS algorithm, in particular its query reformulatoin. Then, I will present three query optimizatoin algorithms for the source selection: the non-structure algorithm, the flat-structure algorithm and the tree-structure algorithm. For each of them, the correctness proof is given. Note, these algorithms do not consider the case of predicates as variables in queries because in the real world, the predicates of most queries (e.g. 79.44% in DBpedia and 99.48% in Semantic Web

Dog Food) are constants [2].

4.1   PDMS

The PDMS is a decentralized and extensible information integration architecture, in which any user can contribute new data, schema information, or even mappings between other peers' schemas. PDMS extends the two most well known information integration approaches: GAV (Section 2.4.1) and LAV (Section 2.4.2) replacing their single mediated schema with an interlinked collection of mappings between peers' individual schemas. Since in the Semantic Web we can and will have queries in any ontology (schemas), I need a mechanism that does not depend on a single mediated schema. Therefore PDMS's "any schema" approach meets such requirement. Hereinafter I refer to the PDMS algorithm simply as the PDMS.

PDMS takes as input a query, a set of views describing the sources and the maps, and computes a reformulation strictly in terms of the sources. Query answering in PDMS is polynomial time [28] if certain restrictions are imposed on the maps. If the maps are cyclic then PDMS becomes undecidable. However, acyclic maps are too restrictive. For example, equality maps (a common usecase) will always introduce cycles. Therefore, PDMS allows for equality maps, provided the relation in the head of a map does not appear in the body of any of other maps.

PDMS constructs a "rule-goal" tree: where goal nodes are labeled with atoms of the peer relations, and rule nodes are labeled with peer maps. The tree is constructed by expanding nodes using the maps between schemas. To expand a goal node the algorithm looks for a match in the maps for the label of that node. When a match

is found a child rule node is created which is labeled with the matched peer map. The rule node is then expanded as follows: if the matched peer map is a GAV-style mapping, a new child goal node is created for each sub goal of the peer map. If the matched peer map is a LAV-style mapping, child goal nodes are formed using the MiniCon algorithm [60].

The MiniCon algorithm identifies a minimal subset of views that is required to answer a query. In the process this algorithm creates a MiniCon (Minimal Construction) description for each set of query sub goals that cover a view. For a given MiniCon description of a goal node (w.r.t. its siblings and the matched peer map), the PDMS creates a rule node with the view of the MiniCon description and creates a child goal node with the head of the view. It also marks in the rule node all of the other sub goals that are covered by the peer map.

Each goal node is also expanded using the maps relating a data source to a peer. These maps are essentially LAV-style mappings with the actual stored relations on the left hand side of an inclusion description. The PDMS algorithm combines and interleaves the two types of reformulation techniques: one type of reformulation replaces a subgoal with a set of subgoals, while the other replaces a set of subgoals with a single sub goal.

The reformulation is a union of conjunctive queries over the stored relations. Each of these conjunctive queries represents one way of obtaining answers to the query from the relations stored at peers. Basically the rule goal tree is an *AND-OR* tree and the reformulations are all of the *AND-OR* traversals from root to leaf.

Apply the PDMS query reformulation into a Semantic Web Space, a conjunctive query is reformulated into a rule-goal tree using a set of GAV and LAV views
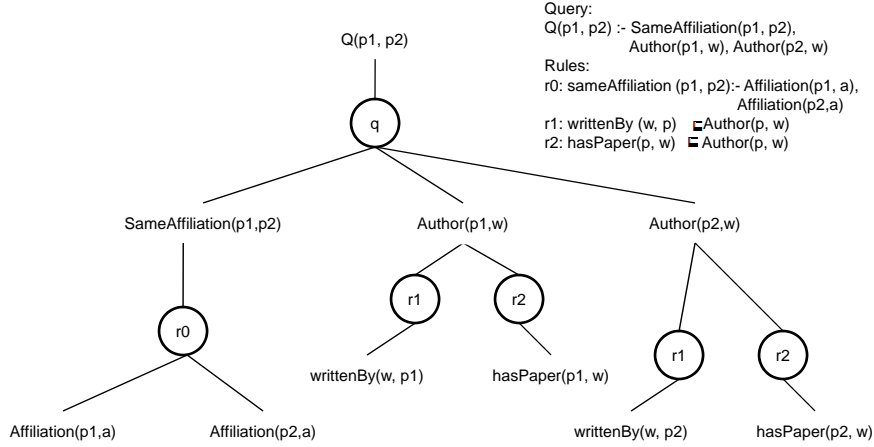
Figure 4.1: A PDMS-based query reformulation tree example

describing mapping ontologies and domain ontologies. In this tree, goal nodes are labeled with atoms of predicate, and rule nodes are labeled with ontology axioms defined in both domain ontologies and mapping ontologies. During the construction of the rule-goal tree, for each GAV mapping rule, an AND child goal node is created, and for each LAV mapping rule, an OR child goal node is created.

One query reformulation example is illustrated in Figure 4.1. Begin with the query Q, which asks for researchers who have worked at the same affiliation and also coauthored a paper. Q is expanded into three subgoals, each of which appears as a goal node of one AND rule node q. The *SameAffiliation* is involved into a GAV rule r_0, hence the reformulation expands the *SameAffiliation* goal nodes with the rule node r_0, whose children are two goal nodes of *Affiliation* relations (each specifying the affiliations that an individual researcher works at).

The *Author* relation is involved into two LAV rules r_1 and r_2, hence the reformulation expands $Author(p_1, w)$ and $Author(p_2, w)$ using two OR rule nodes and

both have two children goal nodes of *writtenBy* and *hasPaper*. As a result, Q is reformulated into a set of relations/subgoals: {*Affiliation, writtenBy, hasPaper*}.

4.2   The Non-structure Algorithm

Given a set of query subgoals (corresponding to all goal nodes over the PDMS rule-goal tree), the non-structure algorithm does an index lookup for each subgoal and loads all relevant sources into a *Reasoner* to solve the original query in form of Boolean query. Each subgoal about class membership is reformulated into a conjunction of *rdf:type* and the class. All others involving a predicate p are reformulated into a conjunction of p and any other constants in the query. For instance, consider a query with the following reformulated subgoals:

- $\langle x, rdf{:}type, u{:}Professor \rangle, \langle x, u{:}teaches, cs{:}proglang \rangle, \langle x, j{:}works\text{-}at, y \rangle$.

It is then translated into the following Boolean query:

- ($u{:}Professor$ AND $rdf{:}type$) OR ($u{:}teaches$ AND $cs{:}proglang$) OR ($j{:}works\text{-}at$).

Assuming the index is fresh, the set of documents returned by the term index are guaranteed to be the only documents that contain matching triples. Note, however, that some documents might have irrelevant triples. For example, a document with the triples $\langle a{:}john, u{:}teaches, math{:}calc \rangle$ and $\langle a{:}john, u{:}audits, cs{:}proglang \rangle$ will be returned for the example query. These irrelevant triples will not affect the overall correctness of the system, since they will not be used in any answers returned by the *Reasoner*. The only impact is the additional time to load an irrelevant document.

I expect that such documents will account for a small fraction of the documents selected, and that these false positives will be far fewer than the false positives returned by the original content-summary indexing scheme. Based on my real world Semantic Web data statistics using Sindice, this expectation can be verified. Over Sindice, my statistcs was executed by respectively issuing a query using Sindice's Boolean query function and Triple pattern query function. For example, the Boolean query "`http://sindice.com/hlisting/0.1/itemName` AND ipod" is to retrieve those documents that contain both terms of "`http://sindice.com/hlisting/0.1/itemName`" and "ipod". On the other hand, the Triple pattern query "$\star$ $\langle$ `http://sindice.com/hlisting/0.1/itemName`$\rangle$ 'ipod' " is to retrieve those documents that contain triples taking "`http://sindice.com/hlisting/0.1/itemName`" as a predicate and "ipod" as its object value. As a result, the ratio of the irrelevant documents selected by term index for a given query Q can be calculated as follows:

$$Irrelevance(Q) = \frac{Num\_of\_Res(Boolean(Q)) - Num\_of\_Res(Triple\_Pattern(Q))}{Num\_of\_Res(Boolean(Q))},$$

$$(4.1)$$

where $Boolean(Q)$ is the Boolean expression of Q and $Triple\_Pattern(Q)$ is the Triple pattern expression of Q.

Using the above formula, in my statistics of Table 4.1, I have issued 10 different queries and averaged their irrelevant document ratio. Finally, I found that the ratio of irrelevant data sources for a given query is 18.72%.

The non-structure source selection algorithm is described in Algorithm 1 (Figure 4.2), which is mainly to construct index compatible Boolean queries and then

Query Terms	# of Results of Boolean Query	# of Results of Triple_Pattern Query
`http://sindice.com/hlisting/0.1/itemName,` ipod	431	314
`http://sindice.com/hlisting/0.1/itemName,` nano	92	63
`http://sindice.com/hlisting/0.1/itemName,` shuffle	31	28
`http://data.semanticweb.org/person/` `james-hendler, http://data.semanticweb` `.org/ns/swc/ontology#holdsRole`	8	5
`http://xmlns.com/foaf/0.1/name`, Jeff Heflin	46	39
`http://xmlns.com/foaf/0.1/name`, Jim Hendler	2386	103
`http://data.semanticweb.org/person/` `james-hendler, http://swrc.ontoware.` `org/ontology#affiliation`	14	14
`http://www.w3.org/1999/02/22-rdf` `-syntax-ns#type, http://xmlns.com/` `foaf/0.1/Person`	67,322,318	62,847,724
`http://www.w3.org/2004/02/skos/core#broader,` `http://dbpedia.org/resource/Category` `:A-Class_articles`	761	740

Table 4.1: Statistics of irrelevant data sources

Algorithm 1 the non-structure source selection

---

function getSourceList(RQN: Reformulated Query Nodes)

        return: a list of sources

        inputs: *RQN*, the reformulated query subgoals;

1:     Let *sources* = ∅;

2:     **for each** $n \in RQN$ **do**

3:         **if** *n* **typeOf** *ClassMemberShip* **then**

4:            *qterm* = "(*rdf:type* AND"+*n.predicate*+")"

5:         **else**

6:            *qterm* = "("+*n.predicate*

7:         **if** *n.subject* **typeOf** *Constant* **then**

8:            *qterm* = *qterm*+" AND"+*n.subject*

9:         **if** *n* **typeOf** *owl:ObjectProperty* **then**

10:          **if** *n.object* **typeOf** *Constant* **then**

11:             *qterm* = *qterm*+" AND"+*n.object*

12:       **else**

13:          **if** *n* **typeOf** *DatatypeProperty* **then**

14:             *lterms* = *lit-terms*(*n*)

15:              **for each** $lterm \in lterms$ **do**

16:                *qterm* = *qterm*+" AND *lterm*"

17:       **if** !(*last_node*(n, RQN)) **then**

18:       *boolean_query.add*(*qterm*, ") OR")

19:       **else**

20:       *boolean_query.add*(*qterm*, ")")

21:    *sources* = index-lookup(*INDEX*, *boolean_query*)

22:    **return** *sources*

Figure 4.2: The non-structure algorithm

identify potentially relevant data sources using these queries. The core function is the Boolean query construction (Lines 2-20). For each subgoal, the algorithm first judges whether it is a class membership or not (Line 3). If yes saying $ns{:}Class(x)$, the Boolean query is then "$ns{:}Class$ AND $rdf{:}type$" because there is one special term $rdf{:}type$, which does not explicitly appear (Line 4). Otherwise, the Boolean query is generated using the predicate (Line 6) concatenated with keyword "AND" and available constants that might appear in either subject (Line 8) or object (Line 11). Note, if the object is a literal, I will extract the terms from the literal using a function $lit\text{-}terms()$ (Line 14) and concatenate each of them using Boolean "AND" (Line 16). After each subgoal's process, if it is not the last one, the generated Boolean query is concatenated using the keyword "OR" with the Boolean queries of other subgoals (Line 18). Otherwise, the keyword "OR" is not concatenated (Line 20). Finally, the whole Boolean query for the original conjunctive query is issued to the term index to find relevant sources (Line 21).

Theorem 1. *Given a Semantic Web Space SWS and a conjunctive query Q, the non-structure source selection algorithm is correct in that given a set of simple domain ontologies, OWLII map ontologies, and a term index over the data source documents within SWS, it will identify exactly the potentially relevant sources for Q.*

Proof. I will first prove the soundness, and then the completeness.

- Soundness:

  Assume the set of sources collected by the non-structure algorithm is $srcs(Q)$ and the set of answers to Q entailed by $srcs(Q)$ is Ans. Then we have

$srcs(Q) \subseteq SWS$. Thus, we can get $\forall \theta (\theta \in Ans \wedge Theory(srcs(Q)) \models Q\theta \rightarrow$
$Theory(SWS) \models Q\theta)$. Therefore, the non-structure algorithm is sound.

- Completeness:

Assume in the given SWS, C refers to the set of all classes, P refers to the
set of all properties, R refers to the set of all constant URIs, L refers to the
set of all literal terms, V refers to the set of all variables, and T refers to the
set of terms indexed by the term index I.

As shown in [61], the OBII-GNS algorithm is complete. Therefore, the sub-
goals identified by OBII-GNS are complete. Given a subgoal in form of $\langle x, p, y \rangle$
or $\langle x, rdf{:}type, c \rangle$ where $p \in P$, $c \in C$, $x/y \in R \sqcup V \sqcup L$ and a source function
s, assume a source $s(u)$ has a triple that unifies with $\langle x, p, y \rangle$ or $\langle x, rdf{:}type, c \rangle$
and a function $t_u = terms(s(u))$,where $terms(s(u))$ is defined to be the set of
terms contained in $s(u)$. One of the following cases has to happen during the
source selection by I:

- No constant constraints: $p/c \in t_u$, and $s(u) \in I(p/c)$.

- Either x or y is a constant constraint: for x/y, if $x/y \in R$, then $x/y \in t_u$,
  and $s(u) \in I((x/y) \wedge (p/c))$.

- Either x or y is a literal constraint: if $x/y \in L$, then $lit\text{-}terms(x/y) \in t_u$,
  and $s(u) \in I(lit\text{-}terms(x/y) \wedge (p/c))$.

Thus, a query reformulated by Algorithm 1 against I returns any $s(u)$ that is
directly relevant to the given subgoal. As a result, based on the complete set of

88

subgoals identified by OBII-GNS, the non-structure source selection algorithm is complete.

Therefore, the non-structure algorithm is correct.

□

However, the non-structure algorithm suffers from the following drawbacks:

- Because the term index only indicates if URIs or Literals are present in a document, specific answers to a subgoal of a given query cannot be calculated until the sources are physically accessed - an expensive operation given disk/network latency.

- Furthermore, even if there is no disk/network latency problem, in the real world, it is also very likely that the number of sources related to a subgoal could be so large that it is very difficult to load all of them into the reasoner to solve the queries in the real time. For example, based on the scalability evaluation using a real world data set in Section 6.2, given a query of $\langle ?x,\ \textit{affiliation},\ \textit{"lehigh-univ"}\rangle \wedge \langle ?x,\ \textit{maker},\ ?y\rangle$, the number of relevant sources of the given two subgoals are 5 and 3,485,607 respectively. As a result, the non-structure algorithm will load $3{,}485{,}607 + 5 = 3{,}485{,}612$ sources in total to solve this query. From the perspective of *Reasoner*, it will take around 7 hours to load this number of sources, which is clearly unsuitable for real-time queries. Also, many sound/complete reasoners do not scale to this size.

- Finally, each query reformulation subgoal is independently counted for the original query. Therefore, the non-structure algorithm does not consider the

89

structure relations among different query subgoals. Consequently, it cannot scale very well into the real world with large volume of semantic data because subgoals have different selectivities (Equation 4.2), which can be used to optimize the query answering through the constant constrait propagation among different subgoals. More details will be introduced in Sections 4.3 and 4.4.

4.3 The Flat-structure Algorithm

In order to overcome the drawbacks of the non-structure algorithm, I need to figure out some heuristics that can be applied to optimize the query answering. In this section, I first present my flat-structure algorithm in Section 4.3.1. Then, its correctness proof is given in Section 4.3.2.

4.3.1 Algorithm Description

Assume we have the following conjunctive query:

- $\langle ?p, rdf{:}type, a{:}Student\rangle \wedge \langle ?p, b{:}hasPhD, ?s\rangle \wedge \langle ?pap, b{:}has\text{-}author, \text{``james-hendler''}\rangle \wedge \langle ?p, b{:}has\text{-}paper, ?pap\rangle$.

In the real world, it is often difficult to judge the selectivities of the given QTPs because their selectivities are closely depending on features of the given semantic web space and QTPs themselves. However, generally speaking, we can still have the following discussion of the selectivities of the contained QTPs in the example query:

- $\langle ?pap, b{:}has\text{-}author, \text{``james-hendler''}\rangle$: this kind of QTPs is generally highly selective with constant constraints. It is because the constant constraint such

as "*james-hendler*" is so specific that the number of sources satisfying it is so few even if the number of matching triples may be large.

- $\langle ?p,\ b{:}hasPhD,\ ?s \rangle$ and $\langle ?p,\ b{:}has\text{-}paper,\ ?pap \rangle$: this kind of QTPs could be selective or low selective. Their selectivities depend on the selectivities of the predicates such as *b:hasPhD* or *b:has-paper* within the given semantic web space. For instance, if the given semantic web space is to describe the domain of general persons, the given QTPs are selective because the given predicates are relatively unique features of a person. On the other hand, if the given semantic web space is about all universities and their faculties and students, the given QTPs are low selective because the given predicates are not relatively unique attributes.

- $\langle ?p,\ rdf{:}type,\ a{:}Student \rangle$: this kind of QTPs could be also selective or low selective. Their selectivities depend on the selectivity of the *rdf:type* class such as *a:Student* within the given semantic web space. For instance, if *a:Student* is rarely used, then we can conclude that these QTPs are selective. Otherwise, we will say that they are low selective.

Based on the above QTP classification, I have gained one hint to take the selectivity of each subgoal (QTP) as the heuristic to plan the query execution. Formally, I define the source selectivity of a selection procedure *sproc* for a query subgoal α as the number of sources not selected divided by the total number of sources available:

$$Sel_{sproc}(\alpha) = \frac{|D| - |sproc(\alpha)|}{|D|} \tag{4.2}$$

According to the above formula, the source selectivity of one subgoal is inversely proportional to the number of sources that can contribute to solving this subgoal.

Furthermore, I also observe that the join selectivity of a pair of QTPs is often higher than the overall selectivity of these two QTPs treated independently. Consider two QTPs q_1 and q_2 from the same conjunctive query that share a variable x, in database parlance this situation is called a join condition and x is the join variable. I note that the number of sources required to answer the query are often less than $(qsources(q_1) \cup qsources(q_2))$. If we load the sources for q_1 first, we can find a set rs of variable bindings for q_1 from the triples contained in the sources. We can then apply each substitution $\theta \in rs$ to q_2 to generate a set of queries and get a set of sources for q_2 by doing index lookups for each $\bigcup_{\theta \in rs} qsources(q_2\theta)$. It should be clear that by adding an additional constant to each QTP (thus, belonging to highly selective QTPs), this join approach often has a higher source selectivity than naively applying *qsources* to each QTP in the query, although note that the join selectivity depends on which QTP is processed first. For example, for a given query $\langle ?x,\ \textit{affiliation},\ \textit{"lehigh-univ"}\rangle \wedge \langle ?x,\ \textit{maker},\ ?y\rangle$ with selectivities being 5 and 3,485,607 respectively, if we solve $\langle ?x,\ \textit{affiliation},\ \textit{"lehigh-univ"}\rangle$ first to obtain x's substitutions and then propagate them to $\langle ?x,\ \textit{maker},\ ?y\rangle$, the selectivity of $\langle ?x,\ \textit{maker},\ ?y\rangle$ could be significantly reduced from 3,485,607 to 114 for instance because of the constant constraint application. Finally, the total number of loaded sources is 119 =114+5 instead of 3,485,612 = 3,485,607+5.

Based on the above heuristic and observation, I propose the flat-structure algorithm, which takes a set of rewritings of the given query as its inputs to solve queries.
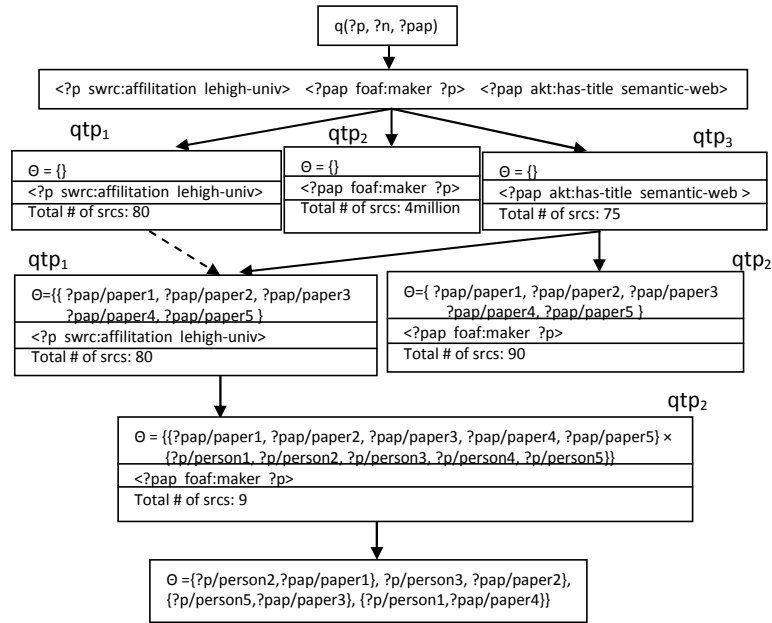
Figure 4.3: One example optimization tree of the flat-structure algorithm

Note, each query rewriting is a conjunctive query that is generated by using the domain and mapping ontologies and has a subset of the answers to the original query as its answers. The union of the answers of all query rewritings is equivalent to the set of answers of the original query. For each rewriting, the algorithm employs a source selection strategy that prioritizes selective subgoals of the query and uses the sources that are relevant to these subgoals to provide constraints that could make other subgoals more selective. In this way, the data sources will be incrementally collected and processed. Once sources are selected, we can load them into *Reasoner* to solve queries over these sources and their corresponding ontologies.

Figure 4.3 shows us an example. In this tree, each node consists of three fields: the available substitutions, the QTP node and the selected sources. This sample

query includes three QTPs: $\langle$ *?p, swrc:affiliation, lehigh-univ* $\rangle$ (qtp_1), $\langle$ *?pap,*
foaf:maker, ?p $\rangle$ (qtp_2) and $\langle$ *?pap, akt:has-title, "semantic-web"* $\rangle$ (qtp_3). Using
the term index, we might find that these QTPs' selectivities are 80, 4 million and
75 respectively. Since qtp_3 is the most selective, we load and evaluate its sources
first. Then, we apply the obtained substitutions for *?pap* into qtp_1 and qtp_2. After
this step, their selectivities are updated to be 80 and 90 respectively. Note, the
dashed line in this step means qtp_1 does not have join relation with qtp_3. Thus, its
selectivity is conserved down. Then, we start to evaluate the next most selective
QTP, qtp_1, and apply its substitutions for *?p* into qtp_2. After this step, we only
have qtp_2 left and evaluate it. Finally, the numbers of sources selected by each QTP
are 75 for qtp_3, 80 for qtp_1 and 9 for qtp_2. Therefore, the total number of sources
identified by the given query is $75+80+9 = 164$. Note, in this process, we keep track
of all sources that have been loaded, and do not repeat the loading of any source
while answering a particular query.

The pseudo code of the flat-structure algorithm is shown in Algorithm 2 (Fig-
ure 4.4). It takes a conjunctive query rewriting as its input. First, the algorithm
initializes the selectivity of each QTP contained in *sibs*, by executing a term index
lookup (Lines 5-6). At this step, if any bindings have been available, each is used as
a substitution to the corresponding *qtp* for its index lookup (Line 7). Here, *sibs* is
an array of sets of sources and indexed by QTPs. Then, it assigns the most selective
QTP to *on* and collects its relevant sources (Lines 8-10). Meanwhile, it removes *on*
from sibs (Line 11) and evaluates *on* to get its substitutions (Lines 12-13). Each sub-
stitution θ is then applied to *on*'s siblings to constrain their individual selectivities
(Line 7). Based on the new selectivity, the next most selective node is chosen and

---

Algorithm 2 the flat-structure algorithm

---

function OptimizeQuery(Query q) returns a list of sources

    inputs: *q*, a conjunctive query

1: Let *allsrcs* = ∅, query = true, sibs = a set of qtps in q, $rs = ∅$

2: *srcs[]* = array of sets of sources, indexed by qtps

3: **while** ($sibs ≠ ∅$)

4:      **for each** $qtp ∈ sibs$ **do**

5:        **if** ($rs = ∅$) **then**

6:           *srcs[qtp]*=index-lookup({*qtp*})

7:        **else** $srcs[qtp] = arg\ min_{θ∈rs}\ \text{index} - \text{lookup}(\{qtpθ\})$

8:        Let $on = \bigcup_{node∈sibs}(|srcs[node]|)$

9:        $allsrcs = allsrcs ∪ srcs[on]$

10:       Loader (*srcs[on]*, *KB*)

11:       *sibs* = sibs - {*on*}

12:       Let $query = query ∧ on$

13:       Let *rs* = Reasoner(*KB*, *query*)

14: **return** *allsrcs*

Figure 4.4: The flat-structure algorithm

the above process is repeated until all QTPs have been processed (Line 3). Finally, the sources collected by q are returned (Line 14).

The flat-structure algorithm can be combined with any query rewriting algorithm that produces a set of conjunctive subqueries. It can support expressive ontology languages such as OWL 2 QL. According to the official OWL 2 description [54], OWL 2 QL is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. In OWL 2 QL, conjunctive query answering can be implemented using conventional relational database systems in LOGSPACE with respect to the size of the data (assertions). Therefore, as long as the *Reasoner* is sound and complet for OWL 2 QL using a suitable reasoning technique, the flat-structure algorithm can fully support OWL 2 QL by rewriting the original query into a set of OWL 2 QL conjunctive subqueries. Here, each OWL 2 QL conjunctive query rewriting is an OWL 2 QL conjunctive query that is generated by using the domain and mapping OWL 2 QL ontologies and has a subset of the answers to the original query as its answers.

However, the flat-structure algorithm has the following problems:

- In order to avoid complications with inference impacting the number of sources for each QTP, it repeats the source selection procedure for each possible query rewrite. However, when there is significant heterogeneity in the ontologies, synonymous ontology expressions can lead to an explosion in the number of query rewrites. Processing a large number of rewrites can slow the system down, even if we cache the results of index lookups and are careful not to load the same source multiple times.

- The inability to use the full structure of query rewrites reduces the possible

source selectivity of the query process. Since source selection is independently executed for each query rewriting, selectivity is based only on local information, and does not account for the possibility that a subgoal that initially appears selective actually is not selective once all of its rewrites are taken into consideration.

4.3.2 Correctness Proof

As introduced in the last section, the flat-structure algorithm executes a constant constraint directed incremental source collection in order to answer given queries. The set of collected sources is actually a superset of the minimal set of sources that are necessary and sufficient to answer given queries. In order to prove the correctness of the flat-structure algorithm, I first define the minimal sources concept for a given query. Then, I define the concept of incremental source evaluation to describe my constant constraint directed source collection. Finally, based on my definitions and lemmas, the correctness of the flat-structure algorithm is described.

Definition 13. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a set of sources S, $Theory(S) = \bigcup_{d \in S} s(d) \cup \bigcup_{u \in U} o(u)$.*

Definition 14. *(Minimal Sources, MS) Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a conjunctive query Q, the $MS(Q)$ is a set of sources meeting the following conditions:*

- *$MS(Q) \subseteq s$,*

- *$Theory(MS(Q)) \models Q\theta$, $\forall \theta \in Ans\_SWS(Q)$. The $Ans\_SWS(Q)$ is defined in Definition 9,*

- $\forall srcs \subseteq MS(Q)$, $\exists \theta \in Ans\_SWS(Q)$, such that $Theory(srcs) \not\models Q\theta$. Note, the $MS(Q)$ is unique when there are no duplicate sources in SWS and no more than one way to infer the same answer.

According to Definition 14, given a conjunctive query $Q(X_0) = p_1(X_1) \wedge \ldots \wedge p_i(X_i) \wedge \ldots \wedge p_n(X_n)$, where X_i is a vector of variables and X_0 represents the variables for which bindings must be returned, if $\bigcup_{u \in U} o(u) = \emptyset$, $MS(Q) = \{src | \exists \theta \in Ans\_SWS(Q) \wedge \exists TP \in qtps(Q\theta) \wedge src \models TP\}$. Note, $\bigcup_{u \in U} o(u) = \emptyset$ means we do not need to consider the ontology inference.

Lemma 1. *(Minimal Sources for single QTP w.r.t. a query Q): Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a conjunctive query $Q(X_0) = p_1(X_1) \wedge \ldots \wedge p_i(X_i) \wedge \ldots \wedge p_n(X_n)$, where X_i is a vector of variables and X_0 represents the variables for which bindings must be returned, and assume $Ans\_SWS(Q) = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$, where θ_i is a substitution/set of bindings to X_i, if $\bigcup_{u \in U} o(u) = \emptyset$ and there are no duplicate sources, $MS_Q(p_i) = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i \theta_i)$.*

Proof. According to Definition 14, $MS(Q) = \{src | \exists \theta \in Ans\_SWS(Q) \wedge \exists TP \in qtps(Q\theta) \wedge src \models TP\}$. Then, for each QTP p_i in Q, we can construct a TP $= p_i \theta_i$. Then, $MS_Q(p_i) = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i \theta_i)$ because the following three conditions hold.

- $MS_Q(p_i) \subseteq s$: since $p_i \in atoms(Q)$, where $atoms(Q) = \{p_1(X_1), \ldots, p_i(X_i), \ldots, p_n(X_n)\}$ and p_i is defined in SWS, $MS_Q(p_i) \subseteq s$.

- $Theory(MS_Q(p_i)) \models p_i \theta_i$, where $\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)$: since TP $= p_i \theta_i$, we can construct $Theory(MS_Q(p_i)) = TP$.

- $\forall srcs \subseteq MS_Q(p_i), \exists \theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)$, such that $Theory(srcs) \not\models p_i\theta_i$:

  assume there is a set of sources $srcs \subseteq MS_Q(p_i)$ and $Theory(srcs) \models p_i\theta_i$.

  Then, we can find a set of sources $MS'(Q) = MS(Q) - MS_Q(p_i) + srcs$, which

  is a mininum set of sources of Q as well. Then, we have two minimum sets

  of sources for Q: $MS(Q)$ and $MS'(Q)$, when $\bigcup_{u \in U} o(u) = \emptyset$ and there are no

  duplicate sources, which is a contradiction to the third condition of Definition

  14. Thus, the given condition holds.

  □

Lemma 2. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a conjunctive query*

$Q(X_0) = p_1(X_1) \wedge \ldots \wedge p_i(X_i) \wedge \ldots \wedge p_n(X_n)$, *where X_i is a vector of variables and*

X_0 *represents the variables for which bindings must be returned, if $\bigcup_{u \in U} o(u) = \emptyset$,*

$MS(Q) = \bigcup_{p_i \in atoms(Q)} MS_Q(p_i)$, *where $atoms(Q) = \{p_1(X_1), \ldots, p_i(X_i), \ldots, p_n(X_n)\}$.*

Proof. Since $\bigcup_{u \in U} o(u) = \emptyset$, we do not need to consider the ontology inference.

According to Definition 14, $MS(Q) = \{src | \exists \theta \in Ans\_SWS(Q) \wedge \exists TP \in qtps(Q\theta) \wedge$

$src \models TP\}$. Assume $Ans\_SWS(Q) = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$. We can get

$MS(Q) = \bigcup_{1 \leq i \leq n} \{src | \exists \theta_i \in \theta \wedge \theta \in Ans\_SWS(Q) \wedge \exists TP_i \in qtps(Q\theta_i) \wedge src \models$

$TP_i\} = \bigcup_{1 \leq i \leq n} \{src | \exists \theta_i \in \theta \wedge \theta \in Ans\_SWS(Q) \wedge \exists TP_i \in p_i\theta_i \wedge src \models TP_i\} =$

$\bigcup_{1 \leq i \leq n} \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} \{src | \exists TP_i \in p_i\theta_i \wedge src \models TP_i\} = \bigcup_{1 \leq i \leq n} \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)}$

$qsources(p_i\theta_i)$.

According to Lemma 1, $MS_Q(p_i) = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i\theta_i)$.

Thus, we can get $MS(Q) = \bigcup_{1 \leq i \leq n} MS_Q(p_i) = \bigcup_{p_i \in atoms(Q)} MS_Q(p_i)$

□

Till now, I have defined the minimal sources concept $MS(Q)$ and $MS_Q(p_i)$ for a

given query and its each QTP respectively in Definition 14 and Lemma 1. Since the flat-structure algorithm executes an incremental source collection, in the following part I will define the concept of incremental source evaluation to describe my constant constraint directed source collection in Definition 16. Then, the correctness proof of the incremental source evaluation is given in Lemma 4.

Definition 15. *(Single QTP Source Evaluation) Given a QTP qtp and an answer set $Ans = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$, its source collection evaluation is defined to be $qtp\_sources(qtp, Ans) = \bigcup_{\theta \in Ans} qsources(qtp\theta)$.*

Definition 16. *(Incremental source evaluation) Given a conjunctive query Q, its incremental source evaluation is defined as follows:*

(1) Initialize a set of QTPs saying $S = \{q | q \in Q\}$ and an intermediate conjunctive query $Q' = \emptyset$.

(2) From S, pick a QTP q.

(3) Construct an intermediate conjunctive query $Q' = Q' \wedge q$.

(4) Compuate $Ans_{Q'} = QE(Q', \bigcup_{q \in atoms(Q')} qsources(q))$, where QE is the query engine function that is defined in Section 3.2.

(5) Remove q from S.

(6) Repeat steps (2) - (5) until $S = \emptyset$.

Lemma 3. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a conjunctive query $Q(X_0) = p_1(X_1) \wedge \ldots \wedge p_i(X_i) \wedge \ldots \wedge p_n(X_n)$, where X_i is a vector of variables and X_0 represents the variables for which bindings must be returned, if $\bigcup_{u \in U} o(u) = \emptyset$, $MS_Q(p_i) \subseteq qtp\_sources(p_i, Ans)$, where Ans is an answer set to a subquery Q' of Q: $atoms(Q') \subseteq atoms(Q)$. Furthermore, the single QTP source evalulation of p_i is complete w.r.t. $MS(Q)$.*

Proof. Assume $Ans\_SWS(Q) = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$.

According to Lemma 1, we have $MS_Q(p_i) = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i \theta_i)$

According to Definition 15, we have $qtp\_sources(p_i, Ans) = \bigcup_{\theta \in Ans} qsources(p_i \theta)$.

According to Definition 16, we have $\pi_{var(p_i)} Ans\_SWS(Q) \subseteq \pi_{var(p_i)} Ans = \pi_{var(p_i)}$ $Ans\_SWS(Q')$, where $var(p_i)$ is the variables contained in the QTP p_i and Q' is a subquery of Q.

Thus, we have $\bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i \theta_i) \subseteq \bigcup_{\theta \in Ans} qsources(p_i \theta)$.

Therefore, $MS_Q(p_i) \subseteq qtp\_sources(p_i, Ans)$.

According to Definition 14, since $MS_Q(p_i)$ is the minimum set of sources for the QTP p_i within the given Q, we can further conclude that the single QTP source evaluation of p_i is complete w.r.t. $MS(Q)$.

Therefore, Lemma 3 holds. $\qquad\square$

Lemma 4. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$ and a conjunctive query Q, if $\bigcup_{u \in U} o(u) = \emptyset$, Q's incremental source evaluation always returns a set of sources that is a superset of $MS(Q)$.*

Proof. I will prove this lemma by two steps: first, I use the mathematical induction to prove Q's incremental source evaluation always returns a set of sources that is a superset of $\bigcup_{p_i \in Q} MS_Q(p_i)$. Then, based on Lemma 2 $MS(Q) = \bigcup_{p_i \in Q} MS_Q(p_i)$, we can prove that this lemma holds.

Assume $Ans\_SWS(Q) = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$.

At the beginning, assume we start with a QTP in Q saying p_b, $(1 \leq b \leq n)$. The available answer set $Ans_b = \emptyset$. Then, the mathematical induction proof is as follows:

101

- Base case:

  According to Definition 15, $qtp\_sources(p_b, Ans_b) = \bigcup_{\theta \in Ans_b} qsources(p_b\theta) = qsources(p_b)$.

  According to Lemma 1, $MS_Q(p_b) = \bigcup_{\theta_b \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_b\theta_b)$.

  Thus, according to Lemma 3, $MS_Q(p_b) \subseteq qtp\_sources(p_b, Ans_b)$.

- Recursive case:

  Assume $MS_Q(p_b) \cup MS_Q(p_{b+1}) \cup \ldots \cup MS_Q(p_{b+k}) \subseteq qtp\_sources(p_b, Ans_b) \cup qtp\_sources(p_{b+1}, Ans_{b+1}) \cup \ldots \cup qtp\_sources(p_{b+k}, Ans_{b+k})$, where $1 \leq (b, \ldots, b+k) \leq n$.

  Then, given Ans_{b+k+1}, we need to prove:

  $MS_Q(p_b) \cup MS_Q(p_{b+1}) \cup \ldots \cup MS_Q(p_{b+k}) \cup MS_Q(p_{b+k+1}) \subseteq qtp\_sources(p_b, Ans_b) \cup qtp\_sources(p_{b+1}, Ans_{b+1}) \cup \ldots \cup qtp\_sources(p_{b+k}, Ans_{b+k}) \cup qtp\_sources(p_{b+k+1}, Ans_{b+k+1})$, where $1 \leq b + k + 1 \leq n$.

  According to Definition 15, $qtp\_sources(p_{b+k+1}, Ans_{b+k+1}) = \bigcup_{\theta \in Ans_{b+k+1}} qsources(p_{b+k+1}\theta)$.

  Then, according to Lemma 3, we have $MS_Q(p_{b+k+1}) \subseteq qtp\_sources(p_{b+k+1}, Ans_{b+k+1})$.

  Thus, $MS_Q(p_b) \cup MS_Q(p_{b+1}) \cup \ldots \cup MS_Q(p_{b+k}) \cup MS_Q(p_{b+k+1}) \subseteq qtp\_sources(p_b, Ans_b) \cup qtp\_sources(p_{b+1}, Ans_{b+1}) \cup \ldots \cup qtp\_sources(p_{b+k}, Ans_{b+k}) \cup qtp\_sources(p_{b+k+1}, Ans_{b+k+1})$.

  Then, the recursive case holds.

Therefore, Q's incremental source evaluation always returns a set of sources that is a superset of $\bigcup_{p_i \in Q} MS_Q(p_i)$.

According to Lemma 2, $\bigcup_{p_i \in Q} MS_Q(p_i) = MS(Q)$.

Then, we can conclude that Q's incremental source evaluation always returns a set of sources that is a superset of $MS(Q)$. $\qquad\qquad\qquad\qquad\qquad\square$

Before the correctness proof of flat-structure algorithm (Theorem 2), I first give the definition of correctness of a source collection algorithm in Definition 17. Additionally, the correctness of flat-structure algorithm depends on the correctness of query rewriting generation, I also give the definition of correct query rewritings in Definition 18.

Definition 17. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$, a conjunctive query Q and a source selection function S, S is correct iff $SWS \models Q\theta$, where $\theta \in Ans\_SWS(Q)$, then $S(Q) \subseteq SWS \wedge S(Q) \models Q\theta$.*

Definition 18. *Given a Semantic Web Space $SWS = \langle \mathcal{U}, o, s \rangle$, a conjunctive query Q and a set of Q's conjunctive query rewritings $Q_r = \{Q_1, \ldots, Q_i, \ldots, Q_n\}$, where Q_i is a conjunctive query rewriting of Q, Q_r is correct iff $Ans\_SWS(Q) = \bigcup_{Q_i \in Q_r} Ans\_SWS(Q_i)$.*

Theorem 2. *Given a Semantic Web Space SWS and a conjunctive query Q, the flat-structure algorithm is sound and complete for any set of correct, conjunctive rewritings of Q.*

Proof. I will first prove the soundness, and then the completeness.

- Soundness:

Assume the set of sources collected by the flat-structure algorithm is $srcs(Q)$ and the set of answers to Q entailed by $srcs(Q)$ is *Ans*. Then we have $srcs(Q) \subseteq SWS$. Thus, we can get $\forall \theta(\theta \in Ans \wedge Theory(srcs(Q)) \models Q\theta \rightarrow Theory(SWS) \models Q\theta)$. Therefore, the flat-structure algorithm is sound.

- Completeness:

  According to the description of the flat-structure algorithm in Section 4.3.1, assume $Q_r = \{Q_1, \ldots, Q_i, \ldots, Q_n\}$ is a correct set of conjunctive query rewritings of Q and the source selection of the flat-structure algorithm is correct.

  For each conjunctive query rewriting Q_i, we execute an incremental source evaluation, whose collected sources is always a superset of $MS(Q_i)$ according to Lemma 4.

  Then the set denoted as *srcs* of sources collected by the flat-structure algorithm for Q is:

  $srcs(Q) = \bigcup_{Q_i \in Q_r} srcs(Q_i) = \bigcup_{Q_i \in Q_r} \bigcup_{q \in atoms(Q_i)} qtp\_sources(q, Ans)$, where *Ans* is an intermediate answer set that is used to evaluate the sources of each QTP q in Q_i.

  Since $\bigcup_{q \in atoms(Q_i)} qtp\_sources(q, Ans)$ is complete according to Lemma 3, then $\bigcup_{Q_i \in Q_r} \bigcup_{q \in atoms(Q_i)} qtp\_sources(q, Ans)$ is also complete according to Definition 18 because Q_r is correct. Thus, $srcs(Q) = \bigcup_{Q_i \in Q_r} srcs(Q_i)$ is also complete.

  Then, we have $\forall \theta(\theta \in Ans\_SWS(Q) \wedge SWS \models Q\theta \rightarrow Theory(srcs(Q)) \models Q\theta)$.

  According to Definition 17, the flat-structure algorithm is complete.

Therefore, the flat-structure algorithm is sound and complete.

□

4.4 The Tree-structure Algorithm

In order to overcome the drawbacks of the flat-structure algorithm, I further propose the tree-structure algorithm. In this section, I first present my tree-structure algorithm in Section 4.4.1. Then, its correctness proof is given in Section 4.4.2.

4.4.1 Algorithm Description

Given a conjunctive query, the tree-structure algorithm first reformulates this query into a rule-goal tree (Section 4.1) and each goal node is associated with its selectivity (Section 4.3). Beginning with this tree, the algorithm chooses the most selective QTP leaf goal node as the starting point from a frontier node set, which consists of the lowest level of unprocessed goal nodes in the rule-goal tree. Then, for each AND mapping rule containing the chosen QTP, the algorithm starts from the most selective QTP and greedily collects sources of the QTPs contained in this rule by applying available constant constraints from the chosen QTP each time into its join sibling QTPs until all nodes of this AND rule has been processed. As a result, we can reduce the number of selected sources and gain higher selectivity. Then, the selected sources will be broadcast upward to the corresponding parent QTP goal node of the current rule node, added into the source list of this goal node and the selectivity of this goal node is updated. After this step, the frontier node set is renewed. With the new selectivities, the algorithm then selects the next most

Q(?p, ?n, ?pap)

(q)

(7)
<?p, akt:full-name, ?n>: $4\text{x}10^6$/4/10

(5)
<?pap, akt:has-author, ?p>: $3\text{x}10^6$/20/20

(4)
<?pap, swat:makerAffiliation, "lehigh-univ">: 20/20/105

(r2)

(6)
<?p, foaf:name, ?n>: $3\text{x}10^6$/6/6

(r0)

(2)
<?m, swrc:affiliation, "lehigh-univ">: 20/20/60

(3)
<?pap, foaf:maker, ?m>: $4\text{x}10^6$/25/25

(r1)

(1)
<?m, akt:has-affiliation, "lehigh-univ">: 40/40/40
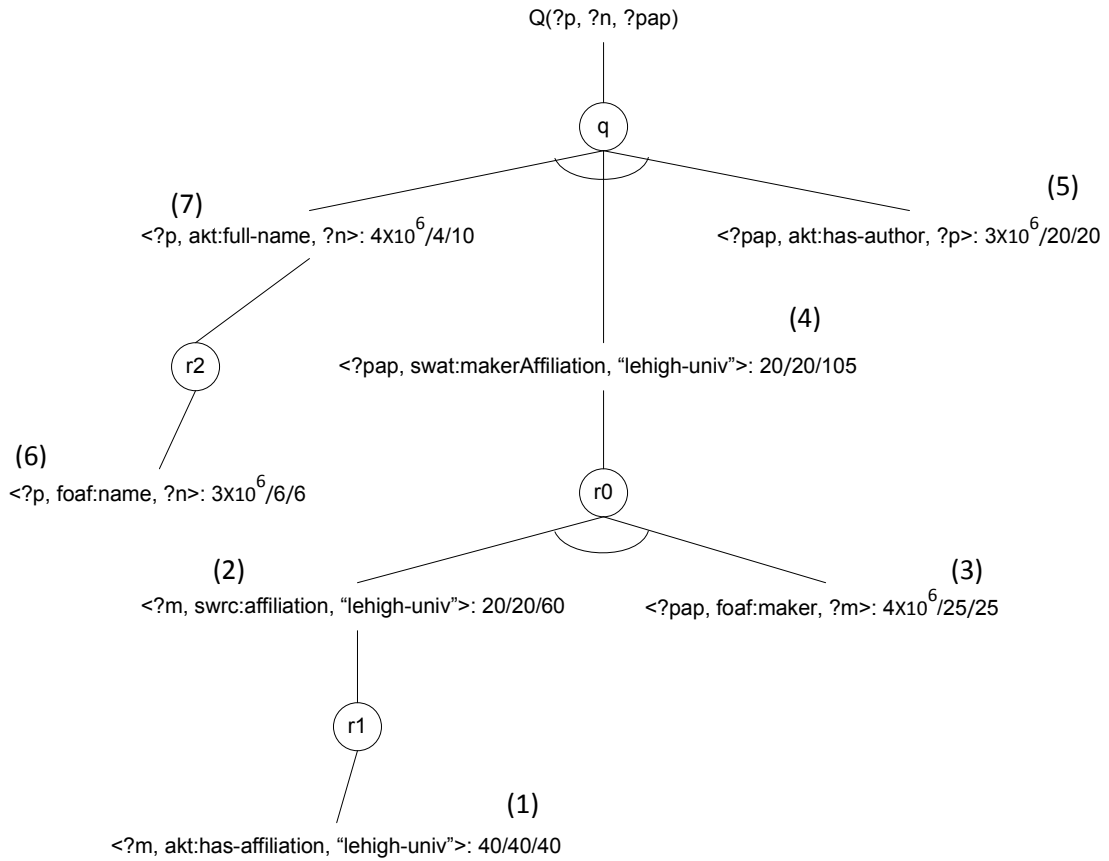
Figure 4.5: Query resolution of one sample query with notations in form of initial-cost/local-optimal-cost/total-cost

selective frontier node to start source collection again. This process is iteratively executed until all QTP goal nodes have been evaluated. Finally, the collected sources are loaded into *Reasoner* to answer the original query. Note, since every collected source has been loaded into the reasoner in process of its being collected, the original query can be directly answered.

Figure 4.5 gives an example to introduce how the tree-structure algorithm works. Consider the rule-goal tree for the given query Q, which asks for the publications

q(?p, ?n, ?pap)

<?p akt:full-name ?n>   <?pap swat:makerAffiliation "lehigh-univ">   <?pap akt:has-author ?p>

Θ={ }
<?p akt:full-name ?n>
<?p foaf:name ?n>
Total # of srcs:7million

Θ={ }
<?pap akt:has-author ?p >
Total # of srcs: 3 million

Θ={ }
<?pap swat:makerAffiliation lehigh-univ>
<?m swrc:affilitation lehigh-univ>
<?m akt:has-affilitation lehigh-univ>
<?pap foaf:maker ?m>
Total # of srcs: 105

Θ ={?pap/paper1, ?pap/paper2, ?pap/paper3, ?pap/paper4}
<?p akt:full-name ?n>
<?p foaf:name ?n>
Total # of srcs: 7 million

Θ = { ?pap/paper1, ?pap/paper2, ?pap/paper3, ?pap/paper4}
<?pap akt:has-author ?p >
Total # of srcs: 20

<?pap swat:makerAffiliation lehigh-univ>

Θ = {}
<?m swrc:affilitation lehigh-univ>
<?m akt:has-affilitation lehigh-univ>
Total # of srcs: 60

Θ = {}
<?pap foaf:maker ?m>
Total # of srcs: 4million

Θ ={{?p/person2,?pap/paper1}, {?p/person3, ?pap/paper2 }, {?p/person5,?pap/paper3 }}
<?p akt:full-name ?n>
<?p foaf:name ?n >
Total # of srcs: 10

Θ={?m/maker1, ?m/maker2, ?m/maker3,
?m/maker4, ?m/maker5}
<?pap foaf:maker ?m>
Total # of srcs: 25

Θ = {{?p/person2,?pap/paper1, ?n/name1}, {?p/person3, ?pap/paper2, ?n/name2},
{?p/person5,?pap/paper3, ?n/name3}}

(a)   (b)

Figure 4.6: AND-optimization. At each level of the tree a QTP is chosen greedily, its sources loaded and queried, and the answers applied to sibling QTPs

107

affiliated with Lehigh University (*"lehigh-univ"*), complete with the *id*s and *name*s of their authors. In the diagram, each goal node has three associated costs: the *initial-cost* is the number of sources relevant to that goal if we do not consider any axioms, the *local-optimal-cost* is the number of relevant sources after applying available constant constraints and the *total-cost* is the number of sources after applying available constant constraints and collecting sources from the descendants. Additionally, the order in which we process goal nodes is indicated by the parenthesized numbers.

The first step is to use the term index to initialize the tree with source selectivity information, represented by initial costs next to each goal node. The algorithm starts with the QTP leaf node that selects the fewest sources: $\langle$*?m, akt:has-affiliation, "lehigh-univ"*$\rangle$ (labeled with (1)). Since this is an *OR* node, the algorithm simply propagates its sources up to its parent goal. Thus, the total-cost for $\langle$*?m,swrc:affiliation, "lehigh-univ"*$\rangle$ is updated to 60 (40 sources from its child plus 20 sources of its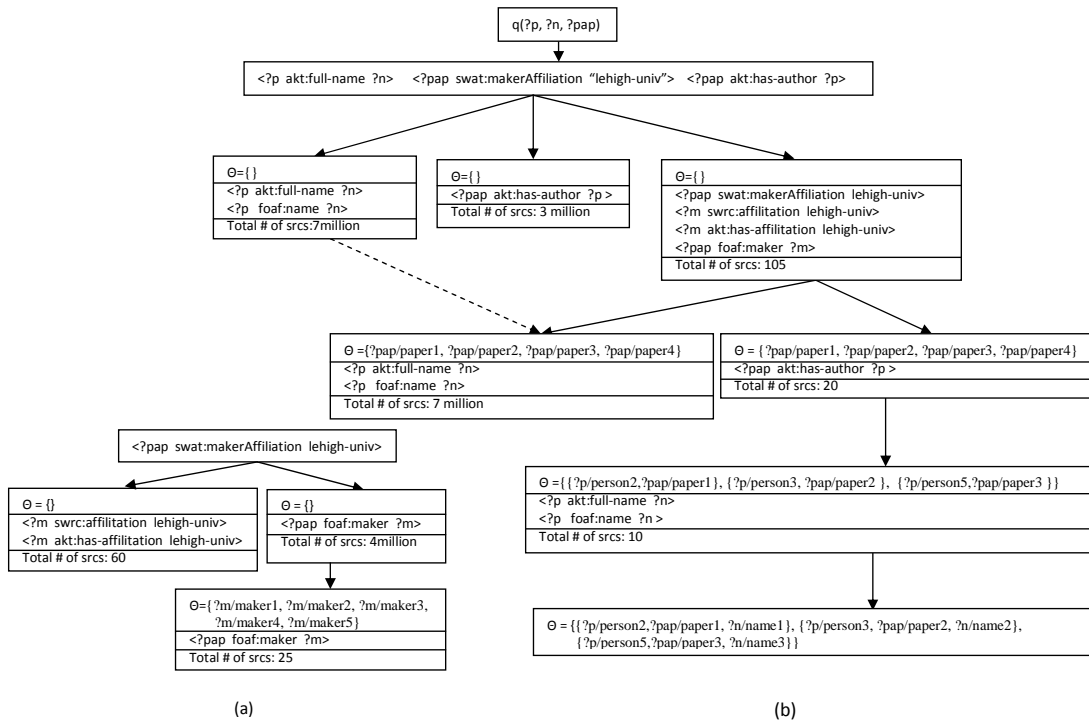elf; for simplicity of exposition I am assuming that the sets of sources are disjoint, but this is not a requirement for the algorithm). Since all children of $\langle$*?m,swrc:affiliation, "lehigh-univ"*$\rangle$ have been processed, it joins the leaf nodes as a candidate for processing, and since it's total cost is 60, which is less than the initial costs of all other candidates, it is the next node to be processed. Since it is a child of $r0$, an *AND* rule node (indicated by the arc), we can use it to constrain its sibling *foaf:maker* as shown in Figure 4.6(a). First, the algorithm loads all sources associated with the goal node and issues the goal as a query for these sources. This query results in the substitutions for *?m*: $\{$*?m/maker1, ?m/maker2, ...*$\}$. Each of these substitutions is then applied to $\langle$*?pap,foaf:maker,?m*$\rangle$, an index lookup is performed

for each resulting QTP, and the total set of sources (in this case 25 of them) is used to update the total cost of this node in Figure 4.5, step (3). In step (4), the total cost of these nodes (60+25=85) is propagated to their parent *swat:makerAffiliation*, and is added to its initial cost (20), resulting in a total cost of 105. Since this node now has the best selectivity and is the child of an *AND* rule node (the original query), another *AND* optimization is performed shown in Figure 4.6(b). As shown, once this node is selected, there are two siblings to choose from. However, before we can determine the cost of these nodes, we must repeat the tree process on the subtrees rooted at these nodes, thus the number of sources for ⟨*?p,akt:full-name,?n*⟩ is 7 million, the sum of its sources and the sources of its child ⟨*?p,foaf:name,?n*⟩. I apply the substitutions from *swat:makerAffiliation* to each sibling, resulting in the number of sources of *akt:has-author* being reduced to 20 (updating its *local-optimal-cost* in Figure 4.5), but not changing the sources of *akt:full-name*. In step (5) of Figure 4.5, the algorithm selects *akt:has-author*, loads its sources, issues a combined query with the previous goal, and get a new set of substitutions. These substitutions are then applied to the subtree of *akt:full-name*, changing the local-optimal-costs of *foaf:name* and *akt:full-name* to 6 and 4, respectively, and changing the total-cost of *akt:full-name* to 10. As a result, the total number of collected sources for the given conjunctive query is $105 + 20 + 10 = 135$, compared to over 11 million if no optimization was done. Once all sources are loaded, we can ask the original query of the *Reasoner* in order to get a final set of substitutions.

Note, a comparison between the tree-structure algorithm and the flat-structure algorithm (Algorithm 2) is given at the end of this section.

The pseudo code for tree-structure algorithm is shown in Figure 4.7. Algorithm

---

Algorithm 3 source selection for tree-structure query optimization

---

function getSourceList(*rtree, rs*) **returns** a list of sources
    inputs: *rtree*, a given rule goal tree *rtree*
          rs, a list of substitutions
1: Let *frontier* = leaf nodes of *rtree,*
    srcs[] = array of sets of sources, indexed by goal nodes
2: **for each** goal node *n* in *rtree* **do**
3:     **for each** $\theta \in rs$ **do**
4:        *srcs[n]* = qsources($n\theta$)
5: **do**
6:     Let $n = arg\ min_{node\ \in\ frontier}\ (|srcs[node]|)$, *p* = getParent(*n*)
7:     **if** *n* is a child of an AND rule node *r* **then**
8:        $srcs[p] = srcs[p] \cup \text{OptimizeANDNode}(n, siblings\ of\ n, srcs)$
9:     **else**
10:       $srcs[p] = srcs[p] \cup srcs[n]$
11:     remove *n* from *frontier*
12:     **if** *p* has no descendants on *frontier* **then**
13:       add *p* to *frontier*
14: **while** ($frontier \neq \{rtree.root\}$)
15: **return** *srcs[rtree.root]*

Algorithm 4 node optimization

---

function OptimizeANDNode(*on*, *sibs, srcs*) **returns** a list of sources
    inputs: *on*, a given goal node in the rule-goal tree
         sibs, a set of *on*'s sibling nodes
         srcs, an array of sets of sources, indexed by goal nodes
1: Let *allsrcs* = $\emptyset$, *query* = true
2: $allsrcs = allsrcs \cup srcs[on]$
3: load(*srcs[on]*, *KB*)
4: **do**
5:     Let *query* = *query* $\wedge$ *on*
6:     Let *rs* = Reasoner (*KB*, *query*)
7:     **for each** $qtp \in sibs$ **do**
8:        *srcs[qtp]* = getSourceList(subtree rooted at *qtp*, *rs*)
9:     Let $on = arg\ min_{t\ \in\ sibs\ that\ join\ with\ query}\ (srcs[t])$
10:     Remove *on* from *sibs*
11:     $allsrcs = allsrcs \cup srcs[on]$
12:     Loader(*srcs[on]*, *KB*)
13: **while** ($sibs \neq \emptyset$)
14: **return** *allsrcs*

Figure 4.7: The tree-structure algorithm

4.4. THE TREE-STRUCTURE ALGORITHM

3 processes a rule-goal tree, where the parameter rs, which provides a set of substitutions, is $\emptyset$ when first called, but instantiated in recursive calls. I use *frontier* to maintain a set of deepest, unprocessed goal nodes in the rule-goal tree; this is initialized to be the set of leaf nodes. In Lines 2-4, I use the term index to determine the initial selectivity of all goal nodes in the rule-goal tree. Then, the most selective node n is chosen from the *frontier* (Line 6). I check if n is a child of an AND rule, and if so Algorithm 4 is called to collect sources by using the greedy strategy (Lines 7-8). If the rule is an OR mapping, the sources from the rule children are directly broadcast upward to the rule parent goal node p (Lines 9-10). Since this completes the processing of n, I remove it from our *frontier* node set (Line 11) and if p currently has no descendants in $frontier$, I add p to the *frontier* (Lines 12-13). When the *frontier* contains only the root of the given rule-goal tree, the *while* loop terminates and the source collection ends (Line 14). Finally, all collected sources are returned (Line 15).

Algorithm 4 optimizes an AND node, given a most selective goal node on, its siblings *sibs*, and an array of the sources for each node in the tree (the latter is used as an output parameter to update the log of sources found for each node). It starts by loading on's sources into the knowledge base KB. Then, it evaluates on by asking the reasoner to get the substitutions of the variables contained in on (Lines 5-6). These substitutions are then applied to on's siblings to enhance their individual selectivity (Lines 7-8). Note the recursive call to $getSourceList()$ in line 8; this ensures that any new constraints specified by rs are effectively applied to the subtree rooted at each sibling. Based on the new selectivity estimations, the algorithm chooses the next most selective node that shares a join variable with the

partial query to be the next *on* (Line 9). Then the algorithm removes *on* from *sibs*, adds its sources to the sources retrieved so far, and loads any newly selected sources (Lines 10-12). In the next iteration, *on* is conjuncted with the partial query *query*, the *reasoner* is queried, and the substitutions applied again to the siblings. This process is repeated until all sibling nodes of the initial given goal node are processed (Line 13). Finally, the sources collected by the current AND mapping rule are returned (Line 14).

Compared to Algorithm 4, the flat-structure algorithm (Algorithm 2 in Figure 4.4) essentially executes a variation of Algorithm 4 for every conjunctive query rewrite. The main difference is that in Lines 7-8, Algorithm 4 iteratively calls Algorithm 3 to execute a source collection by passing the results among different query rewrites. While, in Lines 5-7 of Algorithm 2, the flat-structure algorithm only collects sources within the current processing conjunctive query rewrite and does not pass the results onto other query rewrites, which limits its ability to use the full structure of query rewrites as mentioned in Section 4.3.

Even though the tree-structure algorithm can solve the problems of the flat-structure and non-structure algorithms, it still has the following deficiencies:

- Since finite reformulation trees cannot express rewrites of a query whose re-formulation involves cyclic rules, completeness is only guaranteed for acyclic OWLII axioms.

- This approach is incomplete in the presence of equality reasoning (*owl*:*sameAs*), that is, information about which URIs denote the same objects. Essentially, the equality reasoning is a special case of cyclic axioms because *owl*:*sameAs* is a ubiquitous transitive property.

4.4.2 Correctness Proof

Similar to the flat-structure algorithm, the tree-structure algorithm also executes a constant constraint directed incremental source collection in order to answer given queries. However, it has gained better source selectivity by using the full structure of query rewrites instead of a list of query rewritings used by the flat-structure algorithm. In this section, I will prove the correctness of the tree-structure algorithm based on my previous definitions, lemmas and theorems in Section 4.3.2. First, I will prove that the tree-structure algorithm selects a subset of the sources collected by the flat-structure algorithm in Theorem 3. Then, the correctness proof of the tree-structure algorithm is described in Theorem 4.

Theorem 3. *Given a Semantic Web Space SWS and a conjunctive query Q, the tree-structure algorithm selects a subset of the sources collected by the flat-structure algorithm.*

Proof. Assume an intermediate answer set of solving Q by the tree-structure algorithm is $Ans = \{\theta | \theta = \{\theta_1, \ldots, \theta_i, \ldots, \theta_n\}\}$ and a correct set of Q's conjunctive query rewritings is $Q_r = \{Q_1, \ldots, Q_j, \ldots, Q_n\}$ according to Definition 18.

Let the set of sources collected by the rule-goal tree algorithm be S_1 and the set of sources collected by the flat-structure algorithm be S_2. Now, I will prove $S_1 \subseteq S_2$.

For each $s \in S_1$, we have $\exists p_i(p_i \in atoms(Q) \land s \in qtp\_sources(p_i, Ans) \land (qtp\_sources(p_i, Ans) = qsources(p_i\theta_i)))$, where $\theta_i \in \theta$ and $\theta \in Ans$. Here θ_i has two possibilities:

- $\theta_i = \emptyset$, which means the available constant constraints of p_i is from itself.

- $\theta_i \neq \emptyset$, which means the available constant constraints of p_i is from the evaluation of other join QTPs with p_i.

Now I will prove $s \in S_2$ from p_i's available constant constraints within the rule-goal tree algorithm.

- $\theta_i = \emptyset$: this means the available constant constraints of p_i is from itself. Then,

  $\exists Q_j((Q_j \in Q_r) \wedge p_i \in atoms(Q_j))$.

  Since $s \in S_1$, $S_1 = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i\theta_i)$ and $\theta_i = \emptyset$, $s \in qsources(p_i)$.

  Since $S_2 = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q_j)} qsources(p_i\theta_i)$ and $\theta_i = \emptyset$, $S_2 = qsources(p_i)$.

  Thus, $s \in S_2$.

- $\theta_i \neq \emptyset$: this means the available constant constraints of p_i is from the evaluation of other join QTPs with p_i saying $p_k \in atoms(Q)$. Then, we have

  $Q'_r = \{Q_j | (Q_j \in Q_r) \wedge (p_i \wedge p_k \in atoms(Q_j))\}$.

  Assume Ans_{Q_j} is an intermediate answer set of solving Q_j by the flat-structure algorithm, $vars(p_i)$ is the set of variables in p_i and the rule-goal tree generated by the tree-structure algorithm is T.

  Since $\forall p(p \in atoms(Q_j) \wedge Q_j \in Q'_r \to p \in atoms(T))$, $\pi_{vars(p_i)} Ans \subseteq \bigcup_{Q_j \in Q'_r} \pi_{vars(p_i)} Ans_{Q_j}$.

  Then $qtp\_sources(p_i, Ans) \subseteq qtp\_sources(p_i, \bigcup_{Q_j \in Q'_r} Ans_{Q_j})$.

  Since $s \in qtp\_sources(p_i, Ans)$ and $S_2 = qtp\_sources(p_i, \bigcup_{Q_j \in Q'_r} Ans_{Q_j})$, $s \in S_2$.

Thus, we have $S_1 \subseteq S_2$. Then, the theorem holds. $\qquad\qquad\square$

Theorem 4. *Given a conjunctive query Q and an acyclic semantic web space SWS, the tree-structure algorithm is sound and complete for ontologies that can be represented in AND/OR relations.*

Proof. I will first prove the soundness, and then the completeness.

- Soundness:

  Assume the set of sources collected by the tree-structure algorithm is $srcs(Q)$ and the set of answers to Q entailed by $srcs(Q)$ is Ans. Then we have $srcs(Q) \subseteq SWS$. Thus, we can get $\forall \theta (\theta \in Ans \wedge Theory(srcs(Q)) \models Q\theta \rightarrow Theory(SWS) \models Q\theta)$. Therefore, the tree-structure algorithm is sound.

- Completeness:

  I will use the mathematical induction to prove. According to the tree-structure algorithm, each query Q is transformed into a rule-goal tree using the ontological axioms. This rule-goal tree is actually an AND-OR tree, which means we have two types of rule nodes in the rule-goal tree: AND and OR.

  - Base case:

    * If the rule-goal tree consists of a single AND rule node, then we execute an incremental source evaluation, which is complete according to Lemma 4.

    * If the rule-goal tree consists of a single OR rule node, then we collect a sum set of its each child node's relevant sources, which is also complete.

– Recursive case:

Assume we have collected complete sources for the rule-goal tree consisting of k interactive AND and OR rule nodes: $N_1, \ldots, N_k$.

Given a new rule node N_{k+1} that will be added into the rule-goal tree, we need to prove the source collection of the new rule-goal tree is still complete.

* If N_{k+1} is a single AND node, then we execute an incremental source evaluation for N_{k+1} by using the available constatnt constraints. Since the souce collection for the previous k nodes is complete, according to Lemma 4, the source collection for N_{k+1} is also complete.

* If N_{k+1} is a single OR node, then we collect a sum set of its each child goal node's relevant sources. Similar to the AND case of N_{k+1}, since the souce collection for the previous k nodes is complete, the source collection for N_{k+1} is also complete.

Thus, the recursive case holds.

Therefore, the tree-structure algorithm is complete.

Therefore, the tree-structure algorithm is sound and complete. $\square$

Chapter 5

Cyclic Axiom Handling

The cyclic axiom handling for the query answering is particularly important because cyclic axioms are common in the real world and their correct processing can guarantee query completeness. This is because during the process of cyclic axioms, each iteration of each cyclic axiom could generate new substitutions to those recursive variables and these substitutions can be then propagated into the following iterations of this cyclic axiom. Thus, the process of each cyclic axiom needs a fix point computation in which case no more substitutions can be found. Corresponding to my algorithms, each cyclic axiom needs a fix point computation of the set of its selected relevant sources based on the term index for the given query. Then, those selected sources can be loaded into a complete reasoner to obtain complete answers to the original query. Thus, in order to guarantee the query completeness, I need special treatment for cyclic axioms. Furthermore, this process should be dynamic because data on the web is constantly in flux.

In this chapter, I discuss my improvement of the tree-structure algorithm in Section 4.4 to handle cyclic axioms. Since my algorithm is influenced by the *Magic Sets* theory [4], I will first give a brief introduction to the traditional *Magic Sets* theory, especially the part related to my algorithm. Then, I will describe my cyclic axiom handling algorithm in two parts: the dynamic cyclic axiom handling algorithm not including the equality reasoning and the equality reasoning (instance coreference). Finally, I will give the algorithm's correctness proof. Note, since Datalog notation provides a conventional form to represent cyclic axioms, I will follow the Datalog notation in this chapter. The mapping relation between an OWL axiom and a Datalog axiom is illustrated at Definition 19 in Section 5.1.

5.1 Magic Sets

Definition 19. *(Cyclic Axiom) A cyclic axiom is a rule that references the same atom on both its head and body. It has the form head :- $atom_1$, ..., $atom_n$, where $\exists i(1 \leq i \leq n \wedge head = atom_i)$. Map this rule to an OWL representation, its head and each $atom_i$ in the body is a triple pattern as defined at Definition 8. Formally, $atom_i/head = p(x, y)$ or $c(x) = \langle x, p, y \rangle$ or $\langle x, rdf\text{:}type, c \rangle$, where $p \in P$, $c \in C$, $x/y \in R \sqcup V \sqcup L$. Here, in a given semantic web space, C refers to the set of all classes, P refers to the set of all properties, R refers to the set of all constant URIs, L refers to the set of all literal terms, V refers to the set of all variables.*

According to the above definition, for instance, $ancestor(x, y)$:- $ancestor(x, z), ancestor(z, y)$ is a cyclic axiom. Note, a cyclic axiom may be explicit or inferred through the explicit ones. The *Magic Sets* method provides a strategy to compute

the fix point of cyclic axiom for simulating the top-down evaluation of a query by modifying the original cyclic rule by means of additional rules, which cut down on the irrelevant facts and narrow the computation to what is relevant for answering the query. Compared to the traditional forward chaining mechanism to compute the fix point of a cyclic axiom, the advantage of the *Magic Sets* is that by working top-down, we can take advantage of efficient methods for doing massive joins only using the relevant facts computed from the generated magic and modified rules, which will be introduced later in this part. The *Magic Sets* applies the SIPS strategy that describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. For instance, let V be an atom that has not yet been processed, and Q be the set of already considered atoms, then a SIPS specifies a propagation $Q \rightarrow_X V$, where X is the set of the variables bound by Q, passing their values to V.

The method is structured in four steps: rule adornment, rule generation, rule modification and query processing. They are illustrated as follows by considering the axiom $ancestor(X, Y) \text{ :- } ancestor(X, Z), ancestor(Z, Y)$ together with a query $ancestor(\text{``}John\text{''}, Y)$, where X, Y and Z are variables and $John$ is a given instance.

- Rule adornment: this phase is to materialize, by suitable adornments, binding information for predicates. These are strings of the letters b and f, denoting bound or free for each argument of a predicate in order. First, adornments are created for query predicates. The adorned query is $ancestor^{bf}(\text{``}John\text{''}, Y)$. In the given rule, $ancestor^{bf}(\text{``}John\text{''}, Y)$ passes its binding information to $ancestor(X, Z)$ by $ancestor^{bf}(X, Y) \rightarrow_X ancestor(X, Z)$. Then, $ancestor(X, Z)$ is adorned $ancestor^{bf}(X, Z)$. Now, I consider $ancestor(Z, Y)$, for which there

119

is no binding information and can still use the given axiom to expand it. Finally, I have two resulting adorned rules: $ancestor^{bf}(X,Y) \text{:-} ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$ and $ancestor^{ff}(Z,Y)$:- $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$, where W is a new introduced variable. Note, the second adorned rule is expanded from $ancestor^{ff}(Z,Y)$ using the given axiom.

- Rule generation: the adorned program is used to generate magic rules. For each adorned predicate p in the body of an adorned rule r_a, a magic rule r_m is generated such that (i) the head of r_m consists of $magic(p)$, which is essentially a new introduced predicate, and (ii) the body of r_m consists of the magic version of the head of r_a, followed by all of the regular predicates of r_a which can propagate the binding on p. Take the adorned rule of $ancestor^{bf}(X,Y)$:- $ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$ in Step (1) for example, we can generate two magic rules: $magic\_ancestor^{bf}(X,Z)$:- $magic\_ancestor^{bf}(X,Y)$, $ancestor^{ff}(Z,Y)$ and $magic\_ancestor^{ff}(Z,Y)$:- $magic\_ancestor^{bf}(X,Y)$, $ancestor^{bf}(X,Z)$. For the adorned rule $ancestor^{ff}(Z,Y)$:- $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$, two magic rules are also generated: $magic\_ancestor^{ff}(Z,W)$:- $magic\_ancestor^{ff}(Z,Y)$, $ancestor^{ff}(W,Y)$ and $magic\_ancestor^{ff}(W,Y)$:- $magic\_ancestor^{ff}(Z,Y)$, $ancestor^{ff}(Z,W)$.

- Rule modification: the adorned rules are modified by including magic atoms such as $magic\_ancestor^{bf}(X,Y)$ generated in Step (2) in the rule bodies. The resultant rules are called modified rules. For each adorned rule whose head is h, I extend the rule body by inserting $magic(h)$. Take the adorned rule of $ancestor^{bf}(X,Y)$:- $ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$ in Step (1) for example,

we can generate one modified rule: $ancestor^{bf}(X,Y)$:- $magic\_ancestor^{bf}(X,Y)$, $ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$. For the the adorned rule $ancestor^{ff}(Z,Y)$:- $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$, the generated modified rule is $ancestor^{ff}(Z,Y)$:- $magic\_ancestor^{ff}(Z,Y)$, $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$.

- Query processing: for each adorned predicate g^{α} of the query, (i) the magic seed $magic(g^{\alpha})$ is asserted, and (ii) a rule g :- g^{α} is produced. In the example, I generate $magic\_ancestor^{bf}(\text{``}John\text{''},Y)$ and $ancestor(X,Y)$:- $ancestor^{bf}(X,Y)$.

The complete rewritten program consists of the magic, modified, and query rules. Given a non-disjunctive datalog program P, a query Q, and the rewritten program P', it is well known that P and P' are equivalent w.r.t. Q [4]. In the *Magic Sets*, the adornments of Step (1) aims to cover all possible bound/free information based on the given query and rules. Then, the generated magic rules in the following steps can easily cut down the irrelevant facts and meanwhile guarantee the completeness during the fix point computation of the cyclic axioms. For the tree-structure algorithm in Section 4.4, the constant propagation mechanism is essentially the same as the SIPS strategy. According to the tree-structure algorithm, the available bindings of each goal node in the rule-goal tree are propagated to its child and sibling goal nodes in order to select relevant sources. In addition, because my purpose is to collect relevant sources by constructing boolean queries using the available constant constraint (bound value) and the predicate instead of the real computation of the fix point, which is actually accomplished by the *Reasoner*, it is sufficient to only apply the rule adornment step into my algorithm. Then, based on the adorned rule-goal tree, I can easily detect whether two goal nodes have the same predicate and

adornment. If so, a cycle is formed and I can correspondingly collect those sources that are necessary for this cycle's fix point computation.

5.2 Cyclic Axiom Handling Algorithms

In this section, I first introduce my dynamic cyclic axiom handling algorithm not including equality reasoning. Then, I discuss the equality reasoning (instance coreference).

5.2.1 Cyclic Axiom Handling

When cyclic axioms are considered, the tree-structure algorithm is incomplete. This is because it does not load all relevant sources that corresponds to a query subgoal, but instead only loads those that contain the subgoal predicate and its available variable constraints. On the other hand, each iteration in the cyclic axiom could generate recursive variable constraints that can be propagated into the following iterations to collect sources. Consequently, the tree-structure algorithm will miss those sources collected by applying the recursive variable constraints in each iteration. For instance, take the cyclic axiom $ancestor(x, y) :\text{-} ancestor(x, z), ancestor(z, y)$ and its query $ancestor(\text{``}John\text{''}, y)$. Assume we have collected sources containing the substitutions $\{z/Bob, y/Andy\}$ by using the subgoals $ancestor(\text{``}John\text{''}, z)$ and $ancestor(z, y)$ respectively on the term index, the tree-structure algorithm then finishes processing this axiom because all of its subgoals have been handled and their corresponding sources have been also collected. However, those sources containing the recursive descendants of *Bob* and *Andy* are still relevant and will be missed

because of no recursive source collection of the given axiom. Therefore, in order to guarantee the completeness, we need special treatment for cyclic axioms. Furthermore, this process should be dynamic because data on the web is constantly in flux.

In order to handle cyclic axioms, there are four key points I need to particularly take care of:

- How to represent and annotate cyclic axioms in the original rule-goal tree of the query reformulation?

- Within each iteration of one cyclic axiom, how to compute the new generated substitutions of the given cyclic axiom that will be passed into the next iteration? In this process, we call the set of new substitutions *Relevant Substitutions (RS)*.

- How to apply the RS into the selection of relevant sources by using the term index?

- In case of multiple cyclic axioms mutually nested in one query, how to identify their correct computation order?

For the first point, as the traditional *Magic Sets* theory does, I adorn the cyclic axioms by using their binding information. Then, I mark them in the rule-goal tree. In theory, if one goal node G is detected to be one that can be unified with its one ancestor goal node A on condition that G and A are the same predicate and have the same adornments, then I detect a cycle C starting with A and ending with G. However, in practice, I apply that if A and G also have the same bound value, then

they are not a cycle because A and G collect the same sources by using the term index and there is no recursive source collection. Formally, a cycle C is denoted as $C = \langle A, G \rangle$, where A is C's starting node and G is C's ending node. After the cycle is marked, the rule-goal tree is transformed into a rule-goal graph and each cyclic axiom is converted into one or more rule-goal graph cycles correspondingly. Essentially, a rule-goal graph cycle means its corresponding axioms need to be iteratively traversed until a fix point of its source collection is reached. For the second point, in the rule-goal graph, the RS of each iteration for one cyclic axiom essentially consists of the new generated substitutions of the cycle distinguished variables (CDVs) of this cyclic axiom's rule-goal graph cycles. I define each graph cycle's CDVs to be a set of the distinguished variables of the starting node, which is actually the head of the first axiom involved in this cycle. At the end of each cycle iteration, I compute the RS by issuing a conjunctive query consisting of the child goal nodes of the starting node of the cycle to *Reasoner*. Then, I apply it into the next iteration if the new RS is changed. Otherwise, it means we have reached the source collection fix point of the current cycle. Furthermore, if the RSs of all cycles in the rule-goal graph for one given cyclic axiom are unchanged, it means the source collection fix point of this cyclic axiom has been reached. For the third point, I use the conjunction of each value in the RS and the goal predicate to query the term index. This helps to significantly reduce the number of potentially relevant sources because of the constant constraints. For the fourth point, I will employ a cycle stack to plan the cyclic axiom handling sequence. Each cycle can be pushed onto the stack only if it is not in the stack. Otherwise, its process will be postponed.

I begin with the cyclic axiom $ancestor(x, y) :\text{-} \, ancestor(x, z), ancestor(z, y)$ and
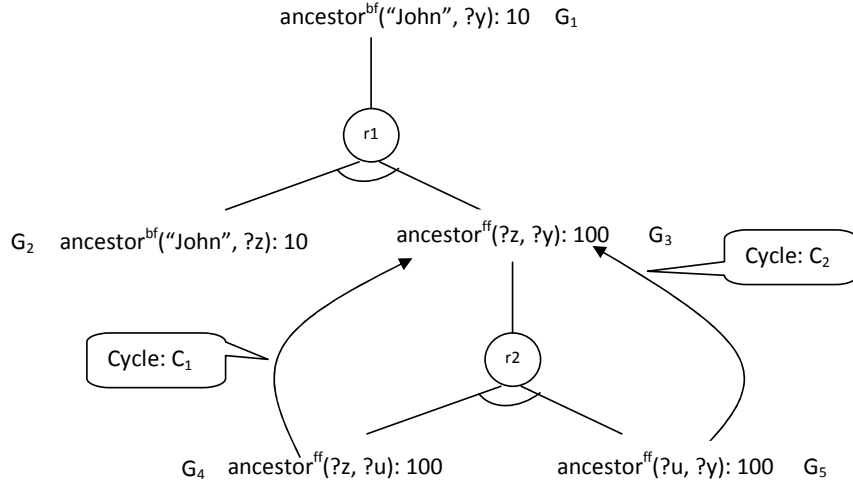
Figure 5.1: An example cyclic axiom

its query $ancestor($"*John*"$, y)$ to introduce my algorithm. Figure 5.1 shows its rule-goal graph. The back arrow means a cycle is marked. Each goal node has associated adornments (*bf* or *ff*) and selectivity (the number of relevant sources).

At the beginning, using the term index, each goal node of the rule-goal graph is initialized with their respective selectivities and bindings. In this example, there are two cycles: $C_1 = \langle G_3, G_4 \rangle$ and $C_2 = \langle G_3, G_5 \rangle$. I use S to stand for the cycle stack. Initially, the algorithm starts with the most selective node G_2 and uses its substitutions e.g. $\{z/Bob\}$ to constrain its sibling G_3. As a result, the selectivities of G_3 and its child G_4 are both updated to be 20 for example by issuing a boolean query "*ancestor* $\wedge$ *Bob*". Then, the algorithm starts with G_3's most selective node G_4. At this time, C_1 is detected and pushed onto S. Then, the algorithm starts to process C_1 by starting with G_4. Now, C_1 is detected again and postponed because it is also already in S. The algorithm evaluates G_4 and applies its available constant substitution e.g. $\{u/Andy\}$ into its sibling G_5, whose selectivity is updated to be 15 for instance by issuing a boolean query "*ancestor* $\wedge$ *Andy*" and where C_2 is detected

and pushed onto S. Now, S contains C_1 and C_2. The algorithm starts to process C_2 still beginning with G_4. Note in this step, the substitution $\{u/Andy\}$ is propagated upward from G_5 to G_3. As a result, the selectivities of G_3 and its child G_4 are both updated to be 15 for instance by issuing a boolean query "$ancestor \wedge Andy$". During this process, C_1 is detected and postponed again. At this step, C_2's initial RS (Relevant Substitutions) is $\{z/Andy, y/Jim\}$. Then, the algorithm evaluates G_4 and applies its substitutions $\{u/Jim\}$ for example to G_5, where C_2 is detected again and postponed. Then, G_5's selectivity is updated to be 5 for example by issuing a boolean query "$\{ancestor \wedge Jim\}$" and its new substitution is propagated upward to G_3 again. Now, we are at the end of the second iteration of C_2 and compute its new RS to be $\{z/Andy, y/Jim\}, \{z/Jim, y/Tim\}$. Compare to the previous RS, we have obtained one new substitution $\{z/Jim, y/Tim\}$, which is then applied into the next iteration of C_2 to select relevant sources. This process is repeated until C_2's next new RS is unchanged compared to the last one. This means C_2's source collection fix point has been reached and C_2 is popped. Now, S only contains C_1. Then, the algorithm goes back to the context of C_1 to do the same process as C_2's. Obviously, in processing the next iteration of C_1, C_2 will be met again. The previous C_2 process is repeated. Meanwhile, at the end of each C_1's iteration, the algorithm also computes C_1's RS and applies it into its next iteration to collect sources until the new RS of C_1 is unchanged meaning C_1's source collection fix point has been reached. Finally, the algorithm finishes processing C_1 and C_2 and correspondingly collects all relevant sources of the given cyclic axiom.

Note, in the above process, $C_1 = \langle G_3, G_4 \rangle$ and $C_2 = \langle G_3, G_5 \rangle$ are actually redundant cycles because they collect the same data sources. Therefore, we need to

avoid such repeated source collections. My algorithm detects if two cycles are redundant, which means that each node in one cycle exactly has the same predicate and adornments as a node in the other one and vice versa. These redundant cycles are categorized into different redundant cycle classes and stored into a structure called Redundant Cycle Base (RCB), which is created to collect and organize redundant cycles. A redundant cycle class in RCB is a set of cycles that contain the same axioms and adornments. Redundant cycles cause redundant source collection because they could generate the same recursive constants and then collect the same sources multiple times. Therefore, during the process of each redundant cycle, I need to check if the new recursive constant has been used by other cycles that are redundant with the current cycle. If not, I continue to start the next generation. Otherwise, I will skip this constant. Here, the algorithm cannot handle only one cycle instead of other cycles belonging to the same redundant cycle class because redundant cycles could have different recursive constants generated to collect different data sources due to their different positions in the rule-goal graph. Then, even though $C_1 = \langle G_3, G_4 \rangle$ and $C_2 = \langle G_3, G_5 \rangle$ are both pushed onto the cycle stack in the given example, I can still avoid the repeated source collection. In addition, for those instances that match query constants or that are used as join conditions, their equivalence (*owl:sameAs*) closures of source collection are computed on the fly by calling the equality reasoning algorithm in Section 5.2.2.

The pseudo code for the cyclic axiom handling algorithms is shown in Figures 5.2 and 5.3. These algorithms are based on the tree-structure algorithm in Figure 4.7. The bold lines in Algorithm 5 and Algorithm 6 and the whole Algorithm 7 are new parts to handle cyclic axioms. During the execution, Algorithm 5 calls Algorithms

---

Algorithm 5 source selection

---

function getSourceList(*rgraph, rs, q*)**returns** a list of sources

        inputs: *rgraph*, a given rule-goal graph (cyclic or non-cyclic)

            rs, a list of substitutions

            q, a list of query triple patterns

1: Let *frontier* = leaf nodes or cycle ending nodes*,* **static *EKB* = ϕ, static *RCB* = ϕ**

        srcs[] = array of sets of sources, indexed by goal nodes

2: **for each** goal node *n* in *rgraph* **do**

3:        **if *n* has constant *C* and *C*.*equivalenceClass* ∉ *EKB* then**

4:         **computeSameAs(*{C}*, *EKB*)**

5:        **for each $\theta \in rs$ do**

6:         ***srcs[n]* = qsources(*nθ*, *EKB*)**

7: **do**

8:        **Let *n* = $min_{node \in frontier}$ (|*srcs*[*node*]|), *p* = *n*.parent**

9:        **if *n* is a cycle ending node AND *n*.*cycle* ∉ *CycleStack* then**

10:         **update(*n.cycle*, *RCB*)**

11:         **push(*CycleStack*, *n.cycle*)**

12:         $srcs[n] = srcs[n] \cup$ **getCyclicSourceList**$(n.cycle, rs, q)$

13:         **pop(*CycleStack*)**

14:        **if *n* is a child of an AND rule node *r* then**

15:            $srcs[p] = srcs[p] \cup$ OptimizeANDNode($rgraph$,

        $n, siblings\ of\ n, srcs, q, EKB, RCB$)

16:        **else**

17:            $srcs[p] = srcs[p] \cup srcs[n]$

18:            **if *n* is a child of *rgraph.root* and *rgraph* is a cycle then**

19:          **load(*srcs[n]*, *KB*)**

20:        **Let *rsc* = askReasoner (*KB*, *rgraph*)**

21:        **Let *insts* = extractJoinInsts(*rsc*)**

22:        **computeSameAs(*insts*, *EKB*)**

23:        ***rgraph.RS* = computeRS(*rgraph.CDVs*, *RCB*)**

24:        remove *n* and its siblings from *frontier*

25:        **if** *p* has no descendants on *frontier* **then**

26:            add *p* to *frontier*

27: **while** ($frontier \neq \{rgraph.root\}$)

28: **return** *srcs[rgraph.root]*

Figure 5.2: The cyclic axiom handling algorithm - part 1

Algorithm 6 node optimization

---

function OptimizeANDNode(*rgraph,on,sibs,srcs,q,EKB,RCB*)**return** a list of sources

       inputs: *rgraph*, a rule-goal graph; *on*, a goal node

         sibs, *on*'s sibling nodes;   *srcs*, an array of sets of sources

         q, a list of query triple patterns; *EKB*, the EquivalenceKB;

         RKB, the redundant cycle base

1:    Let $allsrcs = srcs[on]$, load(*srcs[on]*, *KB*)

2:   **do**

3:   q = $q \land on$, rs = askReasoner (*KB*, *q*)

4:   **Let *insts* = extractJoinInsts(*rs*)**

5:   **computeSameAs(*insts*, *EKB*)**

6:      **for each** $qtp \in sibs$ **do**

7:         *srcs[qtp]* = getSourceList(subgraph rooted at *qtp*, *rs*, *q*)

8:   Let *on* = $min_{t \in sibs\ that\ join\ with\ query} (srcs[t])$

9:   Remove *on* from *sibs*

10:    $allsrcs = allsrcs \cup srcs[on]$, load($srcs[on], KB$)

11:   **if *on* is a child of *rgraph.root* AND *rgraph* is a cycle AND**

    $sibs = \emptyset$ **then**

12:   **load(*srcs[on]*, *KB*)**

13:   **Let rsc = askReasoner (*KB*, $rgraph$)**

14:   **Let *insts* = extractJoinInsts(*rsc*)**

15:   **computeSameAs(*insts*, *EKB*)**

16:   **rgraph.RS = computeRS(*rgraph.CDVs, RCB*)**

17: **while** ($sibs \neq \emptyset$)

18: **return** *allsrcs*

Algorithm 7 source selection for cyclic axioms

---

function getCyclicSourceList(*rgraph, rs, q*) **returns** a list of sources

      inputs: *rgraph*, a given rule-goal graph;   *rs*, a list of substitutions

     q, a list of query triple patterns

1: Let $rsInc = rs$, *firstIt* = true, *allsrcs* = $\emptyset$

2: **while** ($rsInc \neq \emptyset$ **OR** *firstIt*)

3:   $allsrcs = allsrcs \cup$ getSourceList($rgraph, rsInc, q$)

4:   $rsInc$ = *rgraph.RS*

5:   clear(*rgraph.RS*), *firstIt* = false

6: **return** *allsrcs*

Figure 5.3: The cyclic axiom handling algorithm - part 2

6 and 7 to compute the source collection fix point of cyclic axioms. Among them, Lines 3-4 and 22 in Algorithm 5, and Lines 5 and 15 in Algorithm 6 are for equality reasoning that will be introduced in Section 5.2.2. Lines 9-13 in Algorithm 5 are for source collection of cyclic axioms contained in the original query using cycle stack. Lines 18-23 in Algorithm 5 and Lines 11-16 in Algorithm 6 are for recursive constant computation that will be passed into the next iteration to collect new sources.

In Algorithm 5, RCB stands for Redundant Cycle Base. EKB is a structure that collects and organizes equivalence information about instances that will be illustrated in Section 5.2.2. In the given rule-goal graph rgraph, each goal node has been adorned with its own binding information. In line 6 of Algorithm 5, $qsources$ is as defined in Definition 12 in Section 3.3. Given a QTP q and a term index I, $qsources(q, EKB) = \bigcap_{c \in terms(q,EKB)} I(c)$, which is essentially a set of data sources that are relevant with q. The EKB is used here to collect q's relevant sources by using both q's constants and their equivalent constants in EKB. In line 9, when the current most selective QTP (on) is a cycle ending node, it means that a cycle is detected and I can use it to update the RCB and then push it onto the cycle stack (Lines 10 and 11). Note, each goal node in the rule-goal graph can be only involved in one cycle as an ending node because two cycles sharing one ending node is equivalent to one cycle starting and ending at these two cycle's root nodes respectively that should have been annotated before. Then, Algorithm 7 is called to compute the cycle's source collection fix point (Line 12). It repeatedly collects sources by executing Algorithm 5 if the current cycle's RS is changed (Lines 2-5). Here, the RS are computed at the end of each cycle iteration in lines 18-23 of Algorithm 5 and lines 11-16 of Algorithm 6 by extracting the new substitutions of the

current cycle's CDVs (Cycle Distinguished Variables) and then passed to Algorithm 7 for the recursion use. In this process, the function $extractJoinInsts(rsc)$ extracts join instances from the given substitution list rsc (Line 21 in Algorithm 5, and Lines 4 and 14 in Algorithm 6). Its results are passed to instance coreference handling algorithm (Algorithm 8 in Figure 5.4) to compute the owl:$sameAs$ source collection closure (Lines 4 and 22 in Algorithm 5, and Lines 5 and 15 in Algorithm 6). The function $computeRS(rgraph.CDVs.RCB)$ is to compute the RS of the given rule-goal graph rgraph (Line 23 in Algorithm 5 and Line 16 in Algorithm 6). Here, for each recursive constant, the algorithm checks if the redundant cycles of the current cycle have used it before using RCB and skips it from RS if it has been. When the source collection fix point is reached, the algorithm returns all collected sources (Line 6) and goes back to Algorithm 5. Then, Line 13 in Algorithm 5 is continually executed to pop the processed cyclic axiom.

5.2.2 Equality Reasoning

The equality reasoning is handled in purpose of computing the source collection fix point of instant coreference using owl:$sameAs$ in order to guarantee query completeness. The owl:$sameAs$ is a special case of cyclic axiom because it is a ubiquitous transitive property as defined: owl:$sameAs$:- owl:$sameAs$, owl:$sameAs$. Thus, the cyclic axiom handling algorithm in Section 5.2 is able to handle the owl:$sameAs$ reasoning. In order to do so, a query needs to be rewritten using owl:$sameAs$. For example:

- Original query: q :- name(x, "Tim Berners-Lee"), knows(x, y).

131

- Rewritten query: q :- name(x, "Tim Berners-Lee"), $owl{:}sameAs$(x, v_1), knows(v_1, v_2), $owl{:}sameAs$(y, v_2).

After the rewritting, we now have new subgoals $owl{:}sameAs$(x, v_1) and $owl{:}same$ As(y, v_2). In the query reformulation, each of them is marked as a cycle. In order to compute the source collection fix point of the rewritten query, the cyclic axiom handling algorithm starts with the most selective subgoal such as $name(x, "TimBerners$ $-Lee")$ in this example and iteratively applies the available constant constraints to other subgoals to collect sources. During this process, the availabe constant constraints for each variable are computed by answering the intermediate subqueries. As a result, those subqueries including $owl{:}sameAs$ QTPs suffer from the explosive combination of their answers that makes the *Reasoner* stuck because of the complicated equality reasoning caused by $owl{:}sameAs$ that generates large number of answers. Assume the number of answers to q named $ans(q)$ is $num\_ans(q)$, the max cardinality of the set of equivalent instances of each instance substitution to q is $maxEqSize(ans(q))$ and the number of variables in q is $num\_var(q)$. Then, due to the introduction of $owl{:}sameAs$, the numbers of answers to the subqueries of q will increase by $num\_ans(q) \times maxEqSize(ans(q))^{num\_var(q)}$ times compared to those subqueries without the $owl{:}sameAs$ rewriting, which makes the *Reasoner* stuck. Consequently, the rewriting-based equality reasoning cannot scale. Therefore, I propose an optmized equality reasoning algorithm that can overcome this drawback.

The optimized equality reasoning ($owl{:}sameAs$) algorithm is based on the heuristic that within the term index, the QTPs with constant constraints are often highly selective and thus belonging to highly selective QTPs. For instance, given two

QTPs: *owl*:*sameAs*(*rpi*:*james*, ?y) and *owl*:*sameAs*(?x, ?y), the first is much more selective than the second because of the specific constant *rpi*:*james*. Therefore, compared to the way of loading all sources containing the *owl*:*sameAs* predicate to compute the instance coreference closure, this way helps significantly reduce the number of sources that are involved in the closure computation. Given a query, I define the set of all instances that are used for the instance coreference fix point closure computation as Relevant Instances (RI). Since I only compute the equivalence closure of the query constant instances and the join instances during the query solving, the cardinality of RI is often small. In my design, an EquivalenceKB structure (EKB) is created to collect and organize equivalence information about instances in RI. EKB essentially supports the disjoint set data structure operations on sets of equivalence classes of all known instances. An equivalence class in EKB is a set of instances that are equivalent to each other (explicitly or implicitly connected by *owl*:*sameAs*). Given an instance Ins in RI, a boolean query "Ins" AND "*owl*:*sameAs*" is dynamically issued to the term index to find all relevant sources that contain Ins and its explicit equivalent instances. Then, for each new discovered instance $newIns$, I further find $newIns$'s equivalent instances and merge the equivalence classes containing Ins and $newIns$. This process is repeated until no new instances are discovered, which means the source collection fix point of the equality reasoning is achieved. Note, the equivalence class of each instance in RI is only computed once. During this process, since I compute the *owl*:*sameAs* source sollection closure of each instance separately instead of constructing intermediate queries as the rewriting way does, the explosive combination problem can be solved.

Algorithm 8 Fix point computation for equality reasoning

---
function computeSameAs(*insts*, *EKB*) **returns** a list of instances

1:    **inputs**: *insts*, a list of seed URIs

2:    **Let** *inslist* = *insts, oldinsts = insts*

3:    **for each** uri ∈ insts **do**

4:        **Let** *bquery* = *uri* + "AND" + "*owl:sameAs*"

5:        **Let** *srcslist* = askIndexer(*bquery*)

6:        **for each** *s* ∈ *srcslist* **do**

7:            Let *sameAsPairs* = {t | t = < *x, owl: sameAs, y* > ∈ *s, x* = *uri* ⋁ y = uri}

8:            updateEquivalenceKB(*uri*, *sameAsPairs*, *EKB*)

9:            *inslist* = *inslist* ⋃ all instances URIs from sameAsPairs

10: **Let** *newinsts* = *inslist* − *oldinsts*

11: *inslist* = *inslist* ⋃ ComputeSameAs(*newinsts*, *EKB*)

12: **return** *inslist*

Figure 5.4: Equality reasoning algorithm

The pseudo code of equality reasoning algorithm is shown in Figure 5.4. First, the algorithm starts with a set of seed instance URIs (Line 2), and uses the term index to find all sources that contain each of these URIs concatenated with the "*owl:sameAs*" predicate (Lines 4 and 5). Note, the seed instances are not all coreferenced instances, but the instances in the RI of the given query and determined by Algorithm 5 (Figure 5.2). Then, the algorithm extracts new equivalent URIs (Line 7), merges equivalence classes of seed URIs and new extracted URIs (Line 8), and collects new URIs (Line 9). This process is iteratively repeated by using any new URIs discovered as seeds (Lines 10-11) until no more new URIs are discovered.

5.2.3 Correctness Proof

Before I introduce the correctness proof, I first give the definition of a finite Semantic Web Space in Definition 20.

Definition 20. *A Semantic Web Space SWS* $\langle \mathcal{U}, o, s \rangle$ *is finite if its set of document*

identifiers $\mathcal{U}$ *is finite.*

Here, a finite SWS means that it has finite number of ontological axioms and facts, which can be exhausted by the cyclic axiom handling algorithm. As a result, the cyclic axiom handling algorithm can terminate eventually.

Theorem 5. *Given a finite Semantic Web Space SWS that contains cyclic rules and a conjunctive query Q, the cyclic axiom handling algorithm is sound and complete for ontologies that can be represented in AND/OR relations.*

Proof. I will first prove the soundness, and then the completeness.

- Soundness:

  Assume the set of sources collected by the cyclic axiom handling algorithm is $srcs(Q)$ and the set of answers to Q entailed by $srcs(Q)$ is Ans. Then we have $srcs(Q) \subseteq SWS$. Thus, we can get $\forall\theta(\theta \in Ans \wedge Theory(srcs(Q)) \models Q\theta \rightarrow Theory(SWS) \models Q\theta)$. Therefore, the cyclic axiom handling algorithm is sound.

- Completeness:

  Since the given SWS is finite, the algorithm can terminate. In addition, I assume the rule-goal graph generation process is correct. According to Section 5.2.1, the cyclic axiom handling algorithm employs a cycle stack to handle multiple cycles that are mutually nested. Thus, each recursive QTP in the rule-goal graph becomes non-recursive when the cycle which the recursive QTP belongs to is pushed onto the cycle stack. As a result, the algorithm can be able to focus on processing only one cycle by skipping other nested ones that

has been pushed onto the cycle stack. Therefore, the completeness proof of the cyclic axiom handling algorithm has two key parts: the completeness of the cycle stack strategy and the completeness of the source collection of one single cycle handling. I will prove the completeness of the algorithm from the following four cases:

(1) The rule-goal graph generated by the cyclic axiom handling algorithm consists of only one single cycle.

Assume a goal node p_i is the recursive QTP in the only cycle C that needs a source collection fix point computation. At the beginning, the available answer set $Ans_i = \emptyset$. I will prove the completeness using the mathematical induction method as follows:

- Base case:

  According to Definition 15, $qtp\_sources(p_i, Ans_i) = \bigcup_{\theta \in Ans_i} qsources(p_i\theta) = qsources(p_i)$.

  According to Lemma 1, $MS_Q(p_i) = \bigcup_{\theta_i \in \theta \wedge \theta \in Ans\_SWS(Q)} qsources(p_i\theta_i)$.

  Thus, according to Lemma 3, $MS_Q(p_i) \subseteq qtp\_sources(p_i, Ans_i)$.

- Recursive case:

  Assume $MS_Q(p_i) \cup MS_Q(p_{i+1}) \cup \ldots \cup MS_Q(p_{i+k}) \subseteq qtp\_sources(p_i, Ans_i) \cup qtp\_sources(p_{i+1}, Ans_{i+1}) \cup \ldots \cup qtp\_sources(p_{i+k}, Ans_{i+k})$, where $p_i, \ldots, p_{i+k}$ are QTPs that are involved into C taking p_i as the recursive one.

  Then, given Ans_{i+k+1}, we need to prove:

$MS_Q(p_i) \ \cup \ MS_Q(p_{i+1}) \ \cup \ \ldots \ \cup \ MS_Q(p_{i+k}) \ \cup \ MS_Q(p_{i+k+1}) \ \subseteq$ $qtp\_sources(p_i, Ans_i) \ \cup \ qtp\_sources(p_{i+1}, Ans_{i+1}) \ \cup \ \ldots \ \cup \ qtp\_sources$ $(p_{i+k}, Ans_{i+k}) \ \cup qtp\_sources(p_{i+k+1}, Ans_{i+k+1})$, where p_{i+k+1} is also involved into C.

In order to prove this rule, we need to consider two possibilities:

* $p_{i+k+1} \neq p_i$, which means p_{i+k+1} is a non recursive QTP in this cycle. In this case, the proof is the same as the recursive case proof of Lemma 4. Thus, the recursive case holds.

* $p_{i+k+1} = p_i$, which means p_{i+k+1} is the recursive QTP and we are entering the next iteration of p_i's source collection fix point computation. In this case, $qtp\_sources(p_{i+k+1}, Ans_{i+k+1}) = qtp\_sources$ (p_i, Ans_{i+k+1}). Since the function of $qtp\_sources$ for a given QTP is complete according to Lemma 3, the next step is to prove the process of computing Ans_{i+k+1} based on Ans_{i+k} is complete, where either Ans_{i+k+1} or Ans_{i+k} is an intermediate answer set during p_i's source collection fix point computation.

  Since the computation of Ans_{i+k+1} from Ans_{i+k} is the same as the tree-structure algorithm and its corresponding source collection is complete, according to Theorem 4, the computation of Ans_{i+k+1} from Ans_{i+k} is also complete.

Then, $MS_Q(p_{i+k+1}) \subseteq qtp\_sources(p_{i+k+1}, Ans_{i+k+1}) = qtp\_sources(p_i, Ans_{i+k+1})$.

Thus, $MS_Q(p_i) \ \cup \ MS_Q(p_{i+1}) \ \cup \ \ldots \ \cup \ MS_Q(p_{i+k}) \ \cup \ MS_Q(p_{i+k+1}) \ \subseteq$ $qtp\_sources(p_i, Ans_i) \cup qtp\_sources(p_{i+1}, Ans_{i+1}) \cup \ldots \cup qtp\_sources(p_{i+k},$

$Ans_{i+k})\ \cup qtp\_sources(p_{i+k+1}, Ans_{i+k+1})$.

Therefore, the recursive case holds.

According to Lemma 2, we can have $MS_Q(C) = \bigcup_{p_i \in atoms(C)} MS_Q(p_i)$.

Therefore, we can conclude that the cyclic axiom handling algorithm returns a superset of sources of $MS_Q(C)$.

Assume the returned set of sources is srcs(Q). Then, we have $\forall \theta (\theta \in Ans\_SWS$ $(Q) \wedge SWS \models Q\theta \rightarrow Theory(srcs(Q)) \models Q\theta)$.

According to Definition 17, the cyclic axiom handling algorithm is complete in case (1).

(2) The rule-goal graph generated by the cyclic axiom handling algorithm consists of two parts: one single cycle and the acyclic part.

Its proof consists of the following two parts:

- The acyclic part: it is the same as the tree-structure algorithm, which is complete according to Theorem 4.

- The single cycle part: it is the same as case (1), which is also complete.

Thus, the cyclic axiom handling algorithm is complete in case (2).

(3) The rule-goal graph generated by the cyclic axiom handling algorithm contains mulitple non nested cycles.

Its proof consists of the following two parts:

- The acyclic part: it is the same as the tree-structure algorithm, which is complete according to Theorem 4.

– The cycle part: assume in the generated rule-goal graph, we have a set of cycles $C = C_1, \ldots, C_i, \ldots, C_k$, which are non nested. According to case (1), the cyclic axiom handling algorithm is complete for each C_i, where $C_i \in C$. Thus, the cyclic axiom handling algorithm is complete for C.

Thus, the cyclic axiom handling algorithm is complete in case (3).

(4) The rule-goal graph generated by the cyclic axiom handling algorithm contains mulitple nested cycles.

Its proof consists of the following two parts:

– The acyclic part: it is the same as the tree-structure algorithm, which is complete according to Theorem 4.

– The cycle part: assume in the generated rule-goal graph, we have a set of cycles $C = C_1, \ldots, C_i, \ldots, C_k$, which are nested.

According to Section 5.2.1, the cyclic axiom handling algorithm employs a cycle stack to handle multiple cycles that are mutually nested. Since each single cycle process has been approved to be complete in case (1), the completeness of case (4) can be attributed to the completeness of my cycle stack strategy, which I will prove using proof by contradiction.

Assume the set of sources collected by the cyclic axiom handling algorithm in case (4) is S.

Assume the cycle stack strategy is incomplete, $\exists s \exists \theta \exists TP (\theta \in Ans\_SWS$ $(Q) \wedge TP \in qtps(Q\theta) \wedge s \models TP \wedge s \notin S)$.

Since $S = \bigcup_{C_i \in C \wedge p_i \in atoms(C_i)} qtp\_sources(p_i, Ans_i)$, $s \notin \bigcup_{C_i \in C \wedge p_i \in atoms(C_i)}$ $qtp\_sources(p_i, Ans_i)$.

In other words, $\exists C_i (C_i \in C \wedge p_i \in atoms(C_i) \wedge s \notin qtp\_sources(p_i, Ans_i))$. Since the stack operation is standard and the source collection for one single cycle has been proved to be complete in case (1), this is a contradiction.

Thus, my cycle stack strategy is complete.

Therefore, the cyclic axiom handling algorithm is complete for C.

Thus, the cyclic axiom handling algorithm is complete in case (4).

Based on the proof of cases (1), (2), (3) and (4), the cyclic axiom handling algorithm is complete.

Then, we have that the cyclic axiom handlig algorithm is sound and complete.

$\square$

Chapter 6

Evaluation

This chapter describes several experiments that I have conducted to empirically evaluate my algorithms. For the evaluation of each algorithm, I have conducted two groups of experiments: the heterogeneity evaluation with multiple ontologies and the large scale evaluation. I first describe a multi-ontology benchmark - Lehigh Customizable Data-driven Benchmark (LCDBM) that is used in the heterogeneity evaluation. Then, I introduce my real world experimental data set that is used in the large scale evaluation. After that, I describe three sets of experiments I have conducted to evaluate my non-structure (Section 4.2), flat-structure (Section 4.3), tree-structure (Section 4.4) and cyclic axiom handling (Section 5.2) algorithms. All experiments are done under OWLII ontology expressivity (Definition 1) and on a UNIX workstation with Xeon 2.93G CPU and 6G memory. In all cases, I use Lucene as my **Indexer** and KAON2 as my **Reasoner**.

KAON2 [53] is a hybrid reasoner which is able to reason with a larget set of OWL DL apart from nominals, corresponding to the Description Logic $SHIQ(D)$,

and Disjunctive Datalog, along with basic built-in predicates to deal with integers and strings.

For reasoning, contrary to most currently available DL reasoners, KAON2 does not implement the tableau calculus. Rather, reasoning in KAON2 is implemented by novel algorithms which reduce a $SHIQ(D)$ knowledge base to a disjunctive datalog algorithm. These novel algorithms allow applying well-known deductive database techniques, such as join-order optimizations, to DL reasoning.

KAON2 supports answering conjunctive queries that can be formulated using SPARQL. Much of, but not entire SPARQL specification is supported. In particular, only those queries are supported which correspond naturally to conjunctive queries.

The major advantage of KAON2 is that it is a very efficient reasoner when it comes to reasoning with Description Logics ontologies containing very large ABoxes and small TBoxes.

6.1   Lehigh Customizable Data-driven Benchmark (LCDBM)

Generally, the best way to evaluate algorithms is to use real Semantic Web data. However, I observe that currently available real Semantic Web data sets such as Linked Open data cloud and Billion Triple Challenge data suffer from the following drawbacks:

- They have little ontology integration. Even when they do, the mappings between different ontologies are often simple (in one-depth of $rdfs$:$subClass$

6.1. LEHIGH CUSTOMIZABLE DATA-DRIVEN BENCHMARK (LCDBM)

Of/owl:$equivalentClass/rdfs$:$subPropertyOf/owl$:$equivalentProperty$ rela-
tionships). Consequently, real Semantic Web data sources lack heterogeneity.

- The real world data may have a lot of incorrect information and syntactic
flaws, which lead to significant cleaning effort before they can be initialized.

Due to the above problems, I realize that I should not just evaluate my sys-
tem over the real world data, but also a synthetic data set that has better quality
ontology mappings and increased ontology expressivities. For this purpose, I devel-
oped a multi-ontology benchmark - Lehigh Customizable Data-driven Benchmark
(LCDBM) which can also be used to benchmark other scenarios.

The LCDBM takes a two-level user customization model including an ontology
profile and a web profile for users to describe scenarios required in their specific
evaluations. In this model, the ontology profile allows users to customize the ontol-
ogy expressivities by setting the relative frequency of various ontology constructors,
while the web profile provides users a way to customize the distribution of different
types of desired ontologies.

One sample two-level model is shown in Figure 6.1. In this model, the web profile
shows seven types of ontologies to be generated: RDFS, OWL Lite, OWL DL, DHL,
OWL 2 EL, OWL 2 RL, and OWL 2 QL. Their distribution probabilities are set to
be 0.3, 0.1, 0.2, 0.1, 0.1, 0.1 and 0.1 respectively. This configuration means that in
the final generated ontologies, 30% use RDFS, 10% use OWL Lite, 20% use OWL
DL, 10% use DHL and 10% use each of the three OWL 2 profiles. For each ontology
profile, the distributions of different ontology constructors used in the generated
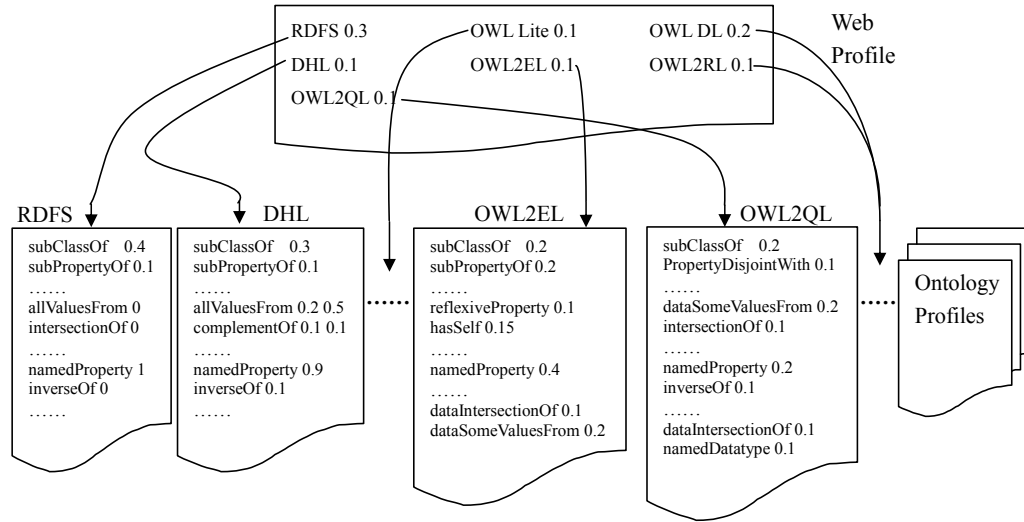ontology are also displayed.

143

Figure 6.1: Two-level customization model.

There are two core functions in the LCDBM implementation: the axiom construction for the domain ontologies and mapping ontologies and the data-driven query generation. In the following parts, I will introduce them respectively.

6.1.1 Axiom Construction

The web profile is used to select the user configured ontology profile(s). Then, the algorithm randomly constructs one parse tree for each ontological axiom by selecting constructors from four constructor tables (Table 6.1): the axiom type (AT) table, the class table (CT), the object property constructor table (OPT) and the datatype constructor table (DTT). There are ten types of operands in total: class type (C), named class type (NC), object property type (OP), named object property (NOP), instance type (I), named datatype property (NDP), facet type (F), data type (D), a literal (L) and an integer number (INT). Note, since the named datatype property (corresponding to the *DatatypeProperty* constructor) is the only data type

144

property in OWL 2, I do not list the *DatatypeProperty* constructor in the table. The C means the operand is either an atomic named class or a complex sub-tree that has a class constructor as its root. The NC means the operand is a named class. The OP means the operand can be one of constructors listed in the table of object property. The NOP, NDP means the operand is not a complex constructor but a named object property or a named datatype property respectively. The I means the operand can be a single instance. The F is the facet type borrowed from XML Schema Datatypes. The D is the data type. The L is a literal. The INT stands for an integer number for the cardinality restriction. In these types, NC, NOP, NDP, F, I, L and INT are leaf node types.

In Table 6.1, $\{x\}$ and $\{l\}$ stand for a set of instances and a set of literals respectively, whose both cardinalities are set by a uniform distribution and each member is randomly generated.

For cardinality constructors such as $minCardinality$, $maxCardinality$, $Cardinality$, $minQualifiedCardinality$, $maxQualifiedCardinality$, $qualifiedCardinality$, since the involved integer value should be positive and 1 is the most common value in the real world, I apply the Gaussian distribution with the mean being 1, the standard deviation being 0.5 (based on my experiences) and each generated value required to be greater than or equal to 1.

For data type constructors, since *String* and *Integer* are the most commonly used in the real world, I chose them to generate my data type statements. Both of them conform to a uniform distribution. The *Integer* is randomly selected from a range between 0 and 9. For instance, when the facet type F is selected, a range restriction of two integers is constructed. For those data type constructors such as

dataComplement with both *String* and *Integer* being able to be applied, I chose either with a probability of 0.5.

For those constructors such as $rdfs{:}subPropertyOf$ taking either OP or NDP as operands, I chose each of them with a probability of 0.5.

The four tables in Table 6.1 enable users to generate any OWL and OWL 2 sublanguage with different ontology constructors. In the parse tree, the root node is only selected from the AT table. Then, each other node is selected from the CT, OPT and DTT as appropriate. Note, the DTT table is only traversed for those constructors which can have the data type property NDP as operands such as *dataAllValuesFromRestriction*. The parse tree expansion terminates when either each branch of the tree ends with a leaf node type or the depth of the parse tree exceeds the given depth threshold, which is set 3 in my current implementation.

In LCDBM, the number of axioms generated in each domain ontology conforms to a Gaussian distribution with mean being 35 and standard deviation being 10. Note, when generating from the ontology profiles, my algorithms generates inconsistent ontologies about 10% of the time. Since it only takes several milliseconds to generate and check consistency of an ontology, we simply discard and regenerate any inconsistent ontologies.

For ontology mappings, since each mapping is essentially an axiom, I apply the same algorithm. Besides, I need to consider the linking strategy of different ontologies. This mechanism is basically the same as the work in [15]. I still create a directed graph of interlinked ontologies, where every edge is a map from a source ontology to a target ontology. Before the mapping creation, in order to guarantee the mapping connectivity and termination, I predetermine the number of terminal

nodes and randomly choose that number of domain ontologies. This prevents a non-terminal node from attaining a zero out-degree and maintain the connectivity. More details can be found in [15].

For every domain ontology, I generate a specified number of data sources. In LCDBM, this number is set by users according to their individual needs. For every source, a particular number of classes and properties are used for creating triples. They can be also controlled by specifying the relevant parameters in my configuration. To determine how many triples each source should have, I collected statistics from 200 randomly selected real-world Semantic Web documents. Since I found that the average number of triples in each result document is around 54.0 with a standard deviation of 163.9, I set the average number of triples in a generated source to be 50 by using a Gaussian distribution with mean 50 and standard deviation 165. In addition, based on my statistics of the ratio between the number of different URIs and the number of data sources in the Hawkeye knowledge base [22], I set the total number of different URIs in the synthetic data set to equal to the number of data sources times a factor around 2 in order to avoid the instance saturation during the source generation. In order to make the synthetic data set much closer to real world data, I ensure that each source is a connected graph, which more accurately reflects most real-world RDF files. To achieve this point, in my implementation, those instances that have already been used in the current source are chosen to generate new triples with a high priority.

147

Table 6.1: Axiom type, class, property and data type constructors.

Axiom Type Constructor (AT)				Class Constructor (CT)				
Constructors	**DL Syntax**	**Op1**	**Op2**	**Constructors**	**DL Syntax**	**Op1**	**Op2**	**Op3**
rdfs:subClassOf	$C1 \sqsubseteq C2$	C	C	allValuesFrom	$\forall P.C$	OP	C	
rdfs:subPropertyOf	$P1 \sqsubseteq P2$	OP, NDP	OP, NDP	someValuesFrom	$\exists P.C$	OP	C	
equivalentClass	$C1 \equiv C2$	C	C	intersectionOf	$C1 \sqcap C2$	C	C	
equivalentProperty	$P1 \equiv P2$	OP, NDP	OP, NDP	one of	$\{x_1,...,x_n\}$	{I}		
disjointWith	$C1 \sqsubseteq \neg C2$	C	C	unionOf	$C1 \sqcup C2$	C	C	
TransitiveProperty	$P^+ \sqsubseteq P$	NOP		complementOf	$\neg C$	C		
SymmetricProperty	$P \equiv (P^-)$	NOP		minCardinality	$\geq nP$	OP	INT	
FunctionalProperty	$T \sqsubseteq \leq 1P^+$	NOP		maxCardinality	$\leq nP$	OP	INT	
InverseFunctionalP.	$T \sqsubseteq \leq 1P$	NOP		Cardinality	$= nP$	OP	INT	
rdfs:domain	$\geq 1P \sqsubseteq C$	NOP, NDP	C	hasValue	$\exists P.\{x\}$	OP	I	
rdfs:range	$T \sqsubseteq \forall U.D$	NOP, NDP	C,D	namedClass	C			
disjointUnionOf	$C^- = C_1^- \sqcup$ $... \sqcup C_n^-$ and $C_i^- \sqcap C_j^- = \emptyset$	C	{C}	dataAllValues	$\forall NDP.D$	NDP	D	
ReflexiveProperty	$\forall x : x \in \Delta^I$ $\rightarrow (x,x) \in P$	NOP		dataSomeValues	$\exists NDP.D$	NDP	D	
IrreflexiveProperty	$\forall x : x \in \Delta^I$ $\rightarrow (x,x) \notin P$	NOP		minQualifiedCard.	$\geq nP$	OP,NDP	INT	C,D
AsymmetricProperty	$\forall x,y : (x,y) \in R$ $\rightarrow (y,x) \notin P$	NOP		maxQualifiedCard.	$\leq nP$	OP,NDP	INT	C,D
propertyDisjointWith	$P_1 \sqcap P_2 = \emptyset$	OP, NDP	OP, NDP	qualifiedCard.	$= nP$	OP,NDP	INT	C,D
hasSelf	$\{x|(x,x) \in P\}$	OP	TRUE	dataHasValue	$\exists NDP.\{l\}$	NDP	L	
Object Property Constructor (OPT)				Datatype Constructor (DTT)				
inverseOf	P^-	OP		dataComplement	$\neg D$	D		
propertyChainAxiom	$P_1 \circ P_2 \sqsubseteq P$	OP	OP, NDP	dataIntersection	$D1 \sqcap D2$	D	D	
namedProperty	P			dataUnionOf	$D_1 \sqcup D_2$	D	D	
				dataOneOf	$\{l_1,...,l_n\}$	L		
				namedDatatype	$\forall f(f \in F \rightarrow$ $f(D, NDP)$ $\in NDP)$	F		
				xsdDatatype				

6.1.2 Data-driven Query Generation

It is well-known that the RDF data format is by its very nature a graph. Therefore, a given semantic web knowledge base (KB) can be modeled as one big (possibly disconnected) graph. On the other hand, each SPARQL query is basically a subgraph and in order to guarantee each query has at least one answer, SPARQL queries should be generated from the subgraphs over the given big KB graph.

In addition, in order to generate reasonable synthetic SPARQL queries, we need to particularly consider two factors: the variable position in each QTP and the join patterns among different QTPs. According to the empricial study of real world SPARQL queries [2], 98.08% of SWDF (Semantic Web Dog Food) queries and 76.8% of DBPedia queries contain variables in either the subject position or the object position. 97.7% of SWDF queries and 99.77% of DBPedia queries are in join patterns of *Subject-Subject*, *Subject-Object* and *Object-Object*. Table 6.2 lists the surveyed queries' distributions of the query graph patterns measured by a serialization of the out-degree of each node of the graph. It also shows that most of the queries in DBPedia and SWDF contain one single triple pattern (66.5% and 97.5% respectively).

Therefore, I have designed and implemented a data-driven query generation algorithm. This algorithm first identifies a subgraph meeting the initial query configuration from a big KB graph. Then in order to deal with the variable position in either the subject or the object in each QTP, I replace some node values within the extracted subgraph with query variables in an empirical probability of 0.5. Finally, a conjunctive query based on the variable assigned graph can be generated. During this process, if the junction node of the subgraph is replaced by a query variable, this variable is counted as a join variable. As a result, we can generate different

Pattern	DBpedia	SWDF
10	66.512%	97.463%
3000	26.683%	0.106%
200	3.773%	1.024%
110	1.371%	0.482%
500000	0.701%	0.010%
2100	0.313%	0.412%
31000	0.195%	0.040%
40000	0.179%	0.020%
6000000	0.107%	0.001%
800000000	0.068%	0.000%
61000000	0.029%	0.001%
others	0.07%	0.420%

Table 6.2: Pattern graph out degree serialization of the real world SPARQL queries [2].

query join patterns in *Subject-Subject*, *Subject-Object* and *Object-Object*. Among the generated synthetic queries, 19% has a longest path length of 1, 77% has 2, and 4% has 3. Table 6.3 lists the distributions of the query graph patterns of the synthetic queries in terms of the out-degree of each node of the graph.

As shown by Table 6.3, the synthetic queries based on my statistics of the real world BTC data set in Section 6.2 cover most of the patterns in Table 6.2. Furthermore, the query generator can generate queries with more than one triple pattern in a greater percentage of 81% than 33.5% of DBPedia and 2.5% of SWDF. Here I hypothesize that the real-world SPARQL queries generally have more than one triple pattern, as opposed to only one triple pattern by Gallego's statistics on DBPedia and SWDF [2]. The query generation process is illustrated by Figure 6.2.

Assume we have constructed a subgraph shown in Figure 6.2(a) based on the

Pattern	Synthetic Queries
10	19%
200	21%
3000	14%
40000	20%
500000	8%
2100	3%
61000000	1%
70000000	5%
800000000	4%
9000000000	2%
10,0000000000	3%

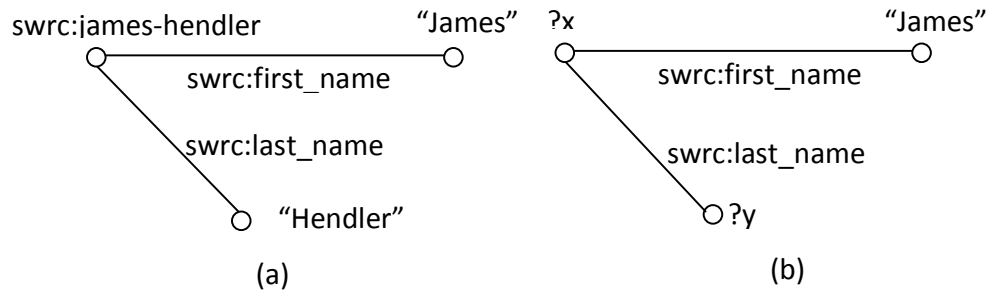Table 6.3: Pattern graph out degree serialization of the synthetic SPARQL queries.

Figure 6.2: Query graph.

KB graph. Within this graph, we could replace *swrc:james-hendler* and "*Hendler*" with the variables ?x and ?y respectively shown in Figure 6.2(b). Then, we could get the following SPARQL query:

SELECT ?x ?y WHERE { ?x *swrc:first_name* "*James*" . ?x *swrc:last_name* ?y . }

Figure 6.3 displays the algorithm. First, I randomly select one node *start* from *KBGraph* as the starting node to construct a query pattern graph *queryGraph* (Lines 2 and 3). Begin with *start*, I randomly select one edge that is starting with

151

Algorithm 9 Query generation

---

function GenerateQueries(KnowledgeBase *KBGraph*, int *numQTP*)

    return: a SPARQL query

    inputs: *KBGraph*, the given Knowledge Base graph

           numQTP, # of query triple patterns in the generated query

1: Let *queryGraph* = {}

2: Let *start* = randomly select one node from *KBGraph*

3: add(*queryGraph*, *start*)

4: **while**(numEdges(*queryGraph*) < *numQTP*) **do**

5:      *Edge* = Randomly select one edge with subject or object "*start*" within *KBGraph*

6:      **if**(isContained(*Edge*, *queryGraph*)) **then**

7:          **continue**

8:      add(*queryGraph*, *the ending node "end" of Edge*)

9:      add(*queryGraph*, *Edge*)

10:     With probability *P*, replace "*end*" with a variable

11:     *start* = Randomly select one node from queryGraph

12: **for each** edge *e* in *queryGraph* **do**

13:     **if**(hasNoVars(*e*)) **then**

14:         Randomly replace one node of *e* with a variable

15: Let *sparqlquery* = formQuery(*queryGraph*)

16: **return** *sparqlquery*

Figure 6.3: Graph-based query generation algorithm.

start and not contained in *queryGraph* and then add this edge with its ending node *end* into the *queryGraph* (Lines 5-9). If the selected edge is already in *queryGraph*, I skip this iteration and go select another edge that is not selected before (Lines 6 and 7). Then, I replace *end* with a new variable in the probability P (Line 10) and update *start* (Line 11). In LCDBM, the default value of P is set 0.5. This process is iterated until the *queryGraph* includes *numQTP* edges (Line 4). By this step, I have successfully constructed one query pattern graph. Next step, I will check if each edge e in *queryGraph* contains at least one variable (Lines 13 and 14). If not, I randomly replace one node of e with a new variable (Line 14). With the variable-assigned *queryGraph*, a SPARQL query can be generated and returned (Lines 15 and 16). Note, if the junction node of *queryGraph* is replaced by a query variable, this variable is counted as a join variable. In order to achieve the scalability, the *KBGraph* may actually be a subset of the original KB.

6.2 Real World Data Set

In all my experiments, I used a subset of the BTC 2009 data set to do the large scale evaluation. Much of this data set comes from the Linked Open Data Project Cloud. I have chosen four collections, as summarized in Table 6.4, with a total of 97,876,622 triples. Using the provenance information that records the websites where the triples were extracted in the BTC, I recreated local N3 versions of the original files from the BTC resulting in 20,332,701 data sources. The size of these data sources varies from roughly 5 to 50 triples each. As a result, their physical size in the disk space is around 42GB.

Data Collections	Namespace	# of Sources	# of Triples
http://data.sem-anticweb.org/	swrc	26,827	205,366
http://sws.geo-names.org/	geonames	2,311,282	14,140,670
http://dbpedia.org	dbpedia	10,779,307	55,264,775
http://dblp.rkb-explorer.com	akt	7,215,285	28,265,811
Total		20,332,701	97,876,622

Table 6.4: Data sources selected from the BTC 2009 dataset.

The involved ontologies in my selected data set are listed in Table 6.5. In order to integrate the four heterogeneous collections, I manually created some mapping ontologies, primarily using $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $owl{:}equivalentClass$ and $owl{:}equivalentProperty$ axioms (these schemas do not have any meaningful alignments that are more complex). The full set of ontology mappings of these ontologies is shown in Table 6.6.

My term index construction time over the selected data set is around 58 hours and its size is around 18GB. Each document takes around 10ms on average to be indexed. The Lucene configurations are 1500MB for RAMBufferSize and 1000 for MergeFactor, which are the best tradeoff between index building and searching for my experiments.

Ontology	Namespace	Mapped Ontology(ies)
DBpedia	http://dbpedia.org/ontology/	AKT, SWRC, GEONAMES, FOAF
AKT	http://www.aktors.org/ontology/ portal	DBpedia, SWRC
SWRC	http://swrc.ontoware.org/ontology	AKT, DBpedia, SWC
SWC	http://data.semanticweb.org/ ns/swc/ontology	SWRC
FOAF	http://xmlns.com/foaf/0.1/	AKT, DBpedia, SWRC
GEONAMES	http://www.geonames.org/ ontology	DBpedia

Table 6.5: Ontologies for the selected data sources

6.3 Evaluated Algorithms

I have conducted three sets of experiments to evaluate the performance of my four algorithms. In Table 6.7, I have summarized the various algorithms that were put under test in different experiments.

6.4 The Non-structure Algorithm Evaluation

6.4.1 Heterogeneity Evaluation

This experiment tests the hypothesis that the term index used by the non-structure algorithm is superior to the relevance file indices used in OBII-GNS proposed by Qasem et al [63]. Compared to the non-structure algorithm, OBII-GNS applies an indexing schema by creating content summary files (relevance statements), which seems like an unnecessary burden that lessens the likelihood that the approach will be adopted. In addition, OBII-GNS frequently selects sources that do not contribute to the eventual results. In the following parts, I will use IR and REL to respectively

| Mapping Ontology	Mapping Axioms
map_dbpedia_akt.owl	$dbpedia{:}Person \equiv akt{:}Person$, $dbpedia{:}Politician \sqsubseteq akt{:}Person$
map_dbpedia_geonames.owl	$geonames{:}Feature \sqsubseteq dbpedia{:}PopulatedPlace$
map_foaf_atk.owl	$foaf{:}name \equiv akt{:}full\text{-}name$, $foaf{:}Person \equiv akt{:}Person$
map_foaf_dbpedia.owl	$foaf{:}Person \equiv akt{:}Person$, $foaf{:}name \equiv dbpedia{:}name$
map_swrc_akt.owl	$swrc{:}Employee \sqsubseteq akt{:}Person$, $swrc{:}Person \equiv akt{:}Person$, $akt{:}Student \sqsubseteq swrc{:}Person$, $swrc{:}affiliation \sqsubseteq akt{:}has\text{-}affiliation$, $swrc{:}author \equiv akt{:}has\text{-}author$, $swrc{:}title \equiv akt{:}has\text{-}title$
map_swrc_dbpedia.owl	$swrc{:}Organization \sqsubseteq dbpedia{:}Organisation$

Table 6.6: Mapping ontologies for the selected data sources

differentiate the term index and the relevance file index. The experimental data set is synthetic and generated by LCDBM. The LCDBM configurations are 50 ontologies, 1000 data source sources, and a diameter of 6, meaning that longest sequence of mapping ontologies between any two domain ontologies was 6.

In this experiment, the IR index size is 10.8MB. The time to construct the IR index is 5,094ms, while it takes 14,593ms to construct the REL index. I issued 200 random queries and computed averages for all metrics mentioned here. Figure 6.4(a) displays the average number of results and average number of selected sources for each query. Observe that IR is more selective than REL in source selection but the query answers are still guaranteed to be the same by checking the query answers of both systems. In this result, the IR method selects 20-25% fewer sources than the REL method without losing any completeness.

Figure 6.4(b) compares the response time of both systems, and breaks out the time to reformulate the query (reform-Time), time to select sources (selectTime),
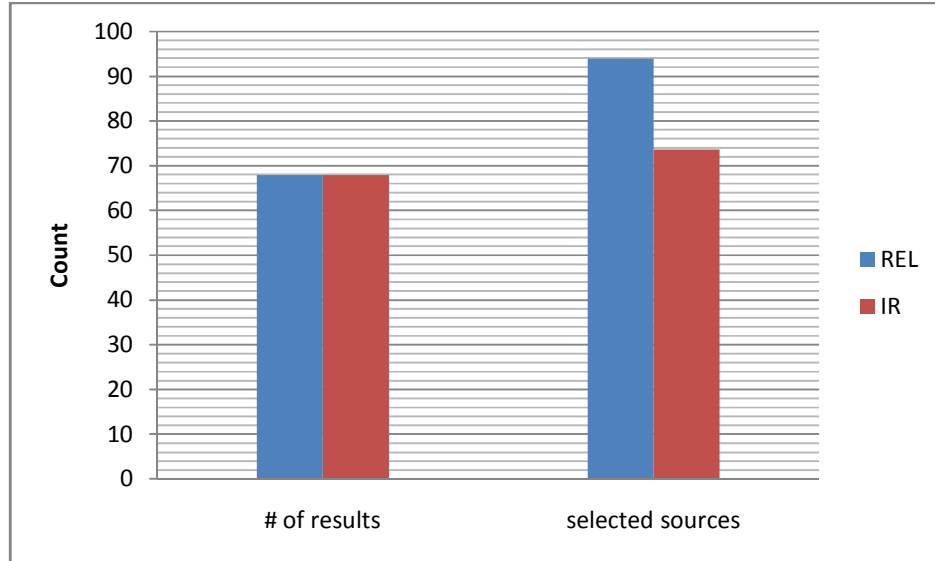
(a)
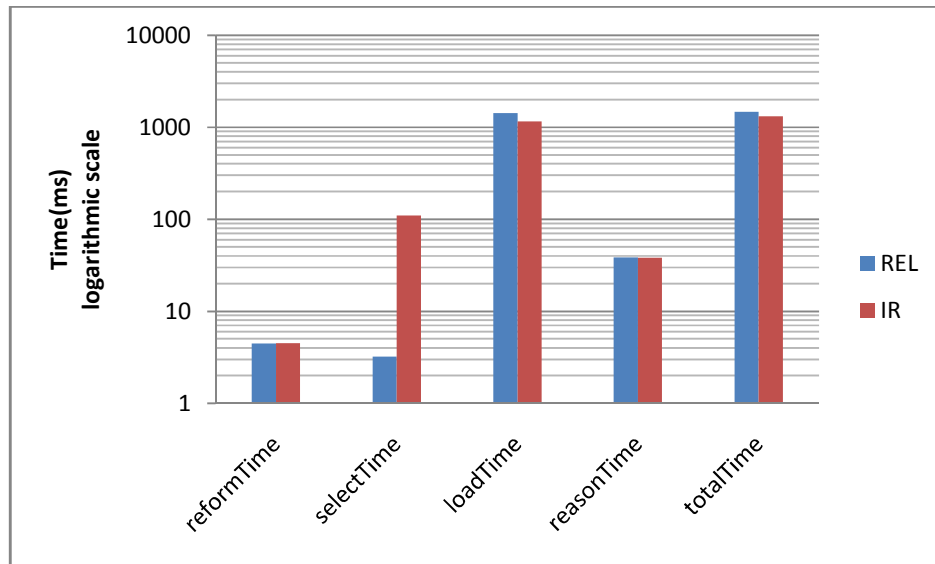
(b)

Figure 6.4: Source selection and response time of IR and REL

Algorithm	Description	Discussed in
Non-structure	Does a term index lookup for each subgoal and loads all relevant sources to answer queries.	Section 4.2
Flat-structure	Greedily collect relevant sources for each query rewrite of the conjunctive query to answer queries.	Section 4.3
Tree-structure	Greedily collect relevant sources by traversing the rule-goal tree of the origianl query to answer queries.	Section 4.4
Tree-structure (cycle w/o sameAs)	The improved tree-structure algorithm to handle cyclic axioms without equality reasoning	Section 5.2.1
Tree-structure (rewrite)	The improved tree-structure algorithm to handle cyclic axioms with equality reasoning by *owl*:*sameAs* rewriting.	Section 5.2.2
Tree-structure (cycle)	The improved tree-structure algorithm to handle cyclic axioms with equality reasoning optimization.	Section 5.2.2

Table 6.7: Algorithms Under Evaluation

time to load sources from local disk files (loadTime) and time spent by the KAON2 reasoner (reasonTime). The y axis is in logarithmic scale. The key observation is that the totalTime of IR being 1.31s is around 10% smaller than that of REL being 1.48s. The reason is that in both systems, loading sources is the dominant system cost, so fewer sources selected result in big gains. In my experiments, the number of selected sources for IR and REL are 73.58 and 93.92 respectively. It should be mentioned the IR system has a worse select time being 110.55ms than REL with 3.21ms. This is mainly because the REL system uses a memory-based index, while IR uses a disk-based index to achieve greater scalability.

6.4.2 Large Scale Evaluation

Based on BTC data set, I designed eight queries with different constant constraints including constant URIs and literals to evaluate the non-structure algorithm (Table 6.8). The number of reformulated query terms for each query is determined by the mapping ontologies and local axioms defined for the selected data sources. Figure 6.5 gives one query reformulation tree for instance Q4. Figure 6.6 shows the performance of the non-structure algorithm for answering these eight queries. Table 6.9 shows the source selectivity of the non-structure algorithm for the given eight queries by triple/document selecitivity, which is the ratio of the number of selected triples/documents over the total number of the triples/documents.

In this experiment, since the non-structure algorithm does not yet select all relevant sources with *owl*:*sameAs* information, I assume an environment where any relevant *owl*:*sameAs* information is already supplied to the **Reasoner**. I do this by initializing the KB with the necessary *owl*:*sameAs* statements.

Through the experimental results, we can have the following three observations:

- The first observation is that the non-structure algorithm is quite selective for the designed queries in both triple and document selectivity, as shown by Table 6.9. In this result, both triple and document selectivity are less than 0.1% of all triples and documents collected.

- The second observation is that the non-structure algorithm can scale to real world data with reasonable reformTime, selectTime, loadTime, reasoning time and totalTime, as shown in Figure 6.6 (in logarithmic scale) under the designed queries.

| Query	Query string	# of Reformulated query terms
1	?person dbpedia:name "James A. Hendler" .	6
2	?person akt:full-name "Abir Qasem" .	6
3	?paper swrc:author swrc:abir-qasem . ?paper swrc:author swrc:jeff-heflin .	4
4	?person akt:full-name "Abir Qasem" . ?person swrc:affiliation swrc:lehigh-university .	11
5	?person swrc:affiliation swrc:lehigh-university .	5
6	?person akt:full-name "Jeff Heflin" . ?person swrc:affiliation ?org .	11
7	dbpedia:Gargantilla dbpedia:subdivisionName ?name . ?ground dbpedia:ground ?name . ?dbpedia:Almendral dbpedia:subdivisionName ?name .	5
8	swrc:uwe-assmann foaf:based_year ?year . ?country dbpedia:countryofbirth ?year . ?dbpedia:BeFour dbpedia:origin ?year .	5

Table 6.8: Test queries
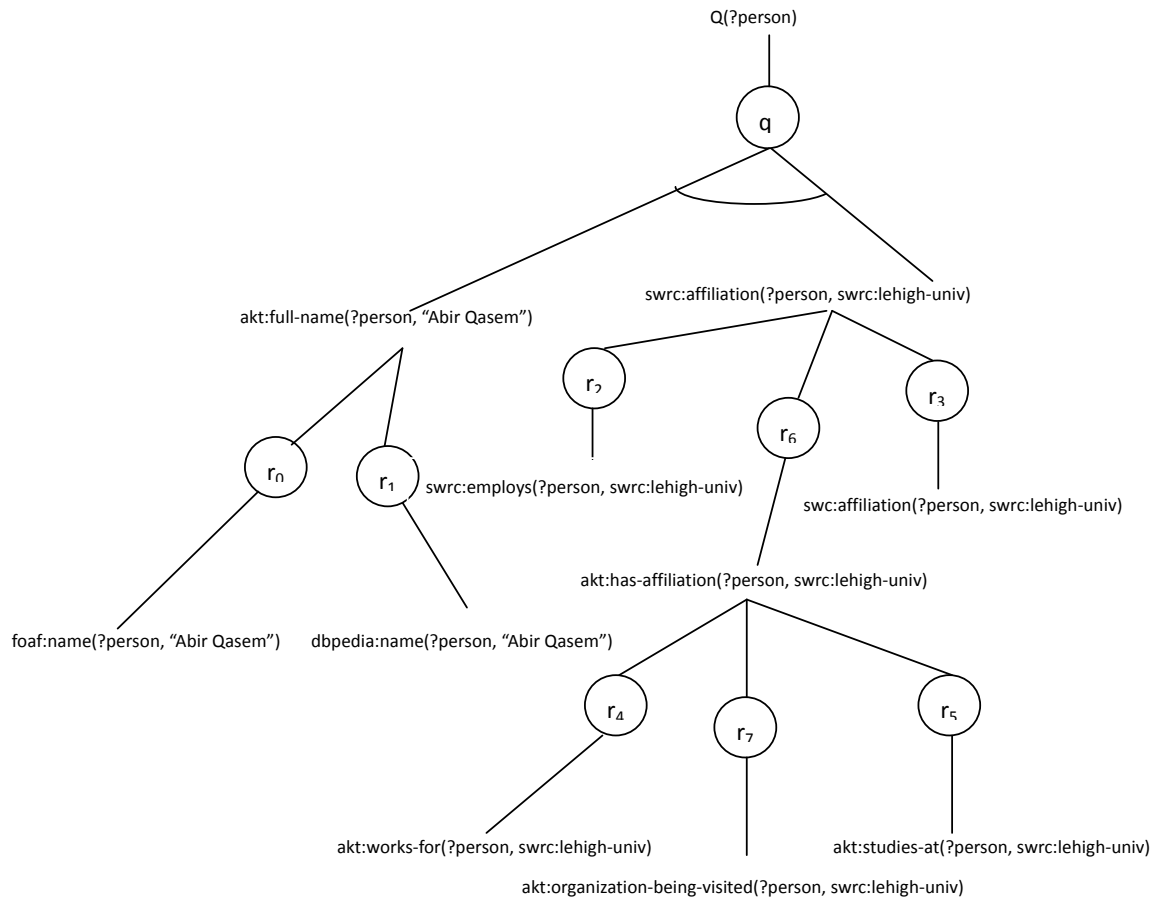
Figure 6.5: The query reformulation tree of the query Q4

Query	# of Results	# of Selected triples	# of Selected documents
1	142	715	143
2	11	36	9
3	2	46	9
4	7	172	29
5	15	163	20
6	16	25342	5069
7	12	24052	5011
8	328	26031	5006

Table 6.9: Source selectivity
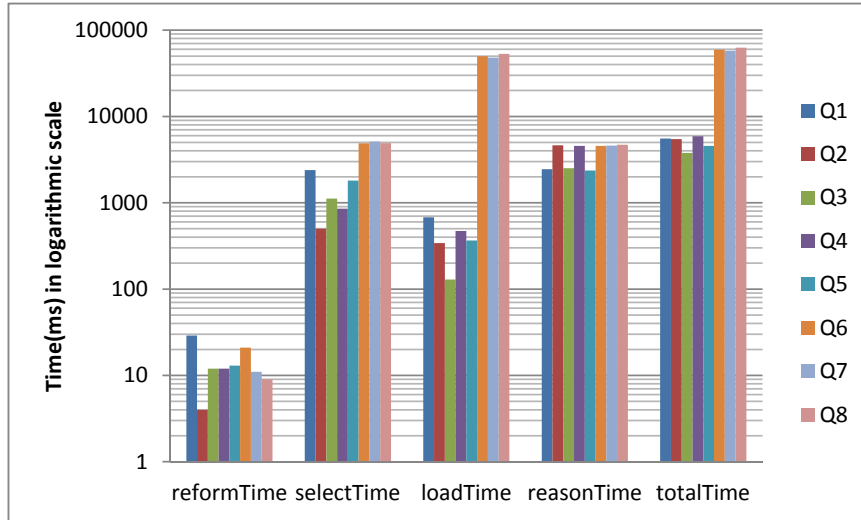
Figure 6.6: Performance of the non-structure algorithm over BTC

- The third observation is that the non-structure algorithm performs better for queries Q1, Q2, Q3, Q4 and Q5 with selective terms such as "James A. Hendler", "Abir Qasem" and "Jeff Heflin" than Q6, Q7 and Q8. This is because these terms make the algorithm select fewer sources. For those queries without selective terms such as Q6, Q7 and Q8 having a triple with two variables, the non-structure algorithm performs worse.

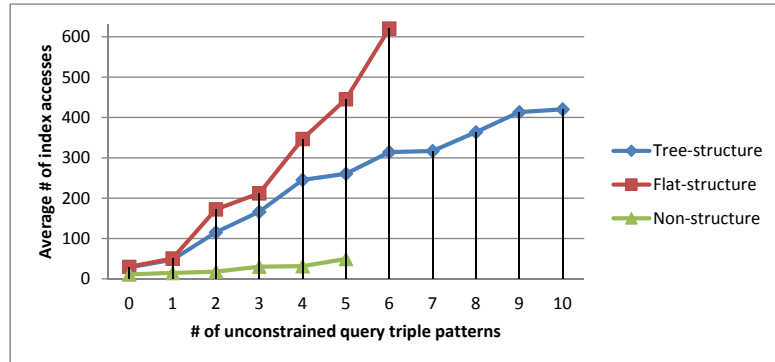## 6.5 The Tree-structure and Flat-structure Evaluation

6.5.1 Heterogeneity Evaluation

The experimental data set is still generated by LCDBM, whose configurations are 20 ontologies, 8000 data sources, and a diameter of 6, meaning that the longest sequence of mapping ontologies between any two domain ontologies is six. It took 21.5 seconds to build the 75.3MB index. I issued 240 random queries, which are grouped by the number of unconstrained QTPs from 0 to 10. An unconstrained QTP is defined to be one with variables for both its subject or object, or with the rdf:type predicate and a constant object. For each group, I computed the average query response time, average number of selected sources and average number of index accesses. Due to the exponential increase in query response time, I only executed queries with up to 5 and 6 unconstrained QTPs for the non-structure algorithm and flat-structure algorithm respectively.

Figure 6.7(a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs. From this result, we can see that the tree-structure algorithm and flat-structure algorithm are faster than the non-structure algorithm. The reason is that unconstrained QTPs are typically the least selective; thus, the more unconstrained QTPs there are, the more opportunities there are for the two optimization algorithms to use constraints to enhance the selectivity of goals. However, the benefits of the tree-structure algorithm become really noticeable for 5 or more unconstrained QTPs; in this situation the flat-structure algorithm begins to reveal exponential behavior while the tree-structure algorithm
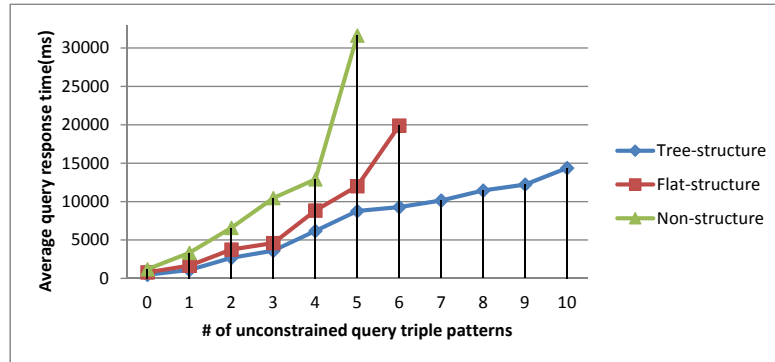
(a)

(b)
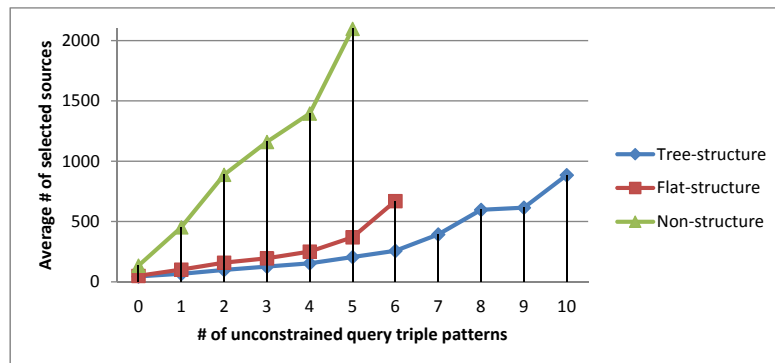
(c)

Figure 6.7: Heterogeneity experimental results. Average query response time (a), index accesses (b) and number of selected sources (c) as the number of unconstrained QTPs varies.

# of unconstrained QTPs	Flat-structure	Tree-structure		
	# of query rewrites	depth	branch factor	# of nodes
0	23	78	40	39
1	48	115	59	57
2	135	217	111	108
3	196	340	185	170
4	313	405	217	202
5	459	503	273	251
6	693	531	297	275
7		557	306	278
8		579	328	294
9		641	352	320
10		740	414	370

Table 6.10: Statistics of the flat-structure query rewrites and the tree-structure tree depth, branch factor and number of nodes.

remains linear. This is because complex mapping ontologies can lead to a number of conjunctive query rewrites that is exponential in the size of the query, as shown in Table 6.10.

Figure 6.7(b) shows how each algorithm's average number of index accesses is affected by the number of unconstrained QTPs. Note the index is stored on disk and is optimized for fast lookups, but a large number of accesses can have a noticeable impact on performance. From this result, we can see that the tree-structure and flat-structure algorithms require more index accesses than the non-structure algorithm: for 5 unconstrained QTPs they require 5.3x and 9.1x more accesses, respectively. This is because both algorithms take into account the query structure information while solving the original query and might need several index lookups for the same

query subgoal but using different substitutions. However, the tree-structure algorithm has 58% fewer index accesses than the flat-structure algorithm. The reason is that when using the flat-structure algorithm, one QTP can appear in multiple query rewritings and receive constraints from different sets of siblings representing different rewrites, while in the tree-structure algorithm the constraints of a sibling already consider all possible rewrites of the sibling.

Figure 6.7(c) shows how the number of unconstrained QTPs impacts the average number of selected sources for each algorithm. From this result, we can see the selectivity of the tree-structure and the flat-structure algorithms are roughly linear, while the non-structure algorithm is exponential in the number of unconstrained QTPs. Furthermore, the tree-structure algorithm has a gentler slope for its source selectivity than the flat-structure algorithm. Note, loading sources is the primary bottleneck of the system, since it requires that triples be read from the disk or network. The similar trends in Figure 6.7(a) and Figure 6.7(c) reflect the importance of source selectivity to overall query response time.

6.5.2 Large Scale Evaluation

Based on the BTC data set, this experiment aims to compare the scalabilities of the tree-structure algorithm, the flat-structure algorithm and the non-structure algorithm. However, because the non-structure algorithm does not propagate constant constraints when answering queries, it cannot scale to the BTC data set since most of the synthetic queries have at least one unconstrained QTP. For example, consider the query $Q:\{\langle ?x_0, swrc:affiliation, "lehigh-univ"\rangle.\langle ?x_2, akt:has-$
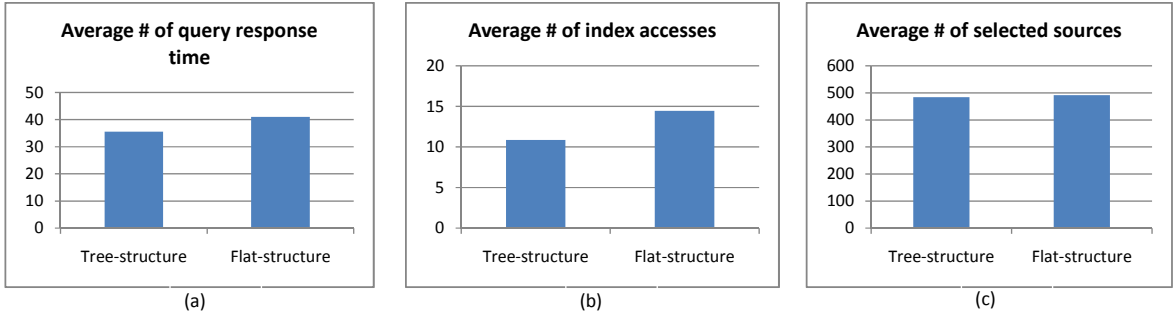
Figure 6.8: BTC data set experimental results of the tree-structure and flat-structure algorithms.

$title$, "$Hawkeye$"$\rangle$.$\langle?x_2, foaf{:}maker, ?x_0\rangle$.$\langle?x_0, akt{:}full-name, ?x_1\rangle\}$. For the non-structure algorithm, the number of sources that can potentially contribute to solving $\langle?x_2, foaf{:}maker, ?x_0\rangle$ is 3,485,607, which is far too many to load into a memory-based reasoner. Even though some reasoners can load this amount of data as long as the system has 3GB of memory, load times are typically in the 7 hours range, which is clearly unsuitable for real-time queries. However, the tree-structure and flat-structure algorithms can easily handle this query because the number of sources for the same QTP becomes 114 for instance after join constants are considered. Thus, I only compare the tree-structure algorithm to the flat-structure algorithm.

I executed 150 synthetic queries with at most 10 QTPs and computed the same metrics as for the prior experiment. As shown in Figure 6.8(a), the average query response time of the tree-structure algorithm is 35 seconds, which is a 13% improvement over the flat-structure algorithm. At the same time, it has 25% fewer index accesses as shown in Figure 6.8(b). Figure 6.8(c) shows that both algorithms select on average between 450 and 500 sources, and the tree-structure algorithm only shows a 1.6% improvement over the flat-structure algorithm. I attribute this to the fact that the semantic mappings of the BTC experiment are not as complex as those

for the synthetic data set, which leads to a small number of rewrites for each query when there are potentially many rewrites for a query.
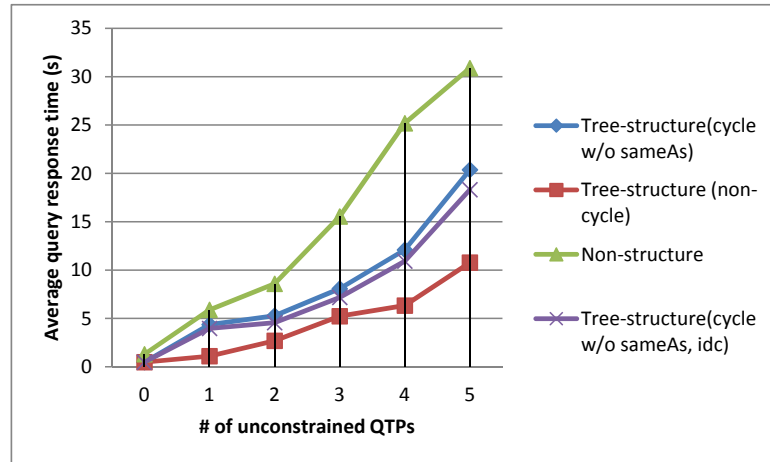
6.6 The Cyclic Axiom Handling Algorithm Evaluation

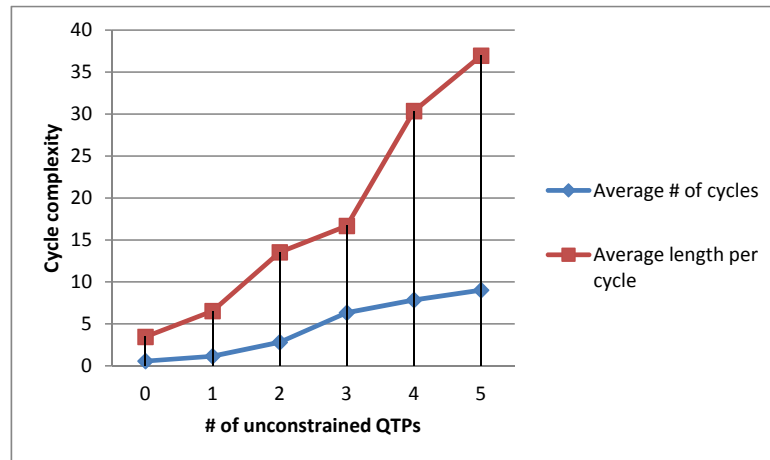### 6.6.1 Heterogeneity Evaluation

In this section, I conducted two separate experiments. The first aims to compare the tree-structure algorithm with the cycle handling algorithm without equality reasoning (tree-structure (cycle w/o sameAs)) to the tree-structure algorithm without the cycle handling (tree-structure (non-cycle)) and the non-structure algorithm. The second aims to compare the tree-structure algorithm with the cycle handling including equality reasoning optimization (tree-structure (cycle)) to the tree-structure algorithm without cycle handling (tree-structure (non-cycle)) and the tree-structure algorithm with *owl*:*sameAs* rewriting (tree-structure (rewrite)). In both experiments, since many URIs in either the synthetic dataset or the BTC data set have the same name space, I have applied the id compression optimization by replacing their name spaces with a number. As a result, the boolean query lengths and index size can be reduced. The index compression rates for the synthetic data set and the BTC data set are 30% and 15.7% respectively. In the evaluated algorithms, the id compression optimization is denoted "idc".

(a)

(b)

Figure 6.9: Cyclic axiom handling algorithm w/o *owl:sameAs* experimental results. Average query response time (a) and cyclic axiom complexity (b) as the number of unconstrained QTPs varies.

Cyclic Axioms Without Equality Reasoning

In this experiment, I issued 120 random queries to the synthetic data set to measure the cycle handling algorithm with the increasing cycle complexity, which is related with two factors: the average number of cycles per query and the average length per cycle. In addition, since the cycle complexity increases with the number of unconstrained QTPs, I group the 120 test queries by the number of unconstrained QTPs (from 0 to 5). In the metrics, I computed the average query response time and the cycle complexity. The experimental results are shown in Figure 6.9.

Figure 6.9(a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs with cycle complexity increasing. From this result, we can see that the tree-structure (cycle w/o sameAs) and tree-structure (cycle w/o sameAs, idc) algorithms are faster than the non-structure algorithm. The reason is that unconstrained QTPs are typically the least selective; thus, the more unconstrained QTPs there are, the more opportunities there are for the tree-structure (cycle w/o sameAs) and tree-structure (cycle w/o sameAs, idc) optimization algorithms to use constraints to enhance the selectivity of goals. Due to the additional cycle handling, the tree-structure (cycle w/o sameAs) and tree-structure (cycle w/o sameAs, idc) algorithms are slower than the tree-structure algorithm (non-cycle), but they bring us more complete results as shown in Figure 6.10(b). In addition, the tree-structure (cycle w/o sameAs, idc) performs 10% better than the tree-structure (cycle w/o sameAs) algorithm brought by the average boolean query length compressed by around 12% because of the id compression optimization.

Figure 6.9(b) shows how the cyclic axiom complexity changes with the increasing number of unconstrained QTPs. As shown in this figure, the most complex test

queries have 5 unconstrained QTPs, 10 cyclic axioms per query reformulation and 35.5 nodes per cyclic axiom. As a result, the average cycle apperance percentage in the whole test query set is 4.8%. On the other hand, according to Wang's survey [79] of 1275 ontologies, there are only 0.9% cyclic axioms (mainly transitive properties), which is much less than the precentage of cyclic axioms in my experiment. Thus, we can see that my cyclic axiom handling algorithm can well scale to the real world.
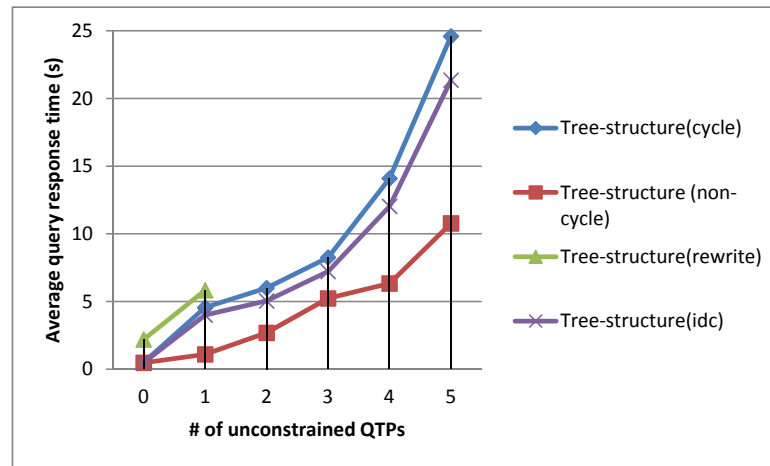
Equality Reasoning Evaluation

In this experiment, I configure LCDBM to generate the *owl*:*sameAs* triples in the synthetic data set based on my *owl*:*sameAs* Sindice statistics of randomly issuing one term query and taking the top 1000 returned sources as samples. The ratio of sources containing *owl*:*sameAs* is 27.1%. Thus, the LCDBM generates *owl*:*sameAs* triples in a ratio of 27.1% of the total number of triples. As a result, the number of *owl*:*sameAs* triples is 2,765 of 45,673 total triples in 8000 sources. Furthermore, for each instance involved into the *owl*:*sameAs* triple, according to my experiences, I take a probability of 0.1 to select it from the set of all generated instances in the whole data set and a probability of 0.9 to select it from the set of all generated instances in the current source. Thus, all of *owl*:*sameAs* triples in my data set are categorized into different equivalence classes. Each equivalence class is defined to be a set of instances that are equivalent to each other (explicitly or implicitly connected by *owl*:*sameAs*). In my experimental dataset, all *owl*:*sameAs* triples are categorized into 571 equivalence classes. The largest equivalence class contains 10 instances and the average equivalence class size is 3.7. Like the last experiment, I still issued 120 random queries and group them by the number of unconstrained

QTPs (from 0 to 5). In the metrics, I computed the average query response time and the query completeness. The experimental results are shown in Figure 6.10.

Figure 6.10(a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs. From this result, we can see that the tree-structure (cycle) and tree-structure (cycle, idc) algorithms are faster and with better scalabilities than the tree-structure (rewrite) algorithm. The reason is that the tree-structure (rewrite) algorithm suffers from the explosive combination of query answers due to the introduction of *owl*:*sameAs* QTP as illustrated in Section 5.2.2. Consequently, the tree-structure (rewrite) algorithm can only scale up to queries with at most one unconstrained QTP in my experiments because **Reasoner** starts to get stuck by those intermediate queries including *owl*:*sameAs* QTPs. Due to the additional cycle and *owl*:*sameAs* handling, the tree-structure (cycle) and tree-structure (cycle, idc) algorithms are slower than the tree-structure algorithm (non-cycle), but they bring us more complete results as shown in Figure 6.10(b). Similar to the tree-structure (cycle w/o sameAs, idc) algorithm, the tree-structure (cycle, idc) algorithm also performs around 10% better than the tree-structure (cycle) algorithm because of the id compression optimization.

Figure 6.10(b) shows the comparison of the completeness of the tree-structure (cycle), the tree-structure (cycle, idc), the tree-structure (non-cycle) and the tree-structure (rewrite) algorithms. I take the results of non-structure algorithm as ground truth because it is complete. The percentage number on top of each tree-structure bar is the completeness of the tree-structure (non-cycle) algorithm in the current point. From this result, we can see that the tree-structure (cycle), tree-structure (cycle, idc) and tree-structure (rewrite) algorithms are more complete than

(a)
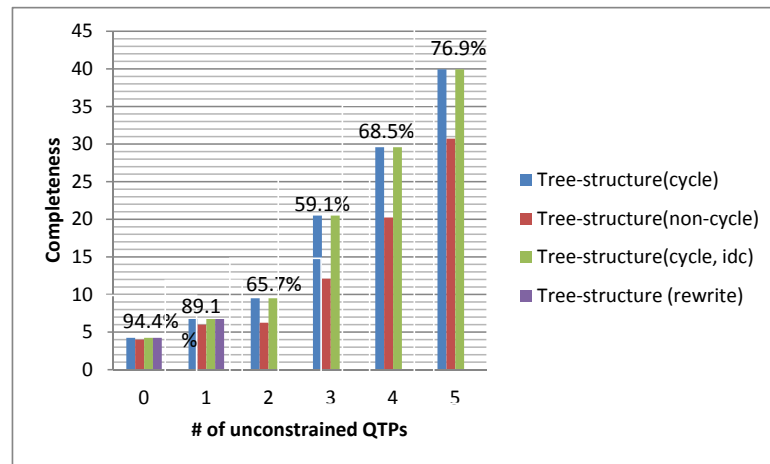
(b)

Figure 6.10: Equality reasoning experimental results. Average query response time (a) and query completeness (b) as the number of unconstrained QTPs varies.

the tree-structure (non-cycle) algorithm. Furthermore, the tree-structure (cycle) algorithm has the same completeness as the tree-structure (rewrite) algorithm but gains better query response time and scalability (as shown in Figure 6.10(a)).

**Source Loading and Query Execution Trade-off Evaluation**

Since my system dynamically selects relevant sources and answers queries, compared to the centralized systems that preload all sources into their repositories and then answer queries, it is meaningful to evaluate the trade-off capability of my system on the source loading cost and query execution cost by comparing my tree-structure (cycle) algorithm to centralized systems. In this section, I select KAON2 and OWLIM [38] as my target systems because KAON2 has been employed to be my query engine in my system implementation and OWLIM scales very well in both reasoning capability and data scalability [47]. The experimental data set contains 20 ontologies, 8000 data sources, whose size is 121M. I issued 120 random queries. The experimental results are shown in Figure 6.11.

Figure 6.11(a) shows that the tree-structure (cycle) algorithm needs more query response time than both OWLIM and KAON2 because it dynamically selects relevant sources and loads them on the fly. According to the results, the source loading cost percentage w.r.t. the whole query answering cost of the tree-structure (cycle) algorithm is around 93.6%. Figure 6.11(b) shows the average source loading time. For OWLIM and KAON2, the average source loading time is calculated through the time of loading all sources divided by the number of test queries. The reason we amortized the loading time cost of OWLIM and KAON2 is that the data sources can be loaded at once and then queries can be answered based on the loaded data.
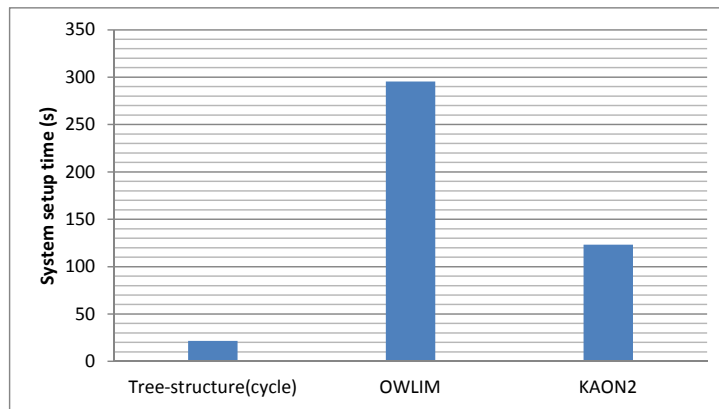
174
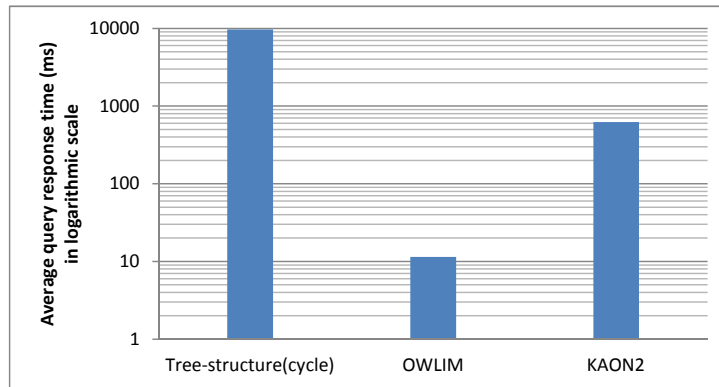
(a)
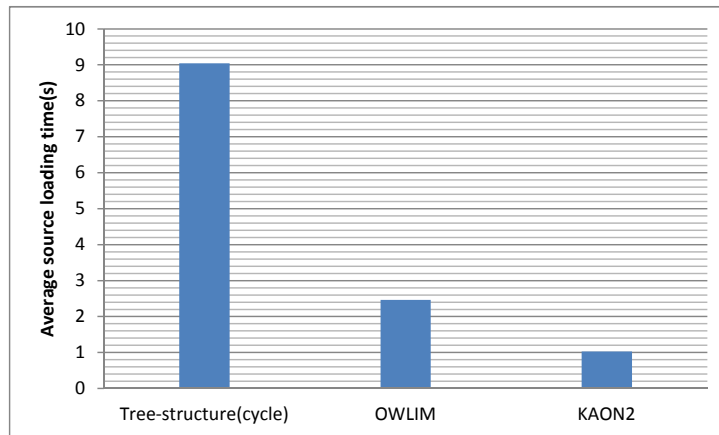
(b)

(c)

Figure 6.11: The tradeoff experimental results. Average query response time (a), source loading time (b) and system setup time (c).

Thus, we do not have to repeatedly load the data sources for each query. However, one drawback of this way is that the longer the loaded data sources are kept in the triple store, the more stale data the queries will be against. From Figure 6.11(b), we can see that OWLIM and KAON2 perform better than the tree-structure (cycle). The reason is that both OWLIM and KAON2 load all sources at once at the beginning, while the tree-structure (cycle) algorithm dynamically plans query execution and load relevant sources on the fly for each query. Figure 6.11(c) shows the three systems' setup time: the index creation time of the tree-structure (cycle) algorithm and the all source preloading time of both OWLIM and KAON2. As shown by the results, we can see that the tree-structure (cycle) algorithm has less system setup time (21.6s) than both OWLIM (295.529s) and KAON2 (121.182s). The reason is that both OWLIM and KAON2 needs time to do the materialization of reasoning over the loaded data. In addition, the reason of OWLIM having more system setup time than KAON2 is because OWLIM materializes its knowledge base into the disk while KAON2 is only memory-based.

6.6.2 Large Scale Evaluation

Since the non-structure cannot scale to the BTC data set, this experiment only compares the tree-structure family algorithms. I executed 150 synthetic queries with at most 10 QTPs. In this experiment, the *owl*:*sameAs* statements are from the BTC data set itself. The percentage of sources containing *owl*:*sameAs* is around 3%. In the metrics, I computed the average number of answers, average query response time, average number of selected sources and average index accesses of three algorithms: the tree-structure (cycle, idc), the tree-structure (cycle, non-idc)
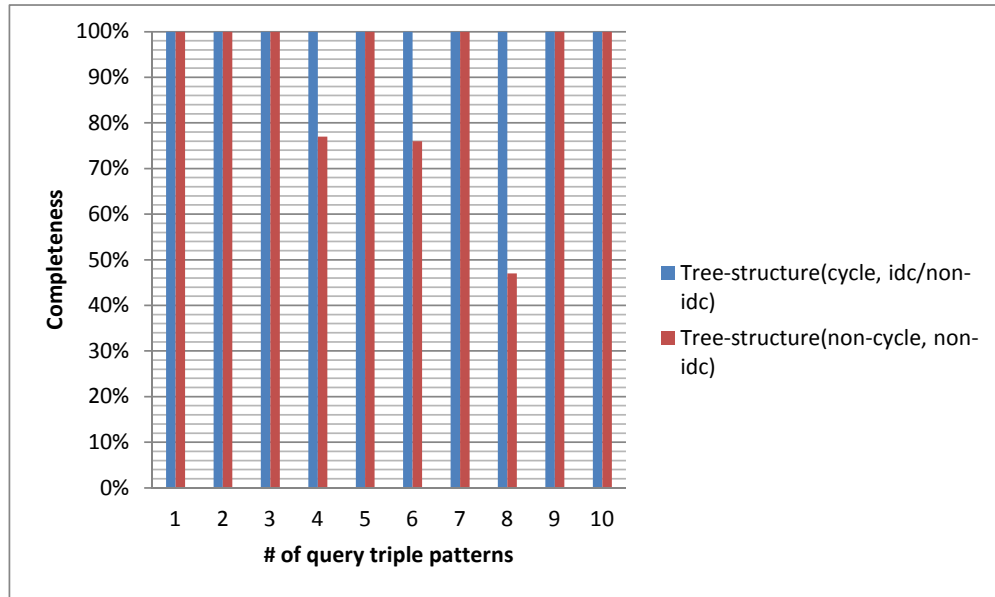
Figure 6.12: The number of results returned by the tree-structure family algorithms over BTC data set.

and the tree-structure(non-cycle, non-idc) algorithm. The experimental results are shown in Figure 6.12 and Figure 6.13. According to Figure 6.12, we can see that the tree-structure (cycle, idc) and the tree-structure (cycle, non-idc) algorithms has returned 26.8% more answers than the tree-structure (non-cycle, non-idc) algorithm because of the additional cycle process. Meanwhile, they only have small increases at the metrics of query response time, index accesses and the number of selected sources as shown by Figure 6.13. In particular, with the id compression optimization, the average query length can be compressed by 26%. As a result, the query response time of the tree-structure (cycle, idc) algorithm has gained around 20% improvement over the tree-structure (cycle, non-idc) algorithm and is only around 5% more than the tree-structure (non-cycle, non-idc) algorithm as shown by Figure 6.13 (a).
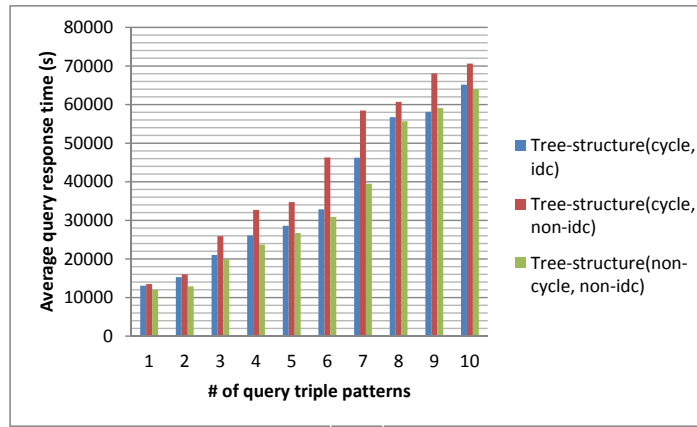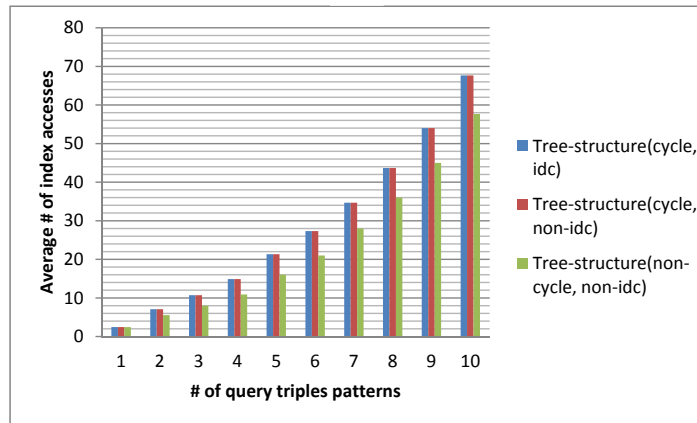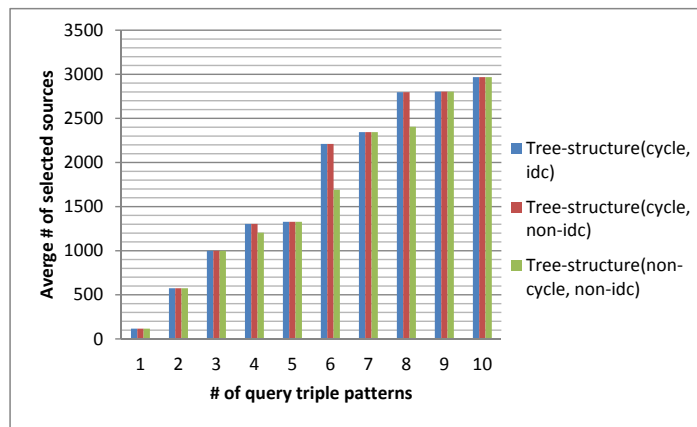
Figure 6.13: BTC data set experimental results of the tree-structure family algorithms. Average query response time (a), index accesses (b) and selected sources as the number of QTPs varies.

Chapter 7

Conclusion

7.1 Summary and Analysis

The birth of the Semantic Web drives the evolution of the Web as a global information space from a Web of documents to a Web of data, where not only documents but also data is linked. Semantic Web data typically exhibits features of heterogeneity, smallness, dynamicity and large scalability. Under such an environment, there is often the need to integrate the ontologies and their data sources and access them without regard to the heterogeneity and the dispersion of the ontologies. The traditional procedure to work with multiple, distributed linked data sources is to load the desired data into a local and centralized system and process queries in a centralized way against the merged data set. One representative solution is the Data Warehouse, which is a database used for reporting and data analysis [64]. The data stored in the data warehouse are uploaded from the operational systems, cleansed, transformed, and placed into the data warehouse or data mart according

to a schema, such as the star schema. The data marts store subsets of data from a warehouse. The star schema is a logical arrangement of tables in a multidimensional database. The goal of data warehouse is to integrate applications at the data level and create a centralized and unified view of enterprise data holdings. The typical data warehouse uses staging, integration, and access layers to house its key functions. The staging layer or staging database stores raw data extracted from each of the disparate source data systems. The integration layer integrates the disparate data sets by transforming the data from the staging layer often storing this transformed data in an operational data store database. The integrated data is then moved to the data warehouse database. The access layer helps users retrieve data.

However, centralized systems have many disadvantages. First, they will become stale unless they are frequently reloaded with fresh data. Second, they can require significant disk space, especially for those ones that use multiple indices to optimize queries. Finally, there may be legal or policy issues that prevent one from copying data or storing it in a centralized place. For this reason, I have designed and developed a federated Semantic Web query answering system. This system applies an automated mechanism for creating the index used in determining source relevance and employs a hybrid approach to answering queries that involves ideas from information retrieval, information integration and knowledge base systems. In particular, this dissertation makes original contributions to the following research problems.

I have designed and implemented an efficient, IR-inspired inverted index to integrate semantic web data sources and determine source relevance. This term index takes the full URIs of subjects, predicates and objects of RDF triples as tokens.

When the object is a literal, it tokenizes terms extracted from the literal. For each term in the term index, there is a posting list that records which documents contains which terms. As a result, the term index provides a function to determine source relevance for any given boolean query.

Based on the term index, I first designed and implemented a non-structure query answering algorithm. This algorithm takes a set of query subgoals as inputs and loads all relevant sources into a reasoner to solve queries on the fly. The initial experiments have shown that when using the term index, my system selects 20-25% fewer sources than in the relevance statements as stated in Section 6.4.1, without losing any completeness. Since loading sources is the dominant system cost, this makes the resulting system around 20% faster. However, because the term index only indicates if URIs or Literals are present in a document, specific answers to a subgoal of the given query cannot be calculated until the sources are physically accessed - an expensive operation given disk/network latency. In addition, in the real world, the number of sources related to a subgoal could be so large that it is impossible to load all of them into a reasoner to answer queries.

In order to overcome the drawbacks of the non-structure algorithm, I designed and implemented a flat-structure algorithm. Given a set of conjunctive query rewritings, this algorithm employs a greedy source selection strategy that prioritizes selective subgoals of the query and uses the sources that are relevant to these subgoals to provide constraints that could make other subgoals more selective. During this process, a selectivity based query execution plan is dynamically generated. In this way, the data sources will be incrementally collected and processed. Once sources are selected, I will load them into a reasoner to solve queries over these sources and their

corresponding ontologies. This algorithm can be combined with any query rewriting algorithm that produces a set of conjunctive subqueries. My experiment has shown that the flat-structure algorithm is superior to the non-structure algorithm with 60% better in the query response time, 70% better in the source selectivity and the ability to solve real world queries. However, when there is significant heterogeneity in the ontologies, synonymous ontology expressions can lead to an explosion in the number of query rewrites. Consequently, the flat-structure algorithm can slow the system down due to the processing of a large number of rewrites. In addition, the flat-structure algorithm suffers from the inability to use the full structure of query rewrites reduces the possible source selectivity of the query process.

In order to overcome the drawbacks of the flat-structure algorithm, I designed and implemented a tree-structure algorithm. Given a rule-goal tree (AND/OR graph) that expresses the reformulation of a conjunctive query, the tree-structure algorithm uses a bottom-up approach to estimating the selectivity of each node. It then prioritizes loading of selective nodes to generate a query execution plan on the fly and uses the information from these sources to further constrain other nodes. As with the flat-structure algorithm, a reasoner is employed to answer queries over the selected sources and their corresponding ontologies. My experiments have shown that the tree-structure algorithm is better in query response time and source selectivity than the flat-structure and non-structure algorithms. In particular, the benefits of the tree-structure algorithm become really noticeable for 6 or more unconstrained QTPs; in this situation the flat-structure algorithm begins to reveal exponential behavior while the tree-structure algorithm remains linear. In the index accesses, the tree-structure algorithm has 58% less than the flat-structure algorithm. However,

the tree-structure algorithm only guarantees completeness for acyclic OWLII axioms and will become incomplete when cyclic axioms are considered. In addition, it is incomplete when equality reasoning (*owl*:*sameAs*) is considered.

In order to handle the cyclic axioms including the equality reasoning (*owl*:*sameAs*), I further designed and implemented a dynamic cyclic axiom handling algorithm. By employing a cycle stack, this algorithm is able to dynamically computes the source collection fix point of cyclic axioms to generate a query execution plan on the fly. Due to the explosive combination of the answers to queries including *owl*:*sameAs* QTP, an equality reasoning optimization algorithm is employed to handle the source collection of *owl*:*sameAs*. My experiments have demonstrated that the cyclic axiom handling algorithm can effectively handle real world queries with cyclic axioms in around 36 seconds and meanwhile gurarantee the query completeness.

Besides the experimental evaluation, for all algorithms, I have also theoretically proved their correctness by two parts: the soundness and the completeness.

Table 7.1 summarizes the advantages and disadvantages of my proposed algorithms.

In summary, this dissertation provides a way to answer distributed queries in a web like environment that is large-scale and heterogeneous. It does so by using a term index to determine source relevance for a given query and applying a hybrid approach to answering queries that involves ideas from information retrieval, information integration and knowledge base systems. Till now, I have summarized the primary research contributions of this dissertation. In the next section, I will discuss several future directions of this work.

Algorithm	Advantages	Disadvantages
Non-structure	Better selectivity and faster than OBII-GNS algorithm; Complete; More expressive than the tree-structure family of algorithms.	Expensive source loading given disk/network latency; Cannot scale to the real world data set; Unconsider the structure relations among different subgoals.
Flat-structure	Better selectivity and faster than the non-structure algorithm; Can scale to the real world data set; More expressive than the the tree-structure family algorithms; Complete for any complete rewrite algorithm.	Complex ontologies leading to explosive query rewrites; Inability to use the full structure of query rewrites.
Tree-structure	Better selectivity and faster than the flat-structure and non-structure algorithms. Can scale to the real world data set; Complete for rewrites with AND/OR tree structure.	Incomplete for cyclic axioms and equality reasoning; Less expressive than the flat-structure and non-structure algorithms.
Tree-structure (cycle w/o sameAs)	Better selectivity and faster than the non-structure algorithm; Complete for rewrites with AND/OR graph structure without equality reasoning; Can scale to the real world data set.	Incomplete for equality reasoning; Less expressive than the flat-structure and non-structure algorithms; Slower than the tree-structure.
Tree-structure (rewrite)	Complete for rewrites with AND/OR graph structure with equality reasoning.	Double number of QTPs; The explosive combination of answers;Less expressive than the flat-structure and non-structure algorithms; Slower than the tree-structure.
Tree-structure (cycle)	Better selectivity and faster than the non-structure algorithm; Complete for rewrites with AND/OR graph structure with equality reasoning; Can scale to the real world data set.	Less expressive than the flat-structure and non-structure algorithms; Slower than the tree-structure.

Table 7.1: The advantages and disadvantages of the proposed algorithms.

7.2 Limitations and Future Work

The work presented in this dissertation may be extended in several promising directions.

7.2.1 Ontology Expressivity Extension

As demonstrated in Chapter 6, my algorithms work when both the domain ontologies and the map ontologies are expressed in OWLII, which is the subset of OWL DL that is common with GAV and LAV (Definition 1). However, in terms of GAV/LAV expressivity, other non-OWLII language constructors such as OWL 2 property composition can be also expressed in GAV/LAV. Thus, it can be added to OWLII without requiring any change to the source selection algorithms. Additionally, I would like to identify a fragment of OWL 2 that cannot be rewritten using GAV/LAV in order to identify my system expressivity. In such cases, my system can still preserve query semantics but will not harm soundness, since a sound reasoner is employed. An example is as follows:

TBox

$A(X) \sqsubseteq B(X) \sqcup C(X)$, which is beyond OWLII (Defintion 1).

ABox

a_1:A, a_2:A, a_3:A, a_1:B, a_2:B, a_3:B, a_4:C, a_5:C.

Given the above knowledge base a DL reasoner can solve a query $Q(X) \leftarrow A(X)$, whose answer set is $\{X/a_1, X/a_2, X/a_3\}$.

Then, if we rewrite $Q(X) \leftarrow A(X)$ into two rules: $Q(X) \rightarrow B(X)$ and $Q(X) \rightarrow C(X)$. The answer set to the rewritten queries is $\{X/a_1, X/a_2, X/a_3, X/a_4, X/a_5\}$.

As shown by the above example, even though the GAV/LAV query rewriting is not equivalent to the origianl query, my system can still find all answers to the original query. From an algorithm point of view, the main impact by such query rewriting is the increase of the rule-goal tree size.

7.2.2 Robustness against Stale Indexes

One challenge faced by my system is its dependence on the accuracy of the index when attempting to select the minimal number of potentially relevant sources. However, the Web changes, and refreshing the index can be expensive. According to my statistics using Sindice to compare the triple changes between the current and cached versions of 1000 different data sources, the ratio of sources with triples added and removed are 2.1% and 20.4% respectively in 25 days on average. The minimum time period is 19 days. The maximum time period is 55 days. Here, the cached version of one document in Sindice is the one that was saved on the Sindice server sometime before the current version of the same document. Thus, by tracking the changes of each document from the current version to the cached version, we can calculate the change rate of data sources in a time period.

My IR-inspired term index is robust in the removal of triples. In the worst case a source will be selected, but have nothing to contribute when the reasoning engine computes the final answer to the query. However, if triples are added to a source, my method could miss a source that is now relevant, resulting in an incomplete answer to the query. Of course, short of loading all sources that have changed since the index was built, there is no way to guarantee completeness in the face of dynamic data. However, it should be possible to identify sources the change frequently and

to identify change patterns in RDF documents that can be exploited to minimize the impact of these changes on completeness. Here are some hypotheses that might be able to be used:

- Data sources rarely change the ontologies from which they use terms.

- Data sources typically use instance names from a small set of namespaces, and rarely add triples involving instances from new namespaces

- Data sources that have many triples mentioning a particular instance are more likely to add additional triples about that instance, than data sources that only have one triple mentioning the instance.

If these hypotheses are correct, then one might choose to extend the existing source selection algorithm to include some additional sources based on them. For example, a source may contain many triples about $jsmith$, however it does not contain spouse information and thus does not match the goal $spouse(jsmith, x)$. However, we might still choose to select this source, especially if it uses other vocabulary from the same ontology that defines spouse, under the assumption that if spouse information become available, this source is likely to contain it.

7.2.3 Query Expressivity Extension

As stated in Chapter 3, my system focuses on conjunctive queries. However, it does not consider the reformulation of queries that allow constraints to be added to filter the query answers. In SPARQL terms, these are queries with FILTER applied. For example:

- $c{:}State(x) \wedge c{:}population(x,p) \wedge regex(x, \text{``}Penn\text{''})$.

- $b{:}Bill(x) \wedge b{:}date(x,d) \wedge d < 2011 - 12 - 31 \wedge d > 2011 - 01 - 01$.

Reformulating such queries needs to record the constraints information. One possible way is to label each subgoal with the corresponding constraints if the constrained variable appears in this subgoal. In the above example, the subgoals $c{:}State(x)$ and $c{:}population(x,p)$ should be labeled with the constraint $regex(x, \text{``}Penn\text{''})$. Then, during source selection, these constraints can be used as filters to improve the source selectivity. This is especially important in situations with a large number of distributed sources. The key point of this method is to design an efficient algorithm that can apply these filters to select potentially relevant sources at the minimum comparison cost.

Another thing I am still missing is how to reformulate queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. This corresponds to SPARQL queries with OPTIONAL applied. Note, for these queries, the OPTIONAL condition cannot help to increase source selectivity because solving the OPTIONAL graph patterns depends on the results of solving the basic graph patterns in SPARQL queries. Thus, I suggest that the OPTIONAL process is left to the reasoner after relevant source collection.

In addition, in order to avoid the computational challenges of higher-order logics, my system does not allow variables in the predicate position. This could be another possible extension.

7.2.4 Question Translation

Currently, my system takes conjunctive queries as inputs. One better alternative choice is to take queries expressed in natural language as input and then return answers from the available knowledge bases. In order to achieve this goal, ontology-based natural language understanding techniques [50] are required. This procedure generaly consists of two steps:

- Linguistic translation: translate the natural language described queries into linguistic triples. For example, the query "Who teaches Semantic Web?" is parsed into { ⟨*?p type Professor*⟩ . ⟨*?p teaches "Semantic Web"*⟩ .}.

- Relation similarity computation: convert the linguistic triples into SPARQL queries by mapping the linguistic terms in linguistic triples into ontological terms defined in either domain or mapping ontologies. For example, the above given linguistic triples are converted into the following SPARQL query:

  SELECT ?p WHERE { *?p rdf:type swat:Professor* . *?p swat:teaches* "Semantic Web" . },

  where the linguistic terms of "*type*", "*Professor*" and "*teaches*" are mapped onto the ontological terms of "*rdf:type*", "*swat:Professor*" and "*swat:teaches*" respectively. Here, the mapping relation can be computed using lexicons such as the well known WordNet or user's own lexicon together with string metrics, which is related to ontology alignment [71].

7.3   A Vision of the Semantic Web

The World Wide Web has radically altered the way we share knowledge by lowering the barrier to publishing and accessing documents as part of a global information space. The inarguable success of search engines may lead one to believe that the Web has reached its full potential as a global knowledge repository. However, the existing search technology is unable to meet our information needs due to its limitations of keyword matching and hypertext linking.

The birth of the Semantic Web helps the Web to reach its true potential by suggesting a way of extending the existing web with structure and providing a mechanism to specify formal semantics that are machine-readable and shareable. The main aim of the Semantic Web is to organize the information found in the Web in a better fashion and interconnect the various pieces of information so that they can be used for discovery, automatization, aggregation, and reuse from various, different and disparate applications, which have not been designed either to work together or to work with every different piece of information found in the Web. In recent years the Semantic Web has driven the Web to evolve from a global information space of linked documents to one where both documents and data are linked. Underpinning this evolution is a set of best practices for publishing and connecting structured data on the Web known as Linked Data. Just as the Web has brought about a revolution in the publication and consumption of documents, Linked Data has the potential to enable a revolution in how data is accessed and utilised. Within Linked Data, the heavy reasoning or the AI vision of the Semantic Web has been replaced by a networked and user-driven Semantic Web. This new view of the Semantic web will be more lightweight, and geared toward the application of a far less structured and

190

more organic approach to dealing with the complexities of the diverse data present in the Web.

In order to consume the linked data, centralized systems provide a solution by loading the desired data into a local and centralized storage and processing queries. However, accounting for the decentralized structure of the Semantic Web, such an approach may not always be practically feasible or desired. It suffers from problems of data staleness, siginificant disk space consumption and legal, policy or security issues. In contrast, the federated query answering system realizes the vision of query answering against a federation of disctributed and heterogeneous data sources by splitting the original query into queries that can be answered by the individual data sources and the results are merged by the federator. Based on the work I (and others) have done, if the research challenges highlighted in Section 7.2 can be adequately addressed, I expect that Semantic Web will enable a significant evolutionary step in leading the Web to its full potential.

Bibliography

[1] Yigal Arens, Chun-Nan Hsu, and Craig A. Knoblock. Readings in agents. chapter Query processing in the SIMS information mediator, pages 82–90. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[2] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *CoRR*, abs/1103.5043, 2011.

[3] Donovan Artz and Yolanda Gil. A survey of trust in computer science and the semantic web. *J. Web Sem.*, 5(2):58–71, 2007.

[4] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.

[5] Jie Bao, Giora Slutzki, and Vasant Honavar. A semantic importing approach to knowledge reuse from multiple ontologies. In *AAAI*, pages 1304–1309, 2007.

[6] Cosmin Basca and Abraham Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *ISWC Posters&Demos*, 2010.

[7] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Alberto Corni, R. Guidetti, G. Malvezzi, Michele Melchiori, and Maurizio Vincini. Information integration: The momis project demonstration. In *VLDB*, pages 611–614, 2000.

[8] Deepavali Bhagwat and Neoklis Polyzotis. Searching a file system using inferred semantic links. In *Hypertext*, pages 85–87, 2005.

[9] Alex Borgida and Luciano Serafini. Distributed description logics: Directed domain correspondences in federated information sources. In *On The Move to Meaningful Internet Systems 2002: CoopIS, Doa, and ODBase, volume 2519 of LNCS*, pages 36–53. Springer Verlag, 2002.

[10] Dan Brickley and R.V. Guha. Resource description framework (RDF) schema specification, February 2004.

[11] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[12] Diego Calvanese, Giuseppe De Giacomo, M. Lenzerini, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *In Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS98)*, pages 149–158, 1998.

[13] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: the garlic approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management*

(RIDE-DOM'95), RIDE '95, pages 124–, Washington, DC, USA, 1995. IEEE Computer Society.

[14] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogenous information sources. In *Information Processing Society of Japan (IPSJ 1994)*, 1994.

[15] Ameet Chitnis, Abir Qasem, and Jeff Heflin. Benchmarking reasoners for multi-ontology applications. In *EON*, pages 21–30, 2007.

[16] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[17] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984.

[18] Li Ding and Timothy W. Finin. Boosting semantic web data access using swoogle. In *AAAI*, pages 1604–1605, 2005.

[19] Daniel Dreilinger and Adele E. Howe. Experiences with selecting search engines using metasearch. *ACM Trans. Inf. Syst.*, 15(3):195–222, 1997.

[20] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *SAC*, pages 109–111, 1997.

[21] Ahmed Elmagarmid, Marek Rusinkiewicz, and Amit Sheth, editors. *Management of heterogeneous and autonomous database systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[22] Z. Pan et al. Hawkeye: A practical large scale demonstration of semantic web integration. Technical Report LU-CSE-07-006, Lehigh University, 2007.

[23] Bernardo Cuenca Grau, Bijan Parsia, and Evren Sirin. Working with multiple ontologies on the semantic web. In *International Semantic Web Conference*, pages 620–634, 2004.

[24] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 48–57, New York, NY, USA, 2003. ACM.

[25] Peter Haase and Boris Motik. A mapping system for the integration of owl-dl ontologies. In *IHIS*, pages 9–16, 2005.

[26] Peter Haase, Bjorn Schnizler, Jeen Broekstra, Marc Ehrig, Frank van Harmelen, Maarten Menken, Peter Mika, Michal Plechawski, Pawel Pyszlak, Ronny Siebes, Steffen Staab, and Christoph Tempich. Bibster - a semantics-based bibliographic peer-to-peer system. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2(1), 2011.

[27] Peter Haase and Yimin Wang. A decentralized infrastructure for query answering over distributed ontologies. In *Proceedings of the 2007 ACM symposium*

on Applied computing, SAC '07, pages 1351–1356, New York, NY, USA, 2007. ACM.

[28] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *ICDE*, pages 505–516, 2003.

[29] Andreas Harth and Stefan Decker. Optimized index structures for querying rdf from the web. In *LA-WEB*, pages 71–80, 2005.

[30] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In *WWW 2010: Proceedings of the 19th World Wide Web Conference*, pages 411–420, Raleigh, NC, USA, 2010. Association for Computing Machinery (ACM), ACM.

[31] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. Yars2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, pages 211–224, 2007.

[32] Ian Horrocks and Sergio Tessaris. A conjunctive query language for description logic Aboxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, pages 399–404, 2000.

[33] Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors. *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*. ACM, 2008.

[34] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One, December 2004.

[35] Xing Jiang and Ah-Hwee Tan. Ontosearch: A full-text search engine for the semantic web. In *AAAI*, pages 1325–1330, 2006.

[36] Eser Kandogan, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and Huaiyu Zhu. Avatar semantic search: a database approach to information retrieval. In *SIGMOD Conference*, pages 790–792, 2006.

[37] David Karger and MC Schraefel. The pathetic fallacy of rdf. Position Paper for SWUI06, 2006.

[38] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.

[39] Günter Ladwig and Thanh Tran. Linked data query processing strategies. In *International Semantic Web Conference (1)*, pages 453–469, 2010.

[40] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, February 1999.

[41] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

BIBLIOGRAPHY

[42] Yingjie Li and Jeff Heflin. Query optimization for ontology-based information integration. In *CIKM*, pages 1369–1372, 2010.

[43] Yingjie Li and Jeff Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *International Semantic Web Conference (1)*, pages 502–517, 2010.

[44] Yingjie Li and Jeff Heflin. Handling cyclic axioms in dynamic, web-scale knowledge bases. In *The 2011 Workshop on Scalable Semantic Web Knowledge Base Systems, ISWC2011*, 2011.

[45] Yingjie Li, Abir Qasem, and Jeff Heflin. A scalable indexing mechanism for ontology-based information integration. In *Web Intelligence*, pages 328–331, 2010.

[46] Yingjie Li, Yang Yu, and Jeff Heflin. A multi-ontology synthetic benchmark for the semantic web. In *The 1st International Workshop on Evaluation of Semantic Technologies (IWEST2010), ISWC2010*, 2010.

[47] Yingjie Li, Yang Yu, and Jeff Heflin. Evaluating reasoners under realistic semantic web conditions. In *The OWL Reasoner Evaluation Workshop 2012, IJCAI 2012*, 2012.

[48] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Continuous RDF query processing over DHTs. In *ISWC/ASWC*, pages 324–339, 2007.

[49] Vanessa Lopez, Enrico Motta, and Victoria S. Uren. Poweraqua: Fishing the semantic web. In *ESWC*, pages 393–410, 2006.

[50] Vanessa Lopez, Victoria S. Uren, Enrico Motta, and Michele Pasin. Aqualog: An ontology-driven question answering system for organizational semantic intranets. *J. Web Sem.*, 5(2):72–105, 2007.

[51] Weiyi Meng, King-Lup Liu, Clement T. Yu, Xiaodong Wang, Yuhsi Chang, and Naphtali Rishe. Determining text databases to search in the internet. In *VLDB*, pages 14–25, 1998.

[52] B. Motik and U. Sattler. A comparison of reasoning techniques for querying large description logic aboxes. In *Proceedings of 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 227–241, Phnom Penh, Cambodia, 2006.

[53] Boris Motik. *Reasoning in description logics using resolution and deductive databases*. PhD thesis, 2006.

[54] Boris Motik, Bernardo Cuenca Grau, and I. Horrocks. OWL 2 Web ontology language profiles. Recommendation, October 2009. http://www.w3.org/TR/owl-profiles/.

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

[56] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello. Sindice.com: a document-oriented lookup index for open linked data. *IJMSO*, 3(1):37–52, 2008.

BIBLIOGRAPHY

[57] Aris M. Ouksel and Amit P. Sheth. Semantic interoperability in global information systems: A brief introduction to the research area and the special section. *SIGMOD Record*, 28(1):5–12, 1999.

[58] Chintan Patel, Kaustubh Supekar, Yugyung Lee, and E. K. Park. Ontokhoj: A semantic web portal for ontology searching, ranking and classification. In *In Proc. 5th ACM Int. Workshop on Web Information and Data Management*, pages 58–61, 2003.

[59] P. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language semantics and abstract syntax. Recommendation, February 2004. http://www.w3.org/TR/owl-semantics/.

[60] Rachel Pottinger and Alon Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.

[61] A. Qasem. *Query-based Selection and Integration of Semantic Web Data Sources*. Dissertation, Computer Science and Engineering, Lehigh University, 2009.

[62] Abir Qasem, Dimitre A. Dimitrov, and Jeff Heflin. Efficient selection and integration of data sources for answering semantic web queries. In *ICSC*, pages 245–252, 2008.

[63] Abir Qasem, Dimitre A. Dimitrov, and Jeff Heflin. Goal node search for semantic web source selection. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 1:566–569, 2008.

[64] M. Ross R. Kimball. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, Second Edition.* John Wiley and Sons, Inc., 2002.

[65] Marie-Christine Rousset, Philippe Adjiman, Philippe Chatalic, François Goasdoué, and Laurent Simon. Somewhere in the semantic web. In *SOFSEM*, pages 84–99, 2006.

[66] Edna Ruckhaus, Maria Esther Vidal, Eduardo Ruiz, and Javier Sierra. A cost-based optimizer for sparql queries. In *The Workshop of Query Optimization for the Semantic Web*, 2007.

[67] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[68] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of sparql query optimization. In *ICDT*, pages 4–33, 2010.

[69] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: A federation layer for distributed query processing on linked open data. In *ESWC (2)*, pages 481–486, 2011.

[70] Luciano Serafini and Andrei Tamilin. Drago: Distributed reasoning architecture for the semantic web. In *ESWC*, pages 361–376, 2005.

[71] P Shvaiko and J Euzenat. A survey of schema-based matching approaches. *Database*, 3730(October 2004):146–171, 2005.

[72] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[73] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL web ontology language guide. Recommendation, February 2004. http://www.w3.org/TR/owl-guide/.

[74] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM.

[75] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 631–639, New York, NY, USA, 2004. ACM.

[76] Geoffrey G. Towell, Ellen M. Voorhees, Narendra Kumar Gupta, and Ben Johnson-Laird. Learning collection fusion strategies for information retrieval. In *ICML*, pages 540–548, 1995.

[77] Thanh Tran, Haofen Wang, and Peter Haase. Hermes: Data web search on a pay-as-you-go integration infrastructure. *J. Web Sem.*, 7(3):189–203, 2009.

[78] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. Grin: A graph based rdf index. In *AAAI*, pages 1465–1470, 2007.

[79] Taowei David Wang, Bijan Parsia, and James A. Hendler. A survey of the web ontology landscape. In *International Semantic Web Conference*, pages 682–694, 2006.

[80] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[81] Joel L. Wolf, Balakrishna R. Iyer, Krishna R. Pattipati, and John Turek. Optimal buffer partitioning for the nested block join algorithm. In *ICDE*, pages 510–519, 1991.

[82] David Wood, Paul Gearon, and Tom Adams. Kowari: A Platform for Semantic Web Storage and Analysis. In *XTech*, Amsterdam, 2005.

[83] Clement T. Yu, Weiyi Meng, Wensheng Wu, and King-Lup Liu. Efficient and effective metasearch for text databases incorporating linkages among documents. In *SIGMOD Conference*, pages 187–198, 2001.

[84] Budi Yuwono and Dik Lun Lee. Server ranking for distributed text retrieval systems on the internet. In *DASFAA*, pages 41–50, 1997.

Vita

Yingjie Li has received both his Undergraduate and Master Degrees in Department of Computer Science from Taiyuan University of Technology, Taiyuan, China. He was twice selected the Shanxi Province Outstanding Gaduate Student in 2004 and 2007 by Taiyuan University of Technology and Shanxi Province in China. He has been both a research assistant and a teaching assistant with the Department of Computer Science and Engineering at Lehigh University, Pennsylvania while completing his Ph.D. He was a Research Scientist intern of Pitney Bowes and Samsung Information System America in 2011 and 2012. His research focuses on finding ways of addressing the scalability and the heterogeneity challenges in developing Semantic Web query answering systems. He will be joining JP Morgan Chase and Corporate as a Data Architect.